Computer Science NEA

Project Title: Book recommendation website

Name: Reuben James

**Contents**

[**Analysis 8**](#_nnq4eje5ti0)

[Introduction 8](#_1fob9te)

[Initial Project Aims 9](#_3znysh7)

[Identification of Requirements 9](#_2et92p0)

[Existing Systems 9](#_tyjcwt)

[Goodreads 9](#_3dy6vkm)

[LibraryThing 11](#_4d34og8)

[End Users 12](#_17dp8vu)

[Interfacing with the system 12](#_9gljhmmnxv2y)

[Reading lists 14](#_hhgi2x31x8wo)

[Recommendations 16](#_mmqd6ehulwm5)

[Reading Diaries 19](#_21nemj18udnp)

[Further Functionality 21](#_jx220vz2o2m1)

[Added Functionality 21](#_cwam2mtbv7e3)

[Project Brief 23](#_7wf8mxcoasng)

[Project Requirements 23](#_3whwml4)

[Project Acceptable Limitations 27](#_2bn6wsx)

[Background Research 27](#_qsh70q)

[Website 28](#_3as4poj)

[Web Servers and Web Hosting 28](#_1pxezwc)

[Apache 29](#_49x2ik5)

[NGINX 29](#_ye6veveuzc3x)

[Lighttpd 30](#_2p2csry)

[PythonAnywhere.com 31](#_147n2zr)

[Dynamic Content Generation 31](#_zd9lk48j6ufp)

[Common Gateway Interface 32](#_3o7alnk)

[CGI 33](#_23ckvvd)

[FastCGI 34](#_ihv636)

[WSGI 35](#_hjcilvfv83x1)

[Middleware 36](#_dzyefg2gvta8)

[Protocols 36](#_1hmsyys)

[HTTP 37](#_vx1227)

[HTTPS 40](#_4f1mdlm)

[HTTP Requests 41](#_eb2oyprezekk)

[JavaScript 43](#_rp90gsaururm)

[File structure 47](#_3tbugp1)

[Website and Domain Name 48](#_46r0co2)

[Database 49](#_3l18frh)

[Object-Based Database Management System 50](#_206ipza)

[Relational Database Management System 51](#_2zbgiuw)

[MySQL Connector 51](#_tbini7wl8u0c)

[Reading diaries 52](#_e3x7u56i2xie)

[Searching the database 53](#_3ygebqi)

[Exact string matching 53](#_2dlolyb)

[Bag Of Words (BoW) 54](#_sqyw64)

[Google Universal Sentence Encoder (Google USE) 55](#_3cqmetx)

[Term Frequency – Inverse Document Frequency (TF-IDF) 56](#_4bvk7pj)

[Data pre-processing 56](#_2r0uhxc)

[Tokenisation 57](#_1664s55)

[Data cleaning 57](#_3q5sasy)

[Stemming and Lemmatisation 57](#_25b2l0r)

[Stop words. 58](#_1jlao46)

[Contractions, slang and numbers. 59](#_43ky6rz)

[Spelling mistakes 59](#_2iq8gzs)

[Term Frequency 60](#_xvir7l)

[Inverse Document Frequency 60](#_3hv69ve)

[Term Frequency – Inverse Document Frequency (TF-IDF) 61](#_1x0gk37)

[Vectorising the search query 61](#_4h042r0)

[Recommendations 62](#_1baon6m)

[Content-Based filtering 62](#_r7p0s495hklr)

[Collaborative Filtering and Matrix Factorisation 64](#_udyh6qowztqz)

[**Documented Design 66**](#_8dhbqb5krq)

[Backend 66](#_861edviwm5j9)

[MySQL Database 66](#_wnsb6mw5kay4)

[Table definitions 67](#_v14jee7d2n2o)

[Database queries 76](#_62wnilfo78ql)

[Accounts 76](#_6c7miunps2w3)

[Authors 77](#_vytqmqu5ugie)

[Books 1](#_p7ilw2uzt2io)

[Diaries 1](#_190ytlnptznc)

[Genres 1](#_t2po2hzgmksh)

[Database Searching 1](#_2zpyh1bqknnt)

[Reading lists 1](#_ca737xnlog5t)

[Recommendations 1](#_nb1uf6ff9t7)

[File system 1](#_7icz929etr9k)

[Handlers and Middleware 1](#_mpri6qwv7h1z)

[Project Maintenance 1](#_j1kd7dd0rm27)

[Lighttpd Configuration 1](#_kzdxsfwxe69p)

[Enabled modules 1](#_aql3jg3p420h)

[Class diagrams and definitions 1](#_1msqxw74j6sp)

[Class diagrams 1](#_er9k9q1unwql)

[Class Definitions 1](#_mr5edok6jjbj)

[Session IDs 1](#_w41p8gqj0v01)

[Books 1](#_84i1jyf7881b)

[Information Retrieval 1](#_pln7zce94l6y)

[Getting clean document names 1](#_7uti6ch7nwxe)

[Loading documents 1](#_hu7jb1487ky)

[Generating TF values 1](#_wg1ridul136)

[Generating IDF value 1](#_pxv61p280evi)

[Generating TF-IDF values 1](#_29fj7goia5nq)

[Search 1](#_4078ou6atcjl)

[Recommendations 1](#_2zg8kh1y1j4j)

[Fitting the model 1](#_5vvk2joza4wl)

[Generating review matrices 1](#_t9ogn0y0st69)

[Generating recommendations 1](#_8a6jywizjxqm)

[Calculating recommendation certainty 1](#_wch6qelstm8)

[Configuration 1](#_cuf4jy9777m)

[Existing configuration methods 1](#_b1m7j0rb8eae)

[Project configuration specification 1](#_jv80i7opoaeu)

[Configurable Parameters 1](#_2breabz5svrm)

[MySQL 1](#_rp4z9d1bhbmf)

[Passwords 1](#_cfbxl379altl)

[Home 1](#_mp5wqf7pu8kj)

[Recommendations 1](#_kzk6d6ps0uah)

[Searching 1](#_whubr27b5q35)

[Miscellaneous 1](#_k3u45jpo570y)

[Design 1](#_3czilwu1ohs5)

[Exceptions 1](#_mwy0gwejdmam)

[Regular Expressions 1](#_eg4om9varv1k)

[Processing and data access 1](#_uqagwic186c3)

[Class definition 1](#_lzez29a6nsd0)

[Frontend 1](#_yqwe3xq3cjb7)

[Website 1](#_cphvbkqvm70z)

[Site hierarchy 1](#_10j1j88e6bsj)

[UI design 1](#_z8y734sfr9em)

[Home Page 1](#_pl859m14r5lf)

[Recommendations Page 1](#_iwx10gmx04ak)

[Diary Page 1](#_c9tn9hqy243i)

[My Books Page 1](#_1spi3qlhcyfq)

[Browse Page 1](#_ymwfmntdi4ky)

[Search Page 1](#_btsa33unq5i4)

[Book About Pages 1](#_ecl8daytge2q)

[Genre About Pages 1](#_6y8es8243sd4)

[Author About Pages 1](#_y9tscaxkhzpm)

[Sign In and Sign Up Pages 1](#_gdsq2o5rdmsv)

[AJAX Requests 1](#_d8fow95snf5q)

[GET requests 1](#_6jfz917rcp7y)

[POST requests 1](#_cnp3ynps9pfe)

[**Technical Solution 1**](#_awh3iybf0qv)

[Requirements 1](#_291eojszsp1e)

[Installation 1](#_txq9e6yj99s7)

[Software installation 1](#_j9k3szx2guku)

[Python package installation 1](#_7uvgsrvcojh)

[MySQL Database setup 1](#_8pczz3qygk4j)

[Lighttpd setup 1](#_97ryw2dwe38g)

[Scheduled Jobs 1](#_tevvnlah87hw)

[Configuration 1](#_pkujleyrair8)

[Test data 1](#_c0vnr79gilks)

[Web server control 1](#_amadobxqshrk)

[Objectives 1](#_mguezxd0otun)

[Backend 1](#_5y6jpxsf8sfl)

[Utilities 1](#_805eunilizex)

[Database Connection Module (./backend/mysql\_handler.py) 1](#_2pwnqu4n68hr)

[Configuration Module (./backend/configuration.py) 1](#_v5rh1jqj2enb)

[Searching algorithms module (./backend/searching\_algorithms.py) 1](#_r2y7os47cnc)

[Data structures module (./backend/data\_structures.py) 1](#_i0bwg7njtoxr)

[Queue 1](#_i0o17fbn0ysm)

[Priority queue 1](#_t1sqe8ispkp5)

[Stack 1](#_7nanobd5k9lj)

[Binary Tree 1](#_4r9vl3x3c32b)

[Environ variable manipulation module (./backend/envrion\_manipulation.py) 1](#_cyl3zro103wu)

[Logging module (./backend/logger.py) 1](#_20chft1ps22a)

[Machine learning utilities module (./backend/ml\_utilities.py) 1](#_tvdu00cc68xt)

[Components Module 1](#_a9erdaslnevn)

[Accounts (./backend/components/accounts.py) 1](#_336d4knhkrd0)

[Accounts class 1](#_qb22sp7co66t)

[Sessions class 1](#_r7mg5mjq5vpn)

[Authors (./backend/components/authors.py) 1](#_5n5ezdb4b3v8)

[Books (./backend/components/books.py) 1](#_78hkojwmxv3d)

[Diaries (./backend/components/diaries.py) 1](#_t9bjqqw5zzzt)

[Genres (./backend/components/genres.py) 1](#_cbp61eafv1uk)

[Information retrieval and database searching (./backend/components/information\_retrieval.py) 1](#_ihslkthnt3ti)

[Reading lists (./backend/components/reading\_lists.py) 1](#_7h30j4of4au5)

[Recommendations (./backend/components/recommendations.py) 1](#_rterxuuecl8k)

[Flup server file (./backend/flup.server.fcgi) 1](#_tykyhg15y6yt)

[Database maintenance (./backend/maintenance.py) 1](#_buft6p2eutaw)

[WSGI Script (./backend/wsgi.py) 1](#_37yqy9s3cw4d)

[Handlers and Middleware 1](#_o8l7rugxyok0)

[Handler Class 1](#_nzhudi4ieom1)

[App/Middleware creation 1](#_t73c9x1n4h3v)

[Error Handler 1](#_i1up0z2gqzru)

[Handler classes 1](#_fohic2ydik7o)

[Accounts Handler 1](#_hwbnv8uuqfqq)

[Reading lists Handler (My Books) 1](#_cvdtn0huha7t)

[Genre Handler 1](#_bm0rzlu25jdy)

[Books Handler 1](#_sbsysrflcm2u)

[Authors Handler 1](#_s3gzd71jq9ag)

[Diaries Handler 1](#_yyuwg1pvy1k8)

[Home Handler 1](#_nl1cxxsslumn)

[Recommendations Handler 1](#_p82f16wzx7ar)

[Searching Handler 1](#_lmooo6hycp7x)

[Frontend 1](#_7hha2etan9r2)

[Javascript 1](#_ny49a13a7arf)

[Utility Functions 1](#_wyi82kzyd03)

[Page switching/loading 1](#_865fukr2fqlh)

[Account Management 1](#_uiuku4px3xv3)

[Sign Up popup 1](#_8um394rof227)

[Sign In popup 1](#_p71k1p22evhz)

[Sign Out 1](#_8r7iwh7ui6a8)

[My Books (reading list) pages 1](#_gksno9s64j1l)

[Genre about pages 1](#_2vaxzilth7fa)

[Book about pages 1](#_ein3f8qmjck5)

[Author about pages 1](#_h6k8jeujwxsh)

[Diary page 1](#_4qt5x5r6ubhx)

[Home page 1](#_3usezwg4cotl)

[Recommendations Page 1](#_x3jvicj9639u)

[Database search 1](#_wg6derp4kgdh)

[Browse page 1](#_la4rs5p9mw6f)

[Proforma 1](#_3352b7ydzy2h)

[**Testing 1**](#_94d44ib069o6)

[Backend 1](#_rxg8k3sp75s)

[Frontend (including wsgi.py) 1](#_coraqknn66l2)

[**Evaluation 1**](#_dmi6m13yve17)

[**Bibliography 1**](#_13lluambi3n0)

[**Appendix 1**](#_34ynjeec2b15)

[Appendix A 1](#_48pi1tg)

[Appendix B 1](#_2nusc19)

[Appendix C 1](#_zd2wf9ydy7kv)

# Analysis

## Introduction

In this current age, with technology intrinsically linked to our daily lives, many different online platforms fulfil many of our daily requirements, including Audible, Spotify, BBC iPlayer, and other streaming services, many of which include recommendation systems. Many other recommendation systems exist outside of these platforms, including within shopping platforms such as Amazon. Remarkably, amongst these many recommendation services making our daily lives easier and more efficient, there is not a fully featured system for all the services that are often used by avid readers.

The number of genres, authors and books that are widely available can be overwhelming, and this diversity of options can make it difficult to discover new authors and genres, especially without any sign of the author’s writing style, genre, character, and plot development, or any of the other nuances that can significantly affect the enjoyment of a book. A successful recommendation system should be able to tackle both issues and, while unlikely to supply correct recommendations one hundred per cent of the time, it will still improve these issues.

Furthermore, keeping a reading diary is beneficial, increasing retention, forcing the reader to contemplate the material, read and articulate their thoughts, and supplying a time and space for writing practice (Donovan, 2019) For many readers, despite these documented benefits, it can be difficult to complete these tasks, resulting in many readers not completing these records, due to the time-consuming nature of the task.

This proposal focuses primarily on fiction over reference books and only for physical book copies. To reduce complexity, support for E-books and audiobooks has been excluded from this proposal, as they would require more databases to store information on them, as not all physical books are available as E-books and audiobooks. Furthermore, with audiobooks, other factors would need to be considered in the recommendation system, such as the narrator, and given that they are available from a range of sources with different narrators, this adds a significant amount of complexity. For this reason, this has also been excluded, but would be something that could be considered in the future to improve the service. By not including support for audiobooks and E-books, a significant proportion of the market may not be able to use the system, as nearly two-thirds (65%) of people who buy books, E-books, or audiobooks said they like physical books best, one in five (21%) preferred E-books and just 14% opted for audiobooks (Williams, 2022). For this reason, it should be considered if the system was widely accepted.

## Initial Project Aims

For the development of this project to be performed successfully and meet all the features that are listed above, a list of achievable and measurable aims is needed. This ensures that the solution meets the requirements set out in the project brief. These requirements were those provided to the end users of this project, which may then develop and be augmented depending on their responses to help tailor this project to their needs more effectively. It is important that these are not the final requirements of the system. For that reason, they are broad, and are difficult to measure. This is addressed in the updated project requirements.

* It must have a simple, easy-to-use interface.
* It must be able to supply recommendations for further books.
* It must have a facility to allow users to record books that they have read, are going to read, and currently reading.
* It must supply a way for the users to record their thoughts and feelings about all the books that they have read in a reading diary.
* There must be a way for users to log in.

## Identification of Requirements

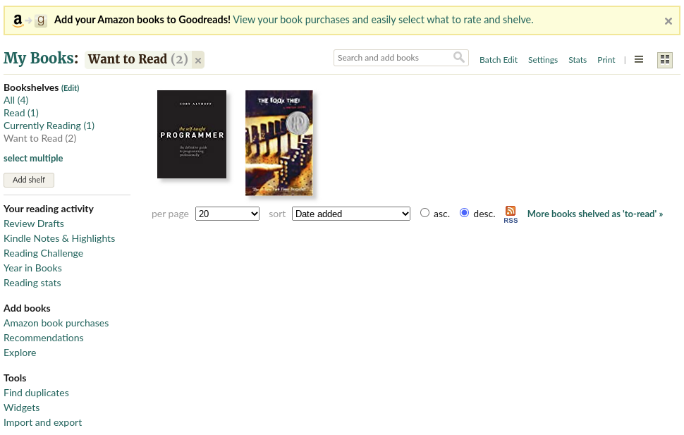
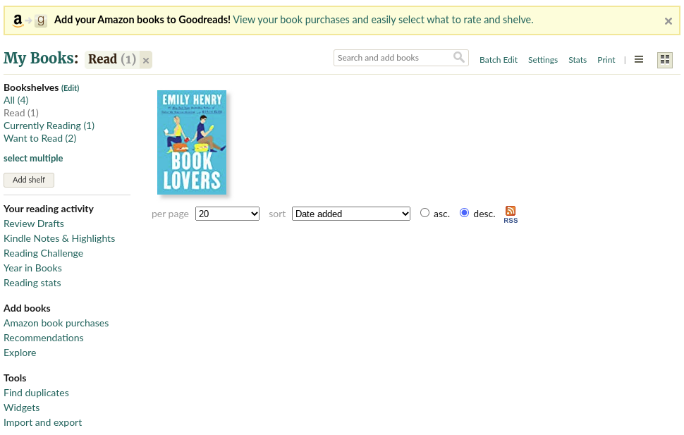
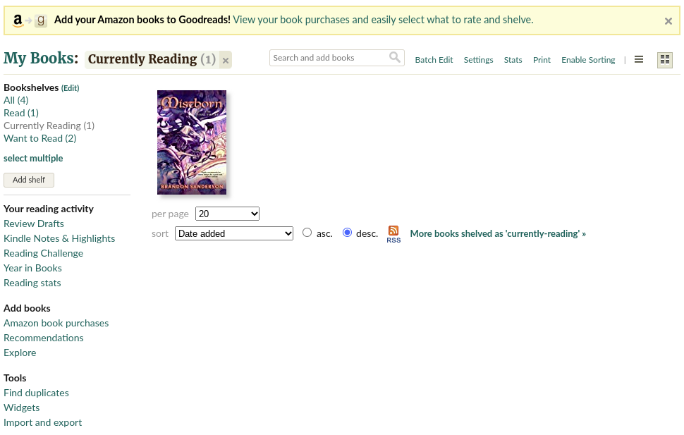
This system is designed to be used by many different users, as it will be either a mobile application, desktop application or website, and is, therefore, easy to access and use. This results in a large user base, and therefore it can be difficult to design a system that can fulfil all their requirements, and there will be compromises made to ensure that it is achievable, and not overly complicated. Furthermore, some of the features of this proposal are performed by other, pre-existing systems. Consequently, this section will also look at pre-existing systems.

### Existing Systems

While there are not any solutions that supply all the features that this proposal includes, there are other applications that perform parts of this system. The most notable of these is *Goodreads* and *LibraryThing*. These websites both have a large user base and include some features that are outside of the scope of this project, although they would make a good extension of the system in the long-term future. However, they both have exceptionally good recommendation systems, and alternative solutions to the reading lists that are in this project.

#### Goodreads

(Goodreads, 2018) is “*the world’s largest site for readers and book recommendations*”. It supplies some of the functionality that this system is proposing recommendations and bookshelves. Bookshelves are a similar concept to reading lists in this application, supplying a way to record the books that the user is currently reading, has previously read, and would like to read in the future, see Figure 1. The bookshelves are not that easy to find, as they are not in the main website navigation, and must be accessed through the home page, or My Books, and as it is not named the same, can be difficult to find. However, it is clear what items are included in them. Furthermore, the ability to add new shelves is useful. The recommendations that are provided by Goodreads (2018) are accurate and are tailored to the user’s interests. Recommendations are in the same location as bookshelves, under My Books, and are therefore intuitive, following a similar layout. Overall, the website is moderately easy to navigate, however, for a first-time user it can be complicated to learn. Compared to many other websites now, it looks very dated. It does offer features that this proposal does not include, for example, daily reading challenges, asking the author, quotes, and discussions, and hence it has something for most people.

Graphical user interface, application, Word

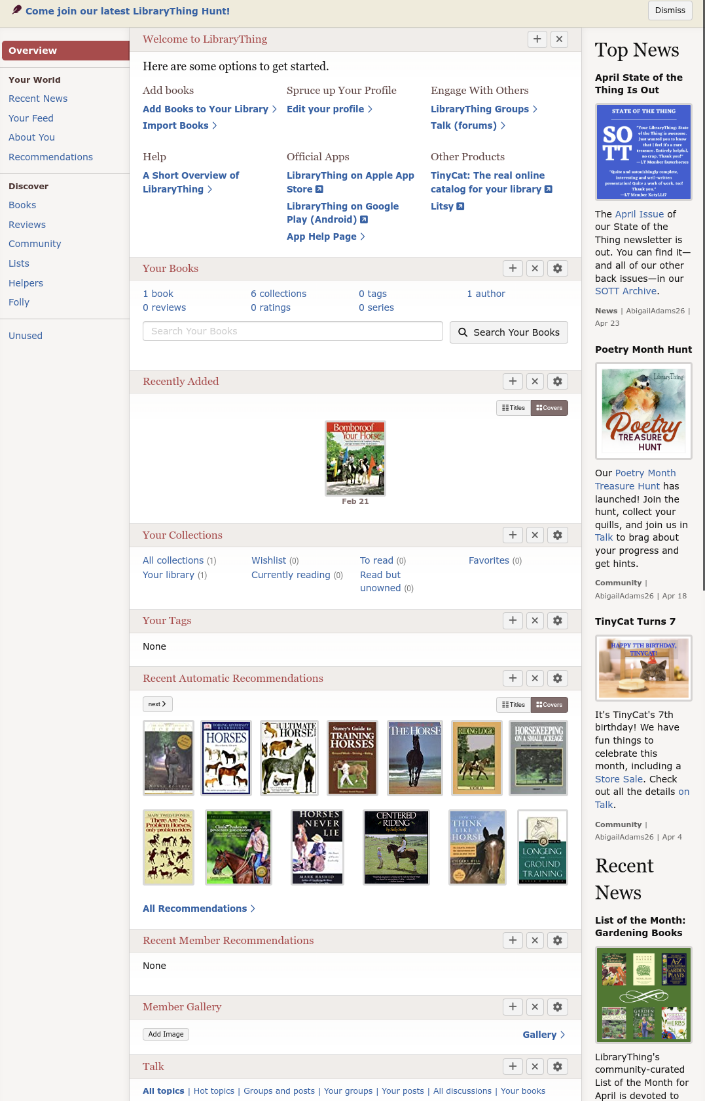
Description automatically generated

*Figure 1 - Example Goodreads bookshelves*

This website does not supply any form of reading diary, however. This leaves a gap in the market, which it does not fill. Furthermore, a system that has a more modern and simpler user interface, which is easier to navigate, could offer a compelling alternative to it, especially with a reading diary included.

#### LibraryThing

LibraryThing (2011) is another alternative to Goodreads (2018). It is designed to catalogue books and connect people with similar books. This enables it to provide recommendations for other books. Like Goodreads, it provides collections, which allow the user to save what they want to read, currently reading, read but unowned and favourite books. It also allows users to create their own collections. This is a useful feature to have, but the added complexity that comes with it may not be worthwhile. Using the user’s catalogues and other information, it generates good recommendations for other books to read, and with many books stored on the database, it can give diverse recommendations. To achieve this, it allows users to create new book entries. This again would be a good feature to have, but with the added complexity of checking the validity, and removing duplicates, may not be worthwhile. The user interface for the website is very convoluted and is difficult to effectively find each part, as shown in Figure 1. However, it provides many other functions, such as member recommendations, galleries, forums, and connections, which does improve its functionality.



*Figure 2 - Example LibraryThing home page*

A way to improve this website would be to make the features much more intuitive and easier to navigate the website. Updating the user interface to improve clarity would also improve the website. Finally, it also does not supply the user with a reading diary. Consequently, this proposal does fill a gap in the market, and, if done well, and extended to include the features that these websites provide, could be competitive.

### End Users

There is a significant gap in the market for software and web-based applications that can successfully recommend users' books and supply a way to save what they want to read, are currently reading and what they have read in the past, whilst also supplying a reading diary for their reading, which is vital because of the reasons outlined previously. If this project is successful, it will have a large user base, as it would be publicly available as a mobile application, a desktop application, or a website. To help gain an understanding of what this project should be able to do, I asked a choice of interested people to supply feedback and insights as to what was needed, the questions for which are included in Appendix A. Through these questions, I have gained a good insight into what is needed, and by using these, I will be able to significantly augment the initial aims to be more beneficial to the end users of the system.

This project targets people who enjoy reading, struggle to find new books that they enjoy reading, and want a way to record their reading history to help understand what they have read, or any combination of these. Furthermore, they want a simple, unified solution that can supply all the features that they require, to avoid the need of using multiple tools, including books and websites. This covers many users, and while they may all want to have different features within the system, there will be a few features that are desired by most, if not all the users. These are what will be included in the project, and the lesser desired ones are to be excluded to help ensure that the project is achievable in the available time.

#### Interfacing with the system

With various options for the users to interact with the system, it is vital that a relevant user interface is chosen, so that the users can interact with the system easily and effectively. Furthermore, the chosen interface may also affect the accessibility of the project for the users, and while there are different ways to avoid this, they all increase the complexity. It also may affect how complex the system is to design, with a command line interface (CLI) being simple to implement, and mobile applications being more complex. This does not directly affect the users, however it must be taken into consideration when choosing the interface, as lowering the complexity of the system will help make the project more achievable.

Through the end user forms, it was made clear that the users would prefer to have the system as a website, closely followed by a mobile application, see Figure 3. Using a website allows most users to be able to use the website, regardless of the computer or device that they are using, a sentiment that is also expressed by Tim James, who also mentions that it removes the need for specific software to be installed. This makes it easier to develop, as it offers a level of system independence, as it would have to be designed differently for different operating systems, including android, IOS, Linux and Windows. Only developing the software for one of these excludes users from using the system, which should be avoided. A website, however, just requires a browser, which using modern browsers, means the interface is compatible across different devices.

Chart, pie chart

Description automatically generated

*Figure 3 - What should the system interface be?*

Using a mobile application would allow for the user to view their recommendations and information without an internet connection. Any changes that were then made offline could then be synchronised to the database once an internet connection is obtained. This could make the user experience more enjoyable and make it more accessible. However, this then significantly increases the complexity, and given that the minority wanted a mobile application, a website is the best method. An extension to this project, if the website were successful, would be to create a partner mobile or desktop application, but this is outside of the scope of this project.

There is also a wide range of website styles, such as modern and minimalistic, with 100% of the users opting for a modern style for the system, see Figure 4. This style can help make information easier to find and make it easier to navigate. Furthermore, it follows the trend with website designs, with most websites, such as Waterstones and Amazon, using modern styles. Having these existing websites with similar designs could be used to help influence the design of the UI. Through the form, see Figure 5, it was expressed that the most popular colour theme was white, black, navy, grey, and red, with 50% of users preferring it. However, with 25% with no preference, and 25% with white, blue, and grey, there is not a rigid colour theme, and can be changed or augmented slightly if needed.

Chart, pie chart

Description automatically generated

*Figure 4 - What style should the website be?*

Chart, pie chart

Description automatically generated

*Figure 5 - What should the website's theme colours be?*

#### Reading lists

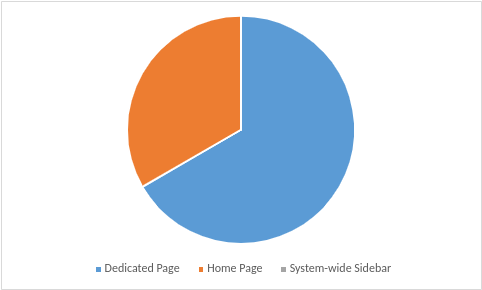
Within the reading list section of the website, there is a wealth of information that could be displayed for each entry, including author, synopsis, genre, cover image, and other information. Most, if not all, of the pieces of information about the book could be displayed, however, this would lead to the page being very cluttered and difficult to use. For this reason, it is essential that only the relevant information should be included with each entry. The end users for this project expressed that the date added, author, genre, and date finished (if applicable) are the most important attributes to have, see Figure 6. In addition, the book title must be included, as it would be difficult to figure out what the entry is referring to. Other information, such as synopsis, cover image, and the length of books are less in demand and can therefore be included as an extension. However, some information about the books, such as how many people are reading each item, would be unnecessary, and will not be included.

Chart, bar chart

Description automatically generated

*Figure 6 - What should be included in the reading list?*

These reading lists should be easily accessible within the website. Sixty-seven percent of the end users suggested that they should be made available on a dedicated page, see Figure 7. However, it has also suggested that they could be on the home page, which would give faster access to the entries. Having a full list, with the relevant details, as outlined previously, on the home page, and a second, shortened list on the home page would allow for faster access, seems to be a satisfactory solution for ease of use and aesthetics. This is further supported by one of the end users, saying that it “*might be nice to have a small section on the home page to display this information*”.



*Figure 7 - Where should reading lists be found on the website?*

#### Recommendations

The recommendations are the most significant part of this project, both in the importance to the users and the complexity of the implementation, see Figure 8, which were calculated by taking the ratings out of ten that were given for each feature by the users, through the form, and taking the geometric mean of them. For this reason, it is vital that the users can access recommendations easily, and therefore should be very accessible.

Chart, bar chart

Description automatically generated

*Figure 8 - How important is each website feature?*

The need for accessibility makes it important to make the recommendations available in a location that is accessible for the users. Many of the end users wanted the recommendations to be on the home page, and 25% wanted it on a dedicated page and system wide sidebar, see Figure 9. Including all the relevant information about each recommendation on the home page would lead to it being very cluttered, and difficult to understand. However, by including a significantly simplified version of each recommendation on the home page, such as title, author, and cover image, would help avoid this. Having a separate, dedicated page for the full information about each entry, and the simple version on the home page, it is then easy to find all the relevant information about each recommendation, whilst still being simple and accessible for the users. Furthermore, this method satisfies all the users who wanted them available on the home page, and those who wanted them on a separate page, so this is the approach that should be taken. With regards to having a sidebar that was on all pages of the website, this seems impractical as it takes up space on each page and makes it harder to navigate. Furthermore, already being available on the home page and on a dedicated recommendations page, makes having a sidebar redundant, and since this is a minority, can be excluded.

Chart, pie chart

Description automatically generated

*Figure 9 - Where should recommendations be found on the website?*

The detailed recommendations on the dedicated recommendations page allows for a large amount of information to be displayed to the user about each entry. However, having the minimum required information that is relevant to the user displayed for each of these entries, will help make it easier to find the desired information. The information that is needed is like that in the reading lists, specifically the want to read list. This means that it should include the date the recommendation was made, author, genre, cover image, synopsis, length of the book and the book title.

Since the recommendations made to the users are like their tastes, it is likely that they are going to want to add them to their want to read list. 100% of the end users thought that this is applicable, see Figure 10. This is an important inclusion, as it would be impractical for end users to find the books manually, then add them to the reading list, so should be included.

Chart, pie chart

Description automatically generated

*Figure 10 - Should recommendations have the option to add it to the user's reading list?*

Furthermore, when users first join, the system is unaware of what they prefer to read. For this reason, the users are likely to need to give some initial information, to improve the accuracy of the recommendations. There are multiple ways that this could be achieved, for example, asking for a preferred list of genres and a preferred list of authors. The end users felt that it was most appropriate to ask for a preferred list of genres as well as a preferred list of authors, see Figure 11. It was also clear that not asking was an inappropriate method. For this reason, when first signing up, users should be prompted to supply the preferred authors and genres.

Chart, bar chart

Description automatically generated

*Figure 11 - How should initial information about the user's tastes be gathered?*

#### Reading Diaries

The final feature that was initially proposed for this project was for users to have a reading diary, where the users could record their thoughts and feelings about the books that they have read, or the part of the book that they have read recently. There are many different options for the data that could be included in this reading diary, the users wanting to have an overall rating, a section for thoughts and feelings, and whether they would want to read it again, see Figure 12. Others also thought that having a way of rating individual aspects of the book, such as character development and plot progression may also be right, however as this is a minority, it is to be left as a project extension only.

Chart, bar chart

Description automatically generated

*Figure 12 - What should be included in a diary entry?*

The information that is to be included in these diary entries, at least for the completed books, is like the information that would be left as a book review. For this reason, they could have the choice to post the information publicly as a book review. Most of the end users thought that this would be a useful feature, see Figure 13. This must be made an optional action, as not all users would want to have their thoughts about the books made public. Having spoken with someone who disagreed with the inclusion of this feature, agreed that it can be included, if it is made optional, which was an oversight in the creation of the form, as it was not clearly explained.

Chart, pie chart

Description automatically generated

*Figure 13 - Should there be an option to make diary entries public as reviews?*

Like any other aspect of this project, the user’s diary entries need to be accessible. Most of the users wanted to have them on a separate, dedicated page, see Figure 14. A smaller proportion wanted to have them available with the reading lists. However, to make this work effectively, there would need to be a link to all diary entries about that book, which significantly increases the complexity as there then also needs to be a way to sort entries by book. For this reason, the diary entries will be found on a dedicated page only, and as it is the minority that wanted this, it is not an issue.

Chart, pie chart

Description automatically generated

*Figure 14 - Where should diary entries be found on the website?*

#### Further Functionality

As mentioned previously, speaking with end users also gave additional features and functionality that is required for the project, often to make the site easier to use and navigate, as well as adding some additional quality-of-life features, see Figure 15. From this, the system should give author’s public profiles, each page about a specific book should include a link to websites to buy the book, display the price of the item, and give the ability for users to leave reviews. Some users also wanted to be able to follow authors and users, which requires users to have public profiles, which adds complexity, with limited gain to the users, given that none marked it as useful. However, given that author’s already have public profiles, it is reasonable to add a way of following them. Similarly, each book having its own corresponding about page was not marked as particularly useful, however, for many of the aforementioned features, it is needed, so will be included.

Chart, bar chart

Description automatically generated

*Figure 15 - What other features should be included?*

### Added Functionality

While the end user form outlined the major functionality of the system, there were details that were excluded from it to keep it concise, and more likely to have more complete responses. This meant that some trivial details for features that would make the system easier to use were excluded. This brief section outlines these, and similarly has influenced the updated project aims.

Every book that is stored in the database for the system will have its own dedicated webpage, with all the information that is stored about that book on it, including the reviews. Each recommendation entry, any reading list entries, and any diary entry has extremely limited information about the corresponding book. While this is satisfactory to gain a basic understanding, any more detailed information is then excluded. However, a direct link to the corresponding books about page would supply a straightforward way to gain more information, without making the page more cluttered and harder to understand. It also does not significantly increase the complexity, and for this reason, is an important feature to include.

Entries into reading lists are likely to need to be moved to different lists at some point. There must be a quick and simple method for users to do this. For example, a user may have a book that they would like to read, then would start reading the book, and at some point, would finish the book. This means that between the *want to read* list, and the *currently reading* list, there needs to be a button to *start reading*. Similarly, between *currently reading* and *have read*, there should be a button to *mark as read*.

Recommendations made to users may be irrelevant to the users. This means that there should be a way to remove recommendations. For this reason, there should also be a delete button on each recommendation entry. A delete button should also be included on each entry into reading lists, as the user may change their mind about each entry.

When first signing up, it is logical to ask for some information to improve the quality of the recommendations. The users may not want to do this initially, because it is time consuming, or they are not fully aware of what they are interested in yet. For this reason, it is important that there is a way to skip this stage and build up a picture of their interests over time, such as through reviews or through what they have added to their reading lists.

The responses to the reading diaries were focussed around having complete the book. However, most reading diaries are filled in after any length of book has been read. The fields that were given by the end users are still applicable, however. The dedicated page for the diary entries should then have an option to make an entry for when the user has read any of the book and record their views then. These entries should also include the number of pages that have been read, which can then be used to display the percentage that each book has been read in the reading lists. This data could also be used to calculate how long it takes the user to read the book and generate an estimated time to read any book. However, to keep the complexity down, this is for an extension to the project only. The diary page must also enable the users to view all the entries that they have made, as part of the reason for documenting the reading history is for future reference.

As well as having customised recommendations, it may be useful for users to have a selection of books that are trending, which are the books that have the highest number of people currently reading. This allows users to get an understanding of what is popular, and therefore are likely to be good books, which can then be used as ideas for what to read, outside of recommendations. This is a simple feature to implement, which can give a significant improvement to the user experience, which is applicable regardless of the user’s recommendations. For this reason, this should be included in the project.

## Project Brief

I am proposing to design and implement a system that provides personalised book recommendations for all the users. It also provides the users with three lists, which allow the user to record what they have read previously, what they are currently reading, and what they want to read in the future. Furthermore, the project should also provide the user with a reading diary, so that they are able to document their reading history. Each book within the website should have an about page, including all information relevant to the book, and a link to a website to buy the book, with the current price. These pages should also show all the reviews for that book and allow users to leave reviews about it. Finally, author should have public-facing profile pages that describe them and their reading history, with users being able to follow them.

## Project Requirements

Having contacted the end users of the project, I have gained a significantly improved understanding of the project requirements. Their responses, outlined above, have been used to significantly improve the project requirements, adding missing functionality, and removing anything that is unnecessary. These broad aims have then been decomposed into smaller subsections that are significantly more measurable, and achievable, giving a more effective way of assessing the system at the end of the project, as well as making the research and development of the project simpler. These update requirements are listed below.

* It must use a website.
  + The website should have a modern design.
  + The website must be easy and intuitive to navigate for third parties.
  + Each book stored in the database should have a dedicated *about* page, which displays all the available information within the system.
  + Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system.
  + Each author stored in the database should have a dedicated *about* page, which displays all the available information within the system.
  + The users’ reading lists must be available on a dedicated webpage on the website.
  + The users’ recommendations must be available on a dedicated webpage on the website.
  + The users’ diary entries must be available on a dedicated webpage on the website.
  + The website must use a hierarchical website structure, with all pages being able to be navigated to from the home page, or further pages in a logical format.
  + The website’s colour theme should be white, black, navy, grey and red colour scheme.
  + The website should be able to support at least five concurrent users.
  + Pages should be switched and navigated between without refreshing the page
* The website’s home page should present users with a list of frequently accessed information.
  + The home page should give users a summary of their recommended books.
  + The home page should give users a summary of what is in their *want to read* list.
  + The home page should give users a summary of what is in their c*urrently reading* list.
  + The home page should give users a list of trending books, which have the highest number of users currently reading them.
  + Book summaries should include the book’s cover image, author, and title.
  + Each book summary should link the specific books about the page.
  + The home page should have links to the other main sub-pages so that users can navigate easily around it.
* Users should have their own accounts on the website.
  + Users must be able to create their own accounts on the website.
  + Users must be able to log in to the website to access their personalised information.
  + Recommendations and reading list entries must be kept private.
  + Each user’s diary entries must be kept private
  + The website should be able to save a user’s login
* Personal information that is stored within the database should use appropriate data protection methods.
  + All passwords should be hashed before being stored in the database.
* Users should be able to create collections of books called reading lists
  + Users should be able to record the books that they have read in the past.
  + Users should be able to record the books that they are currently reading.
  + Users should be able to record the books that they would like to read in the future.
  + *Have Read*, *Currently Reading*, and *Want to Read* reading lists should be automatically created when a user signs up.
  + Users should be able to create their own custom reading lists
  + The user should not be able to delete the default *Have Read*, *Currently Reading*, and *Want to Read* reading lists
  + Users should be able to delete their own custom lists
  + Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list
  + Each entry into a reading list should allow the user to delete the entry.
  + Each entry into the *have read* reading list should also supply a link to all the user’s diary entries for that book.
  + Each entry into a reading list should supply a link to the books about page.
  + Reading lists, and the entries within them should be specific to the user.
  + Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list.
  + Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list.
* The system should generate recommendations for the users.
  + The system should generate recommendations that are unique to each individual user’s tastes.
  + When signing up for the first time, users should be prompted for their favourite authors and genres.
  + Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre.
  + Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation.
  + Recommendation entries should have a method to move it to the *want to read* reading list.
  + Recommendations should include whether the user follows an author
  + Recommendations should include whether the book is in a reading list
  + Recommendations should include the user’s diary entries
  + Recommendations should be regenerated daily
  + Recommendation entries should show the certainty of the recommendation
* Users should be able to have a diary to record what they have read.
  + Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating.
  + Users must specify the number of pages that they read.
  + When the user finishes the book, the user should be given the option to make the diary entry available as a review.
  + Each diary entry should supply a link to the corresponding book about page.
  + Users should be able to delete diary entries
* Each book should have an *about* page, which gives all the related information about the book.
  + The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book
  + A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website.
  + It should have a method to add the book to the *have read*, *currently reading*, and *want to read* lists.
  + There should be a list of books that are related to each other displayed.
  + Each related book should link to the corresponding about page.
* Each author should have a profile page.
  + It should display the author’s average book rating.
  + It should display all the genres that the author writes for.
  + It should show all the books that the author has written.
  + It should supply an *about the author* section.
  + It should show the author’s name (or the name that they write under if applicable).
  + Users should be able to follow each author.
  + The page should display how many followers each author has.
* Users should be able to leave reviews about books that they have read.
  + Users should be able to leave reviews about each book
  + Users should be able to delete any reviews that they have left
  + Reviews should be listed on the corresponding book *about* page.
  + Reviews must have an overall rating.
  + Reviews can have a character rating, plot rating, review summary, and review body.
  + Reviews on the book should show the user’s username.
  + Users should only be able to review a book if they have read it, marked by it being in the users’ *have read* list.
* Each genre should have an about page
  + It should display the genre’s name
  + It should show an about summary for the genre, if applicable
  + It should show the books that are part of the genre
* There should be a quick way for users to find books, authors and genres.
  + There should be a *browse* page on the website to display popular books, highest rated, and the newest additions the website
  + The *browse* page should display books related to the last one added to a reading list
  + The *browse* page should display books related to the last read book
  + The *browse* page should display books from the authors the user is following
  + There must be a search bar that searches for books, authors and genres within the website.
  + The search bar should be able to search by name or ISBN number
* There should be a simple way to configure the system
  + Parameters that can affect performance should be configurable
  + A default configuration should be used, which can be updated as needed based upon available hardware
  + Recommendations should be configurable based upon the books and data stored in the database
  + Issues with the configuration should be clear, easy to understand, and simple to debug
  + Configurable parameters should be modified through an external configuration file, which is in a clear and readable format

## Project Acceptable Limitations

There are many extensions to this project, with many ways to improve the functionality, performance, and accessibility. However, to ensure that it is achievable in available time, a set of limitations has been put in place. They help make it simpler to develop and reduce the complexity.

* It only needs to work correctly with Google Chrome. Support for legacy browsers does not need to be guaranteed.
* HTTPS is not required
* It does not need to have support for audiobooks, e-books, or films in any way.
* It does not need to support a high number of concurrent users.
* It does not need to have audit trails for system changes.
* It only needs to work with HTML5 (may not be able to support legacy browsers)
* It only needs to work for browsers that support scripting and have it enabled.
* It does not need to supply any form of API.
* There does not need to be a web-based interface for adding books or authors.
* It only needs to work on displays larger than 950px, so it does not need to be developed for mobile devices in mind.
* It can assume that all the requests made to the server are only made through the webpage
* The system does not need to supply two-factor authentication, or any level of security beyond a username and password.

## Background Research

To gain a suitable understanding of the project requirements, and how they will function, every aspect of the proposed system has been researched with sufficient detail to help ease the design and implementation. This was performed by breaking down the entire system into its components, as shown in the revised system requirements. For added help with the system, I also contacted several libraries from around the UK, with Natasha Kennedy from Bodleian Library and Timothy Collinson from Portsmouth University Library, to help with the criteria for parts of the platform. This research is then going to be used to design and implement this system.

### Website

As mentioned in the project specification, the user must interact with the system through a website. This approach negates some issues of developing for Android and IOS, and for desktop applications for the same degree of accessibility, but brings other challenges, including hosting the website, running web servers and a dependence on JavaScript, HTML and CSS, which may make it harder to develop. Furthermore, other considerations include the use of HTTPS, and the type of gateway interface to use for the server to communicate with the script that generates web pages and responds to any requests made, outside of static file requests.

#### Web Servers and Web Hosting

A web server is an application that runs on a server that makes use of a range of protocols, including HTTP and HTTPS to display website content by storing, processing, and delivering webpages to clients (Gillis, 2020). To be able to host a website, the computer that is being used must run any type of web server software, as without it, data would not be sent to the client, and hence, the webpages would not load. There are different web server applications that could be used, with varying advantages and disadvantages. These are outlined in the later sections.

Web servers work equivalently, regardless of which software is used. According to (Gillis, 2020), when a web browser requests a page from a web server, such as when visiting a new webpage, the specified URL is then used to find the IP address of the domain, where the web server is, by either using a Domain Name System (DNS) or searching in the browser’s cache. An HTTP or HTTPS request is then made, from the browser to the web server specified by the IP address given, which then sends the requested file back to the browser, for it to display. In case of an error occurring, such as a page not existing, or an error occurring, the server responds with an error message, and consequently, the page does not load.

Web servers can be broken down into two categories: dynamic and static. A static server is only able to serve static files, such as HTML, CSS, and JavaScript, and is unable to dynamically generate or change them before sending them to the client, so they are served as they are stored on the server. According to (Gillis, 2020), these servers only consist of the machine and the HTTP(S) software. Dynamic servers can change any files before sending them to the client or create entire web pages dynamically depending on the request. These consist of a web server, and associated software, such as an application server and a database, and as a result are more flexible, at the expense of simplicity (Gillis, 2020). For this deployment, a dynamic server is needed for multiple reasons. First, every book that is stored on the system requires its webpage, and hardcoding all of those is impractical, especially as the size of the system increases, and secondly, each user has their login, and therefore personalised information, including their recommendations and reading lists, which varies depending on the user. This means that it must be dynamically generated, forcing this system to use a dynamic web server.

Web servers can also be further broken down into synchronous and asynchronous web servers. A synchronous web server blocks the execution of any code on the client’s browser, while using asynchronous transmission does not do this, allowing the client’s browser to continue processing normally (Mozilla, n.d.). Asynchronous transmission is better as, (Mozilla, n.d.), outlines, synchronous requests cause the browser to freeze and appear unresponsive. However, by allowing processing, this does not happen, and the browser continues to function as normal until the response has been sent, giving a more seamless user experience. Consequently, an asynchronous web server is ideal for this project, because of the better user experience.

##### Apache

The Apache HTTP Server Project (Apache, 2015) is a “*collaborative software development effort aimed at creating a robust, commercial grade, featureful, and freely available source code implementation of an HTTP (Web) server*”. It is an extremely popular web server according to (Hernandez, 2019), which, because of being open source, has many added modules to give more functionality and improve its features, such as modifications to include FastCGI. This web server can serve many concurrent users with few alterations in the configuration files and can scale (Hernandez, 2019). Furthermore, the modules that are used within Apache allow for lots of configuration, including adding more to improve functionality, and removing unused or unwanted modules to help improve the efficiency and size of the server (Hernandez, 2019).

For this project, Apache will not be used. This is because there are more lightweight alternatives, which is important if it is to be run on a Raspberry Pi. Furthermore, the level of customisation adds an unnecessary degree of complexity, which, while beneficial for larger sites, adds complexity to this project, which is only a small-scale website. Finally, the speed that it supplies is not necessary for this project, as it is very unlikely to experience high numbers of concurrent users.

##### NGINX

According to (Nginx, n.d.), NGINX is an “*open-source software for web serving, reverse proxying, caching, load balancing, media streaming, and more”.* It is a popular web server, initially developed, that often outperforms other web servers in benchmarking tasks, especially in applications which have static content or high concurrent requests, as well as being designed to offer low memory usage (Kinsta, 2022). NGINX uses a single master process, which is used to control worker processes, which are then used to perform the web server’s processing asynchronously (Kinsta, 2022). Asynchronous processing allows for processes to be executed, whilst not blocking any other requests, as previously explained.

For the final deployment of this project, NGINX will not be used. This is because, while it provides a high level of customisation and performance, the ability to scale to a high user base is not necessary, and the increased level of configuration is not necessary for this project and makes it more complicated to set up and run correctly, especially when compared to Lighttpd, which provides many of the same functionality. If this project gained a large user base in the long-term future, NGINX could be considered, as it does offer high performance and stability, but as this is not the case, it will not be used.

##### Lighttpd

According to (Lighttpd, 2023), Lighttpd is a “*secure, fast, compliant, and very flexible web server that has been optimized for high-performance environments. Lighttpd uses memory and CPU efficiently and has lower resource use than other popular web servers”.* It is an open-source web server software, and due to its low hardware usage, it is ideal for environments with limited processing and memory resources (Morris, 2022). Lighttpd was the first web server that was able to perform the C10K problem, so it can sustain 10,000 simultaneous connections (Morris, 2022). Furthermore, (Morris, 2022) states that it manages the requests asynchronously. This means, because of the previously mentioned reasons, it supplies a more seamless user experience, which is preferable.

(Morris, 2022) also outlines several advantages and disadvantages of using Lighttpd. One of the biggest advantages of using Lighttpd is the low computational cost. If this is deployed using a raspberry pi, the available memory and processing power are limited, and therefore this is a major advantage. Furthermore, it supports all programming languages and therefore will work with Python, which is what this will be programmed in. However, it is much less popular than other alternatives, such as Apache, and therefore has a much smaller user base. Consequently, there is less community support and documentation. In addition, it has fewer features and customisations than other alternatives have, and finally, it has limited performance advantages for larger websites, but given that, at least initially, this project will not be for a large website, these are not problems.

When deploying to the Raspberry Pi, this will be the preferred web server. This is because of the low hardware usage, and therefore superior performance on low-performance hardware, as well as responding to requests asynchronously, and being able to supply dynamic web pages, as is required by this deployment.

##### PythonAnywhere.com

(PythonAnywhere, n.d.) is a web-based Python editor, which can host websites. It is important to note that PythonAnywhere is not a web server, but is instead a web hosting provider, which then runs and supports its own web server, and consequently I do not have any knowledge of, or ability to change the web server they use. It can host websites using custom domain names and remove the need for manual configuration of the configuration files for the web server. This makes it easier to develop a website and could be used for a full deployment of the system. Furthermore, valid SSL certificates enable the website to use HTTPS without any added cost and therefore have access to the improved security that it provides. Furthermore, with high performance, it is easily scalable. Finally, the inbuilt Postgres and MySQL integration also makes it easy to run the database on the cloud, and access it through Python.

This will be used over Lighttpd running on Raspberry Pi because of the higher performance that it offers, and the ability to use HTTPS finally it is hosted, so could go from a prototype to a functioning application very easily. However, using Lighttpd on the Raspberry Pi does allow for a higher level of customisation, which may then enable the webpage to perform better, but does mean that an SSL certificate needs to be bought as well.

#### Dynamic Content Generation

Dynamic content, or adaptive content, are web pages whose content changes based upon the user, such as their behaviour, preferences, and interests (Omniconvert, 2023). The content of the webpages is then personalised to the user, and are therefore more relevant to them, creating a user-focused website. Dynamic content generation can also be used to generate web pages to display information, where the number of required pages would mean creating the relevant number of static web files would be unmaintainable. For example, according to (StackPath, 2023), if Wikipedia stored all the individual pages as static files, it would be storing around forty million static files, which would be unmaintainable. Furthermore, server-side and in-browser scripts can then be used together to load dynamic content over already loaded pages (StackPath, 2023). This could be done using the Fetch API, AJAX, or any other method within the browser script, which is usually written in JavaScript.

For this project, dynamic content generation should be used instead of static files. It enables the website to be user-driven, and therefore the content displayed is relevant to them only. This is done through the user signing into the webpage, and then being shown their own specific book recommendations, reading lists and diary entries. While this project is on a small scale, the ability to generate the *about* pages for authors and books when needed helps make it much more maintainable and faster to add records onto the system. This functionality will also make it more scalable, although may increase the times to access the web pages. Being able to communicate back to the server, also allows for any changes made to the user’s information, including recommendations, and reading lists, to be sent back to the server, and update the database to reflect the changes.

Using dynamic content generation has many advantages over using static content. According to (StackPath, 2023), it can be personalised to the user, so responses can be based upon the user’s request or earlier activity on the website. (StackPath, 2023) also explain that it is extensible, so the scripts that generate the content can use external resources, such as local files or databases, and dynamic content is also interactive, so the users can send information back to the server. Furthermore, according to (Omniconvert, 2023), it allows for a more user-friendly experience, which should be prioritised, as it could improve engagement with the project. As previously mentioned, it also helps keep the project easy to support, with *about* pages being generated from the information stored in the database, instead of requiring a static file for each of the items stored in it. This also means that once the website is set up, it no longer needs to be changed and altered, regardless of whether new books or authors are added to the system, which improves the maintainability of the system.

There are a few disadvantages of this approach over using static files only. It is significantly more complicated, and therefore will increase the complexity of the project. Furthermore, the script that generates the files must run, before it is generated, and can be sent to the user. This can increase the page load times. However, the ability for the content to personalised to the user, which is essential for this project, and how easy it is to add new pages, means that this is the approach that should be taken.

Dynamic content generation is performed by a server-side script, which runs on the server hosting the website, which then works with the server to parse the request and generate content based upon it (StackPath, 2023). The server uses the URL to decide how the request should be processed, for example, a question mark (?) often precedes query information. After the request has been received by the web server, it is passed to a script running on the server, which then processes the request and generates a response, often in HTML, which is then passed back to the client (StackPath, 2023). The script can be coded in most languages, such as ASP.NET, Perl or PHP, although in this case Python will be used.

#### Common Gateway Interface

Common Gateway Interface (CGI) is a web protocol that defines a way for a web server to interact with external applications (superhosting.bg, 2016). For this project, these external applications will be coded in Python. CGI allows for dynamic content generation and processing (superhosting.bg, 2016), and without dynamic content generation, all web pages are static, and therefore any alterations need to be made to the corresponding HTML file directly.

This ability to dynamically generate content is especially important when there are many different pages needed, following the same template, but with different information displayed. In this project, the information about each book would follow the same template, and have different content included. For each of these items, without using any form of content generation, there would have to be the same number of HTML files created as the number of books stored on the system, the same also applies to the authors. This approach would be time-consuming, impractical, and most significantly, making it difficult to add new books and authors onto the system. Generating these pages when needed would save development time, and make adding new books faster and simpler, as only the database would need to be updated. Furthermore, each of these items would follow the same template, so that the website is coherent, and so, the relevant data can just be inserted into the page when it is needed. In addition to this, this would also be applied to any user-specific pages, such as reading lists and recommendations, where the content changes depending on the user that is logged into the page.

Furthermore, being able to run code that varies due to the user, webpage, and any data inputted through forms, also allows for increased functionality. For example, when using the search tool on the page, a script must be run to generate the top results, or when adding a book to a reading list, the database must be updated to add the item to the list. Most of the functionality of the website needs the ability to respond to data inputted into the website or send more information to the client, by responding to post or get requests made from the client. Other frequent uses of CGI systems, some of which are used in this project, include shopping baskets, comments, forms, website statistics and server-side includes (IONOS, 2020).

There are many different web servers available, as previously mentioned, some of which use different implementations of CGI, FastCGI or WSGI. For example, Apache uses *mod\_fcgid*, *mod\_cgi*, or *mod\_cgid* for CGI and FastCGI (Apache, 2013). However, these different implementations work in similar ways.

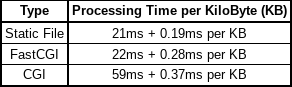
##### CGI

Common Gateway Interface (CGI) is the simplest method that is outlined in this document. The server recognises a URL which has a request for a CGI script, and then the server calls a program for the client (IBM, 2022). The CGI script, which can be written in most languages, in this case, Python, is then called and then generates the response, such as generating a web page. Using CGI has its advantages, for example, easy integration into existing infrastructure and being open source (IONOS, 2020), however, it does have its drawbacks. For each CGI request, a new instance of the script must be created each time, making the time to reply time lengthy (IONOS, 2020). This restarting of the script comes with an overhead, which reduces the performance. Furthermore, on high-traffic websites, new queries are added to a queue or are rejected, and on many servers, only a few of these CGI queries can be managed simultaneously (IONOS, 2020), which further compounds the long reply time issues. Finally, CGI can pose a security risk, as, in theory, external programs can gain unauthorised access to data stored on the web server through CGI, through query manipulation, so there must be restrictions in place to prevent CGI scripts from causing any damage to the server (IONOS, 2020).

In many respects, CGI would be applicable for this deployment, as it is unlikely to experience high traffic. However, for some of the scripts, some pre-processing will be required for them to function properly, without any reasonable way to negate this issue. For example, if the *symspellpy* module was used, the *load\_dictionary* method would need to be called before the spell check would work. On the Raspberry Pi 3b that will be used for testing, over 2,400 tests, it took an average of approximately 18.5526 seconds to load into memory. When using CGI, this must be performed every time a search is made, rendering the search feature useless, as, according to (Anderson, 2018), “*53% of mobile site visits are abandoned if pages take longer than 3 seconds to load*”. For this reason, this deployment is pushed towards alternatives, which increases the complexity.

##### FastCGI

FastCGI is an alternative solution to GGI, which solves the fundamental issues that come with CGI. With CGI, for every request made, the web server spawns a new process (Open Market, n.d.), which comes with a processing overhead, and consequently can be slower. In comparison, the web server creates FastCGI processes to manage the request, either when in demand or at start-up (Open Market, n.d.). However, these processes are persistent and are then used for later requests (Open Market, n.d.), instead of ending and reinitialising as with CGI, consequently negating the performance issues associated with CGI. Table 1 outlines the performance gains between CGI and FastCGI.

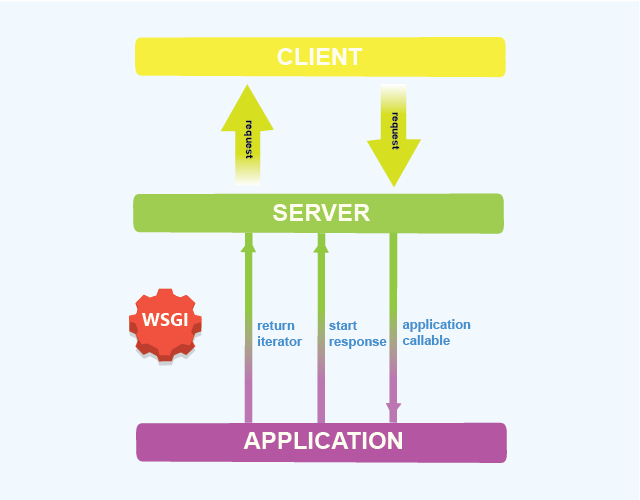


*Table 1 - Processing times for CGI methods, with static file loading for comparison (Open Market, n.d.)*

Outside of performance benefits, FastCGI also provides several advantages, outlined by (Open Market, n.d.), including improved performance because of persistent processes, the ability to be written in any language, FastCGI processes being unable to corrupt or crash the core server or other applications, malicious FastCGI applications are unable to obtain any secrets from the Web Server, being independent of any specific server architecture and finally being able to run applications remotely, and an consequently help with load distribution.

##### WSGI

The Web Service Gateway Interface (WSGI) is an API that is defined in PEP-333, which allows communication between web servers and Python web applications (Rees, 2016). However, a revised version of the PEP-333 specification is PEP-3333, written by (Eby, 2010), which is updated for use with Python 3, although code that conforms to PEP-333 will also conform to PEP-3333. According to (Prasad, 2018), WSGI uses a standard interface for routing web apps and frameworks to servers, based upon CGI. Whenever it receives a HTTP request from a client, a callable is invoked by the server, which then produces a response. The web server triggers the web application, and sends the related data, with a call-back function to the app, and the processing of the request takes place on the application side, and the web server receives the response using the call-back function, see Figure 16 (Prasad, 2018). An application, according to (Eby, 2010), is a callable which takes two parameters, environ, a dictionary of variables which describe the HTTP request, and start\_response, which is a function that starts the generation of the HTTP response.



*Figure 16 - WSGI diagram (Prasad, 2018)*

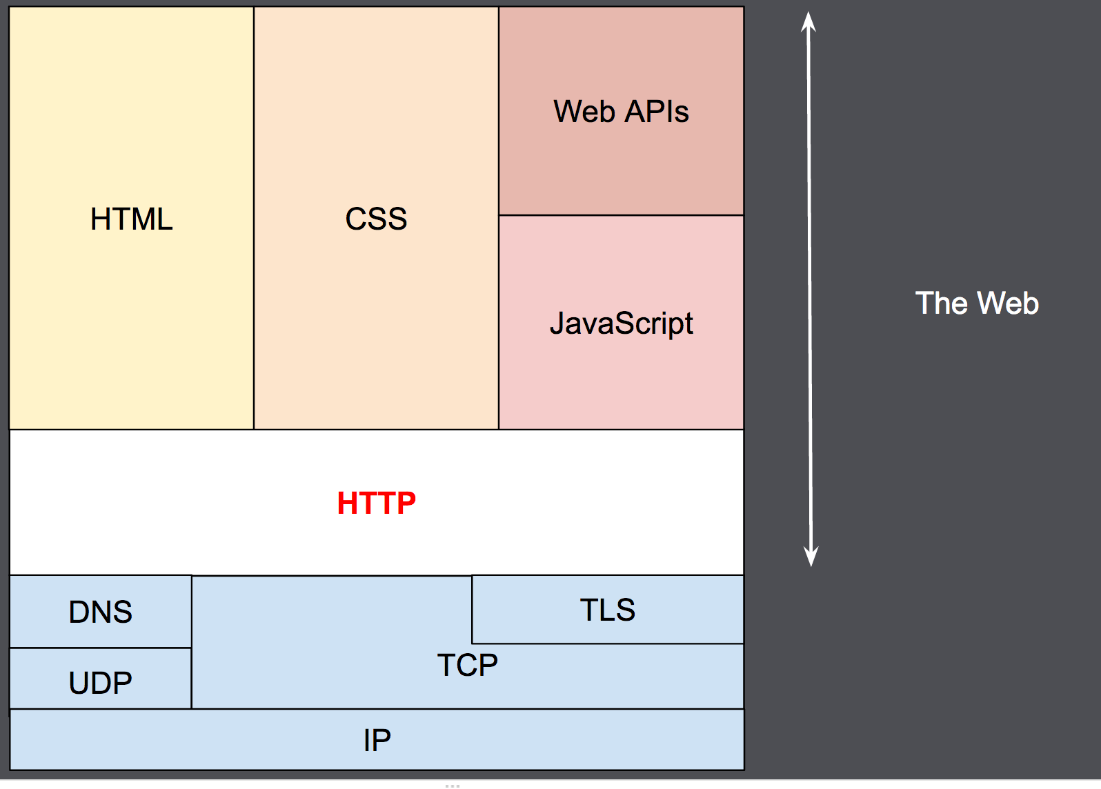
WSGI has multiple advantages over CGI or FastCGI, explained by (Prasad, 2018). It is very flexible, as the web stack components can be changed without alterations to the code, or the application that is running the WSGI server. WSGI servers can manage thousands of concurrent requests, and route them from the web server in the most effective way possible, so it is scalable (Prasad, 2018). Furthermore, the middleware can be reused, which saves time (Prasad, 2018). However, the biggest advantage for this project is how simple it is to learn, develop web applications with, and use with Python. For this reason, this is the method that will be used for this project.

###### Middleware

A WSGI middleware according to (Eby, 2010), is a single object that “*may play the role of a server with respect to some application(s), while also acting as an application with respect to some server(s)*”. These are then able to perform many distinct functions, such as, according to (Eby, 2010), rewriting the environ dictionary according to the target URL to reroute requests to different objects, allowing multiple applications or frameworks to run simultaneously within the same process, perform load balancing and remote processing by forwarding requests and responses, and post process content. Whether there is a middleware is normally transparent to both the server and the application and should follow the restrictions given by the web server and application sides of the interface (Eby, 2010).

#### Protocols

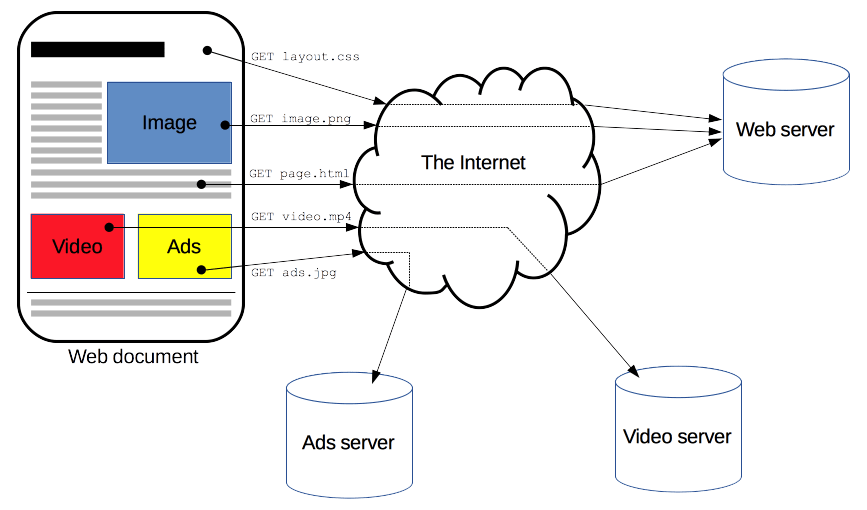
When a request is made from a browser, a series of protocols are used to supply the contents to the user. This is achieved by using a series of protocols, defining how the communication should occur between the client and the web server. First, according to (Khan Academy, n.d.), the Domain Name System (DNS) protocol is used. The DNS protocol is a protocol that is used to take the URL entered into the browser and provide the client with the server’s IP address. This then allows the HTTPS(S) protocols to be used to retrieve the website data from the server (Cloudflare, 2022). Following this, the HTTP or HTTPS protocol is used to retrieve the data from the web server (Khan Academy, n.d.). The protocol that is used is specified within the URL, http://domain.namefor HTTP, and https://domain.name for HTTPS. An overview of this system can be found in Figure 17 (Mozilla, 2019). The main difference between HTTP and HTTPS is that HTTPS includes encryption for data that is sent over it. This brings multiple advantages, outlined by (Cloudflare, n.d.). Firstly, websites using HTTPS are more trustworthy for the users, with many browsers marking them as *“not secure”.* Furthermore, HTTPS is more secure for the users and website owners, and finally, authenticates websites, so users are aware if the website is what it claims to be.



*Figure 17 - Interaction of protocols on the Web (Mozilla, 2019)*

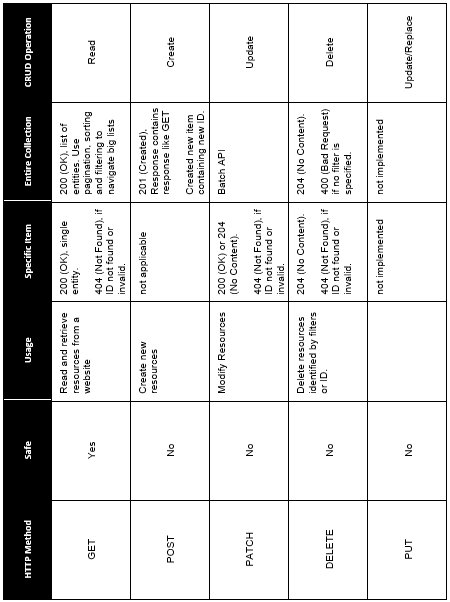
##### HTTP

Hypertext Transfer Protocol (HTTP) is a web-based protocol for fetching resources from a web server (Mozilla, 2019). These requests are directed to the relevant server, which sends the data back to the single client, shown in Figure 18. These resources could be of any form, including XML, PNGs, JPEGs, MP4s and HTML files. When the server receives the HTTP request, it produces a response, holding any error codes, and the requested file.



*Figure 18 - Diagram showing how HTTP requests are routed (Mozilla, 2019)*

There are different forms of HTTP requests, which each work in a separate way, to perform a different function. These are outlined in Table 2 (ORO, n.d.). Each different request can be managed by the web server in separate ways and therefore can be used to perform different operations. For example, a post request sends data to the web server, such as for forms. GET requests, however, request specific data to be sent to the browser, such as HTML, CSS, and image files.



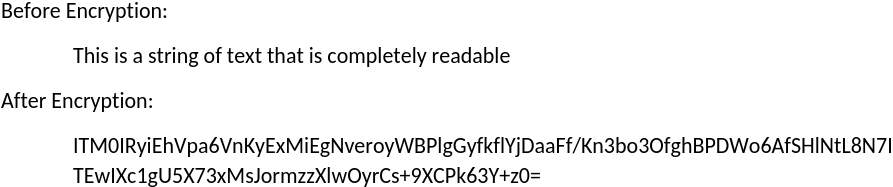
*Table 1 - Types of HTTP Request (ORO, n.d.)*

##### HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is HTTP, with encryption added at each end, preventing a man-in-the-middle attack. It encrypts the data to increase security, which is vital when transmitting sensitive data, such as logging into accounts, such as emails (Cloudflare, 2023). HTTPS is not a separate protocol from HTTP, which uses TLS and SSL encryption protocols on top of the HTTP protocol to verify the authenticity of the website (Cloudflare, 2023).

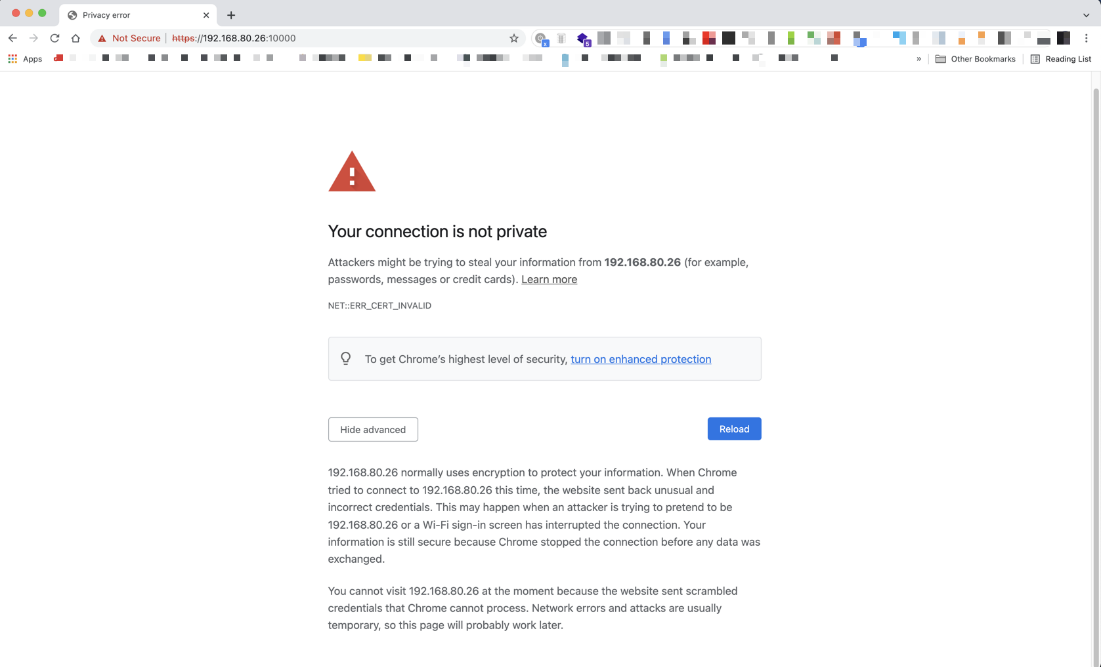
When the user connects to a webpage using an HTTPS connection, it sends the client its SSL certificate, which contains the public key required to start the secure session (Cloudflare, 2023). The client and the server then perform an SSL/TLS handshake, consisting of a series of back-and-forth communications to establish a secure connection (Cloudflare, 2023).

By encrypting the communication between the client and server, it prevents the information from being intercepted by unintended third parties. With HTTP, the data that is sent between the server and the client is plain text, so can be intercepted and read, whereas HTTPS encrypts the data, meaning that it cannot be read if it is intercepted, see Figure 19, (Cloudflare, 2023). This is particularly important for sending login credentials, as they could be intercepted and used by unauthorised users. Even if this system does not store any sensitive data, this should still be avoided. For this reason, this deployment should use HTTPS if possible. Furthermore, with HTTP websites, intermediaries or internet service providers can inject content, usually advertisements, into the web pages, without the permission of the website owner, and consequently, any profits that are made do not go to the website owner (Cloudflare, 2023). This should also be avoided as it negatively impacts on the user’s experience on the website and could lead to lower uptake and usage of the website. Finally, as previously mentioned, HTTP is not secure, and many browsers, including Google Chrome, mark HTTP websites as “*not secure*.” This could dissuade users from visiting the website and should then be avoided. For these reasons, this system should use HTTPS where possible.



*Figure 19 - Example encryption (Cloudflare, 2023)*

HTTPS, as previously explained, needs an SSL certificate. These can be bought from companies, such as Cloudflare, or can be created, known as self-signed certificates. Self-signed certificates are certificates that are not signed by a trusted certificate authority but are signed by the developer of the website (Entrust, n.d.). Without any external verification to guarantee the validity of the information contained in a certificate, they are usually considered unsafe for public applications and websites (Entrust, n.d.). These provide warnings in most browsers that the website is insecure, such as Google Chrome, see Figure 20, and as stated by (Entrust, n.d.), this could be the result of malicious third parties attempting to gain access to the network and launch malware attacks, including man-in-the-middle, phishing, and botnets. With some browsers, it can prevent the page from loading, and this should be avoided. For this reason, a certificate signed by a trusted certificate authority should be used. It is important to note, however, that if this is run using PythonAnywhere, HTTPS can be used, which has a valid certificate, and should not cause errors.



*Figure 20 - Google Chrome error due to a Self-Signed Certificate (EnoVision, 2021)*

##### HTTP Requests

As previously mentioned, there are many different HTTP requests that could be used, including GET, POST, PATCH, DELETE and PUT (ORO, n.d.). Most of these are not relevant for this project, and the ones that will be used are POST and GET. GET requests cover a substantial portion of the project and are used whenever data from the server is requested by the client, through the webpage, and POST to send data to the server from the client, to be acted upon.

GET requests, according to (Mozilla, 2023), the GET method is a HTTP method that “*requests a representation of the specified resource*”. It is the primary method of information retrieval (Fielding et al., 2022). These requests can send a payload within the request, however, may cause existing implementations to reject it, this should be avoided, even though the specification does not prevent this, the semantics are not defined (Mozilla, 2023). This rejection of requests can occur because it could be a request smuggling attack (Fielding et al., 2022). These GET requests can be cached, which may then be used for further GET or HEAD requests, unless the cache-control header field states otherwise (Fielding et al., 2022), and caching can increase the speed of page load times. A GET request is made up of the target URI, and any payload, although this is advisable, for the aforementioned reasons, see Figure 21. A URI (universal resource identifier) identifies a resource and differentiates it by using a name, or location, while a URL (uniform resource locator) has the same data as a URI as well as a domain name and port (Sopha, 2022).



*Figure 21 - Example GET request for an image.*

POST is a similar type of HTTP, which “*requests that the target resource process the representation enclosed in the request according to the resource’s own specific semantics''*, which could be used for providing data blocks, such as the fields entered into an HTML form, to a data-handling process, posting a message, creating a new resource that has yet to be identified by the origin server, or appending data to a resource’s existing representations, amongst other use cases (Fielding et al., 2022). It is used to send data to a server, with the data included in the payload, rather than the URI, so should be used where the data needs to be kept private. These requests are idempotent and are only cacheable if it includes explicit freshness information, and a Content-Location header field with the same value as the POST’s target URI (Fielding et al., 2022). According to (Mozilla, 2023b), POST requests are normally sent through an HTML form, where the payload can take a fixed form, although any other method can take any type (Mozilla, 2023b), if the semantics of the payload agree with the server’s. When using a form, the content type is specified by the enctype attribute of the <form> element of the formenctype attribute of the <button> or <input> elements. An example POST request is shown in Figure 22 (Saha et al., 2020).



*Figure 22 - Example POST request within a web-server's access log (Saha et al., 2020)*

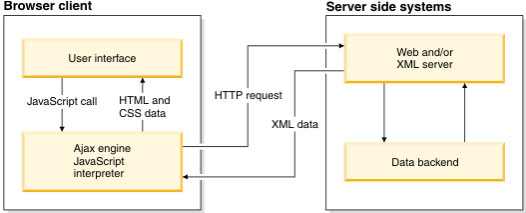
GET requests are able send a payload to the server, however, this is not advised, so are only intended to retrieve data from a server, and not modify its state, whereas POST requests are the used to send data back to the server, to process, or modify it state (baeldung, 2023). Furthermore, the GET request parameters are contained within the URI, and therefore are visible, however, when using POST, the data is contained within the HTTP request body so can only be viewed by the recipient. Consequently, POST is more secure, as well as the fact that the requests cannot be stored in the server logs, browser history, and cannot be bookmarked. For this reason, GET requests for any data requests from the server, including recommendations, reading list data, and so on. POST requests will be used for form entries, such as logging in, or the search bar.

#### JavaScript

As previously mentioned, JavaScript is a programming language that can be used to add functionality and interactivity to a website, which can be included into the HTML code for a website using the <script>tag. This link to the JavaScript file should be at the end of the HTML, as, according to (Frisbie & Zach Tratar, 2020), it makes the page appear to load faster, as it does not display a white screen, until the file is obtained. This occurs because the page loads sequentially, so the page waits for the contents of the file to be retrieved from the web server, before progressing onto the next section.

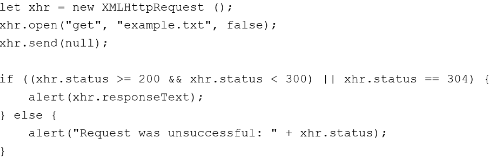
Through the JavaScript, POST requests, and GET requests can be issued, without any direct input from the user. This can then be used for many features on the website, such as submitting reviews, searching for books and authors, deleting recommendations, and items from reading lists, and the moving books to reading lists, amongst other functionalities. It allows users to view, change and update their data, and information on the system through the web page. In contrast, GET requests could be used to make the website more fluid, and smoother to use. For example, they can be used to switch between web pages within the website, by requesting new page content, without refreshing the page. This makes navigating between pages feel faster, and more fluid. These requests need to be through JavaScript, using a <button> element. This will increase the complexity of the website, which does increase the complexity, so for this project, it will use static files instead.

This can be done using Asynchronous JavaScript and XML (AJAX). It refers to “*group of technologies that are used to develop web applications*”, which can be used to make web pages seem more responsive (IBM, 2017). This works by sending data packets between the web server and client, without reloading the web page, which uses HTML or HTML and CSS for presenting data, the Document Object Model (DOM) to dynamically interact with and display information, the XMLHttpRequest object to transmit data between the client and server, XML, HTML and XSLT for data interchange and manipulation, and JavaScript for binding data requests and information display (IBM, 2017). The data transmitted is in XML, however, can be of any form, such as JSON, which is given in response to a JavaScript call, triggered by a user input, then the JavaScript/AJAX makes a HTTP request to the web-server, which sends back XML data, which is received, processed, and displayed using the JavaScript running in the client’s browser, see Figure 23 (IBM, 2017).



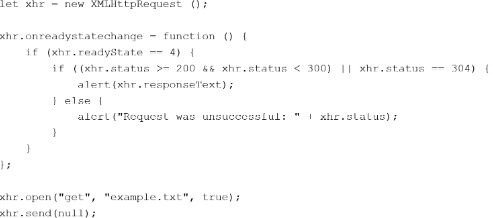
*Figure 23 - AJAX request diagram (IBM, 2017)*

As mentioned previously, this can be done with the XMLHttpRequest object in JavaScript. The process outlined by (Frisbie & Zach Tratar, 2020), is to first instantiate a new instance of the object XMLHttpRequest, then the open method needs to be called, with the parameters of the request type, URL, and a Boolean value as to whether it should be sent asynchronously. This prepares the request to be sent, to send the request, the send method needs to be called, which should have a single argument as some browsers require some data to be sent. After this is called, the data is sent to the server. The XHR response that is returned contains responseText which contains the data that was sent as the response body, responseXML which contains an XML DOM document with the response data if the response is either a text/xml or application/xml type, status, which is the HTTP response status, and the statusText which is the description of the HTTP status. The HTTP status codes contained in the status property of the object should be checked to ensure that the response was returned correctly. Generally, the 2xx codes are successful and some content will be available in responseText and responseXML, if the content type is correct, and the 304 code shows that a resource has not been modified and is being from the browser’s cache (Frisbie & Zach Tratar, 2020).



*Algorithm 1 - Example synchronous GET request (Frisbie & Zach Tratar, 2020)*

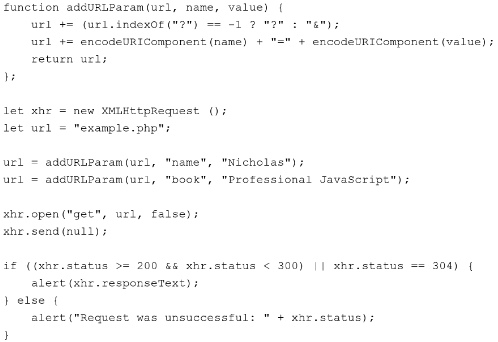
Asynchronous transmission is normally preferred to using synchronous, so that the execution of the code does not need to wait for the server’s response. The XMLHttpRequest object has a readyState property that indicates what phase of the request/response process, which has five different possible values, uninitialized (0) so the open method has not been called, open (1) so the open method has been called but the send method has not, sent (2) where the send method has been called but no response has been received, receiving (3) so some response data has been received, and complete (4), where all the response data has been received (Frisbie & Zach Tratar, 2020). Whenever these states change, the readystatechange event handler fires, so the onreadystatechange event handler should be assigned before calling the open method for browser compatibility, which can have a function assigned to it, that runs whenever it occurs, to check whether the readyState property has reached 4 yet, as well as the checks used for synchronous transmission (Frisbie & Zach Tratar, 2020).



*Algorithm 2 - Example asynchronous GET request (Frisbie & Zach Tratar, 2020)*

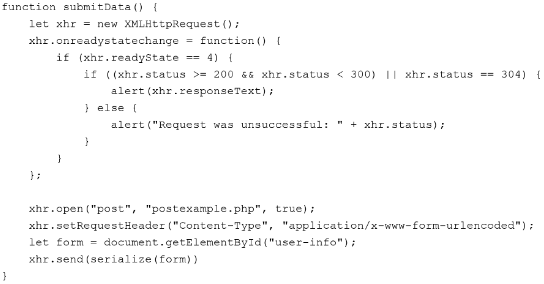
Furthermore, when an HTTP request is sent, or responded to, it is also sent with headers, which by default include Accept, Accept-Charset, Accept-Encoding, Accept-Language, Connection, Cookie, Host, Referer, and User-Agent (Frisbie & Zach Tratar, 2020). These can be added to, using the setRequestHeader method, which takes the header name and value, which the server can use to determine the correct response (Frisbie & Zach Tratar, 2020). It is also advised to use customer header names, as default headers could impact on the server’s response. (Frisbie & Zach Tratar, 2020). To access headers in the JavaScript, the getAllResponseHeaders can be used to get all headers in the request, and getResponseHeader to get a specific header, specified in the passed parameter (Frisbie & Zach Tratar, 2020).

Creating GET requests with parameters is required for dynamic websites, so parameters can be passed to the web server to specify what page should be created. Any parameters are in the form name=value, and any multiple parameters need to be separated with ampersands (Frisbie & Zach Tratar, 2020). Each parameter also needs to be encoded using the encodeURIComponent method (Frisbie & Zach Tratar, 2020). Algorithm 3 is an example GET request, with a function to add parameters to the URI (Frisbie & Zach Tratar, 2020).



*Algorithm 3 - Example parametised GET request (Frisbie & Zach Tratar, 2020)*

POST requests are expected to have data submitted in the request body, which can be a large amount of data in any format (Frisbie & Zach Tratar, 2020). The process to make a POST request is explained by (Frisbie & Zach Tratar, 2020), first the open method needs to be called on the XHR object, and the request parameter specified as “post”. The Content-Type header then needs to be modified to application/x-www-from-urlencoded, which is the format used for any form submission, as the data does not appear to the web server in the same way that a form submission does. Finally, the send method then needs to be used to send the data, which could be an XML DOM element, and then be serialised, or any string. The serialize function can be used to serialise any XML DOM elements (Frisbie & Zach Tratar, 2020).



*Algorithm 4 - Example POST request with an XML DOM element*

#### File structure

Many different organisation techniques are recommended for storing files on a website and can come down to personal preference. Below are a few different techniques that could be used, with a final decision made at the end. The approach that is used to storing files comes down to personal preference, as stated by (Learn the Web, n.d.), and therefore the one that will be used in this project is the one which is simplest to use.

(Stein, 2019) explains a simple approach to managing files, by sorting the files according to type into folders, see Figure 24. Each folder is named after the file type that is stored within it, for example, the images folder stores any image file, the CSS folder stores any CSS and the JavaScript folder stores any JavaScript files. Any HTML files are stored in the top level of the directory. To make this work with this project, the wsgi.py file would need to be included within the folder, so that it can be used and executed as part of the website.

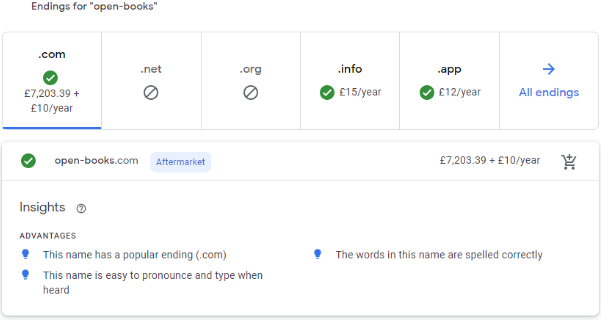


*Figure 24 - Simple website file structure*

Other approaches are more focused to larger scale projects, for example another approach is to have a www folder for files which holds the files on the website, which follows the structure given by (Learn the Web, n.d.). It uses an assets folder to store any files that are used, but not created specifically for the website, and a production folder contains source files used for the website, with compressed images in the www folder. As this is small scale, this is not required, and for this reason the first approach will be used. However, it is important to note that this can be changed, for example, the index.html file does not normally require an extension, but then to navigate to page1.html, the URI would need to be /page1.html, instead of /page1. A way to avoid this is to create a folder called page1, and have a HTML file in it, called index.html.

#### Website and Domain Name

The name of the website needs to be descriptive of its purpose. For this project, Open Book seems a suitable name, with, at the time of writing, there is an available domain name that can go with it. The domain is “open-book.uk”, which at the time of writing, can be purchased from Google for £10 each year. However, .uk is not particularly descriptive, so a .com ending would be better. “open-books.com” is available, at least on June 4th, 2023, from Google for £10 a year and an initial cost of £7203.39, see Figure 25. As this is not being used as a complete website, and as PythonAnywhere supplies a usable domain, this will not be purchased, but it could be done as a significant project extension if it were ever widely used. Open Book refers to the books stored on the website, but also how it opens the user to a new range of books and ideas that they would have otherwise not found, and therefore not engaged with.



*Figure 23 - Domain name pricing from Google*

As mentioned at the start of this research, I reached out to third parties to gain some insight into the creation of this project. In this, I also clarified whether Open Book was an appropriate name for the system, and this was agreed upon by the third parties, see Figure 26.

Chart, pie chart

Description automatically generated

*Figure 26 - Is the name Open Book suitable?*

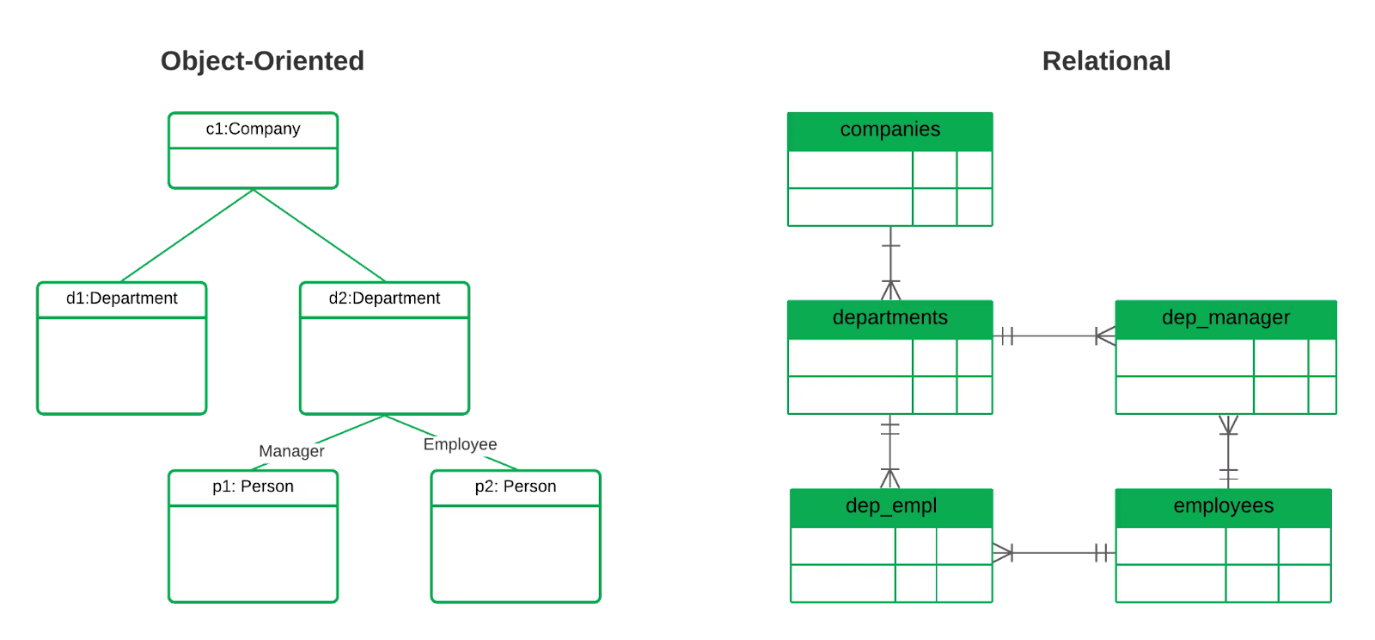
### Database

This entire project requires a method of storing authors, users, book entries and reviews. As a result, a database is required, which can be done in a variety of different ways, namely relational and non-relational databases. According to (Amazon Web Services, 2019), a relational database is one which stores a collection of data that are linked by pre-defined relationships. A non-relational database, however, according to (EdPrice-MSFT, n.d.) is a database where data is stored in a format that is optimised for the application, such as key/value pairs, JSON files, or graphs. For the type of data that will need to be stored in the database for this system, a relational database, like SQL, is more appropriate. For example, each author may have written multiple books, and a user may have reviewed separate books, the data for which needs to be accessed by the book, and user. While this may be possible using a non-relational database, it is likely to be much simpler when using a relational one.

There are many different relational databases available, however, with many being subscription or pay-to-use, the choice is limited. For example, (Microsoft, n.d.)’s Azure SQL platform has associated monetary costs, which makes it inappropriate for this application. However, MySQL is free, as well as a custom object-oriented database.

#### Object-Based Database Management System

According to (MongoDB, n.d.), an object-oriented database (OOD) is “*a database system that can work with complex data objects*”. Each individual object, like in any object-oriented program, encapsulates its own data within itself. With an object-oriented database, a hierarchy, as shown in Figure 27 is created, with each object referring to a single item in the database. The class hierarchy between different classes allows for the inheritance of different attributes and methods. For example, in this deployment, there would be a genre class, aggregated with author objects and book objects. This would then relate these together, so the program could get all authors that write that genre, and all books in that genre easily. However, the difference between a standard OOP-based program and an OOD, is data persistence, as with OOP-based programs, when the execution of the program ends, the objects are lost, whereas in an OOD, they remain stored (MongoDB, n.d.).



*Figure 27 - Comparison between object-oriented and relational databases (MongoDB, n.d.)*

This approach has its advantages. Using a OOD instead of a relational database (RDBMS) allows for fast queries, as the links between tables in the RDBMS do not need to be transferred, only object attributes accessed (MongoDB, n.d.). The issue with this approach is slow performance of simple operations, and the syntax is dependent on the language it is programmed in, instead of a fixed, highly documented syntax, like found in SQL (MongoDB, n.d.). Furthermore, for this deployment, this would need to be designed and coded from scratch, which increases the complexity of this system, whereas, by using a relational database, such as MySQL, this has already been done, so only efficient queries need to be created, instead of an entire database.

#### Relational Database Management System

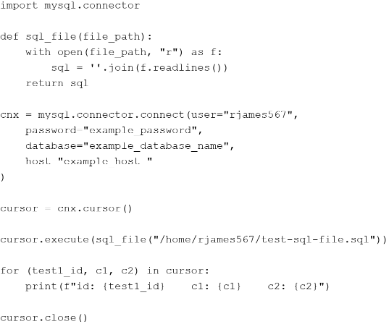
In comparison to an OOD, as explained above, MySQL is a relational database management system (RDBMS). According to (Oracle, 2019), the owners of MySQL, a relational database stores the data within separate, distinct tables within the database, which are then structured for query speed. The tables are the linked together with IDs, or other common characteristics, with one table linking to one table in a one to one relationship, or one table linking to many in a one to many relationship. One of the advantages of using MySQL for this project is that it already includes all the necessary features for this deployment, including being able to query data, as well as including rules, such as order, by id, or genre, removing the need to create this functionality in python. Queries to the database are made using the SQL syntax, which makes querying the database much simpler to do efficiently. On top of that, multiple users can access the data within the database simultaneously, with anyone with access to the database being able to navigate, query and modify the stored data (EASA, 2022). Furthermore, it also negates the issues with data persistence that is found using an OOD approach.

##### MySQL Connector

With this project, most of the requirements require access to the database and querying or storing data in it, including the searching algorithm, book/author pages, reviews and reading lists. For these to work effectively, the database must be queried depending on the type of request that is being performed. For this reason, the database must be queried through python. MySQL provides a python connector for this purpose, which can query and amend the data stored within the database through a python program, in this case a WSGI instance. The MySQL connector is able to execute MySQL commands and queries from within an executing python script. It can be used to execute individual commands, see Algorithm 5, or execute SQL files, see Algorithm 6.



*Algorithm 5 - SQL query through Python script*



*Algorithm 6 - Executing a SQL file through Python script*

### Reading diaries

In the project requirements, the contents of the reading diaries were outlined. However, for them to be effective, they will need to have the correct content. To help establish what was needed, I asked my third parties what would need to be included in these reading diaries. They established that they must have the title, author, overall rating, and any take-aways and reflections. Furthermore, the book genre, start and finish date, a rating of engagement, an ease-of-reading rating, a plot rating, and any quotes should be included. Natasha Kennedy also wanted to have “Links to particularly helpful sections/chapters if an E-book”, however, this project does support E-books so this is not needed. It was also said that sections cannot be compulsory. Outside of what the third parties said, the number of pages read, and time read for each entry should also be included. This time can then be used to work out the amount of time reading per day, or the level of engagement the user had with the book, which then could impact on the recommendations.

### Searching the database

As stated in the requirements for this project, the user needs to be able to search for books and authors that are stored within the system. There are different methods that could be used to do this, and these are outlined below. However, all these algorithms require some pre-processing beforehand to be able to function effectively. It is important to note that other methods could be used to improve the quality of the results, such as stemming, lemmatisation, removal of stop words, removal of contractions, and removal of numbers. These would significantly improve the results, being able to give multiple valid options, which do not directly match the input string. These methods are complicated and may reduce the performance of the search. Regardless, this adds significant complexity, especially if these are designed and built from scratch. For this reason, these features will be an extension only, but are still explained below.

#### Exact string matching

Exact string matching is the simplest way of searching for items within the system. It searches through the pre-processed data, stored within the database. The process of pre-processing the data is outlined later, in the TF-IDF section of this document. Pre-processing the input data and the data search items is important, as it removes the impact of punctuation and capitalisation of letters. Removing the stop words in the phrase also further reduces the dependence on the accuracy between the search query and the item, however its impact is likely to be less. Any discrepancies between the search query and the title or author’s name would then result in it not being found, including the capitalisation, which is improved by the pre-processing.

Following the pre-processing of the search query, it then iterates through all stored entries, and compares the search query to the current item. When a match is found, it is the search result. However, it may still be beneficial to iterate through the rest of the data to find any other matches, as it is still possible there are other items with the same name in the dataset. This system only works of exact matches of the pre-processed data, so any misspelt words, words missing, synonyms or any other differences between the two, excluding those remedied in the pre-processing will lead to the item not being found. Furthermore, it is unable to give any other suggestions of items that may still be valid or relevant, and hence is an exact searching method. By using this method, any matches given will have a certainty close to, or exactly 100%, as it has matched the query to the item exactly. It is possible that the pre-processing could result in false matches but should still be performed for the aforementioned reasons.

This approach, while simple and easy to implement, will not be used in the final website, because of lack of any other suggestions when an item is searched for. If the data does not exist, no result is provided, and is not useful for the users. Other methods outlined here can sometimes not produce a result but will provide other suggestions most of the time. This method, in comparison, will either produce the correct result or no result at all. However, by using a pass-fail test, this method could be used to augment the other methods if they experience issues whereby perfect matches are not always given as the top result.

#### Bag Of Words (BoW)

Bag of Words (BoW) is a natural language processing (NLP) method used for information retrieval and, in this case, can be applied to the search application. It includes a vocabulary of known words, and the model that it creates counts the frequency of each word in this vocabulary. With this data, it turns the text into fixed-length vectors (Engati, n.d.).

First, BoW pre-processes the input data, performing a stock set of operations, including converting to lowercase and removing non-alphabetic characters (Engati, n.d.), as well as any other operations that are required specific t the application. By doing this before tokenize the data, BoW avoids words being viewed as separate due to either punctuation or capitalisation differences, even if the items are the same. After preprocessing the data, the algorithm then performs tokenization, taking the string and splitting it into separate words. It then counts the frequency of the unique words (Engati, n.d.), producing a fixed length vector, , where n is the length of the vector. By vectorising the search input, these can then be compared and ranked giving the ordered search results.

The bag of words model is simple and easy to use (Engati, n.d.), and is normally less computationally expensive than other methods, as the calculations can be precomputed. The number of these calculations is also low, helping to reduce the cost. It does have its disadvantages, and according to (Engati, n.d.), its vocabulary library must be designed, where its size has an impact on the sparsity of the representations, the context is ignored as it disregards meaning and order, focusing solely on the number of occurrences. Furthermore, the context is also ignored, disregarding meaning and word order. Finally, it is difficult to model sparse representations because of a lack of information and computational issues.

Frequently occurring words can have a significant impact on the results. These common words are known as stop words (Ganesan, 2014). Removing these stop words can improve the performance of the algorithm as it removes any bias towards any results. The removal of these words would be performed in the data pre-processing part of the search.

This model will not be used for the information retrieval function in the project. This is because, as previously mentioned, the impact of the stop-words can significantly skew the search results. Furthermore, without rating the importance of each word that is used, the absence of significant words in a search query may then lead to a valid result being missed. However, the simplicity of this approach is beneficial.

#### Google Universal Sentence Encoder (Google USE)

Google Universal Sentence Encoder (Google USE) is a type of vectorizer developed by (Cer et al., 2018) that “encodes text into high-dimensional vectors that can be used for text classification, semantic similarity, clustering and other natural language tasks” (Google, n.d.). For the use of the search feature in this project, it would be used for its semantic similarity ability, comparing the book titles from the database and the inputted search query. This means, instead of matching by words in the statement like the other methods listed, it takes the meaning of the sentence and uses this to make the matches. An example of this can be found in A Skinful of Shadows (Francis Hardinge) and When Shadows Fall (Alex Gray). When searching for shadows dropping, BoW would see that both entries had shadows in the name and would rank them as equal. Google USE, however, could tell that they had different meanings and therefore would link the search query to When Shadows Fall, which is the more appropriate result. Figure 28 shows an example search using the system, comparing how similar sentences with different meanings elicit different responses.

Chart, treemap chart

Description automatically generated with medium confidence

*Figure 28 - Sentence similarity scores using Google USE embeddings (Cer et al., 2018)*

Google USE is not a single, stand-alone method, but instead has multiple different versions, one of which uses a transformer encoder and the other using a Deep Averaging Network (DAN). These each provide a different balance of computational resource requirements and accuracy. Transformer encoding is more accurate but has a high computational requirement, whereas DAN is less accurate but requires less computational power (Mahapatra, 2019). However, compared to the other listed methods, it is still expensive.

For the deployment of this feature in the project, Google USE will not be used, even though it is freely available on TensorFlow. Author names do not require the semantic similarity comparisons that this can provide, and in fact, could be detrimental to the performance. Furthermore, for some book titles, there is not necessarily a valid semantic meaning that can be drawn from them, which could again reduce the performance. Furthermore, the server is running on a Raspberry Pi 3b or PythonAnywhere which have limited computational power, and using a demanding searching algorithm could hinder its overall performance. This would not provide an enjoyable user experience and should be avoided where possible.

#### Term Frequency – Inverse Document Frequency (TF-IDF)

Term frequency – inverse document frequency (TF-IDF) is a measure used in information retrieval to quantify the importance or relevance of a string representation in a corpus (Simha, 2021). It vectorises both the corpus data and the search query and uses them to compare the two. It can take into the account the frequency of the unique words in the titles, in a similar way to the Bag of Words model but can also consider how common the words are. This means that the more unique the word, the greater the influence it has on the search results and therefore this method can provide a more accurate result than BoW. Furthermore, this increased accuracy does not require a significant increase in computational cost. When providing the matches, it can give several suggestions, regardless of whether the exact match has been found or not. For this reason, TF-IDF is considered a fuzzy searching method (Brans, 2022).

This is the method that will be used in the final system because it is able to give accurate results, at a low cost, while also being able to give reasonable alternative suggestions if no item is found. Furthermore, adjusting for the importance of each word improves the accuracy, which could improve the search results. Below, is a detailed explanation of how TF-IDF works, and all of the steps required to use it within the searching feature within the website. Some of the features that are outlined below are extensions, which may be beneficial or detrimental to the performance. For this reason, I coded a short, simple implementation of this algorithm to use as a basis to test whether each of these features where necessary, the code for which can be found in Appendix B.

##### Data pre-processing

Data preprocessing is the process of preparing data ready for use in the information retrieval algorithm. The data must be in a consistent form for whatever model that is used, as any differences between the format of the search query and data in the corpus may affect the results. It also enables other processes to be applied to the data to improve the quality of the results, such as stemming and lemmatisation, to unify different word formats, widening the possible search results. This process follows a number of different steps, which are not all required to produce an effective searching feature. These are detailed below.

###### Tokenisation

The tokenisation stage is an important part of data pre-processing for an information retrieval algorithm. The string that is entered by the user into the search bar needs to be split into individual elements, or tokens. It is the first step that must be taken, like any other Natural Language Processing (NLP) problem. It takes the unstructured input data and splits it into sections that can be considered discrete elements (Menzli, 2021). This can be performed in various ways, such as splitting into sentences, or into words. In this case, the search query will be split into words, as it is designed to search for book titles or author names, so have a limited length. According to (Blaisdell, n.d.), book titles should be five words or fewer, so splitting into sentences is unnecessary. Furthermore, there are different forms of Tokenization, such as white space and punctuation-based tokenizers (Menzli, 2021). As previously mentioned, this system is designed for book titles and names rather than sentences where punctuation is important. Consequently, it will perform white space tokenization. However, it should still work with sentence inputs.

###### Data cleaning

The data needs to be cleaned, which is dependent on the type of data, and given that the data in this use case is textual, this process is even more important (Rastogi, 2022). While there is not a definitive outline for the steps that need to be taken, (Rastogi, 2022) suggests some common actions that need to be taken, such as converting text to lowercase and removing punctuation, numbers, spaces, emojis and emoticons. This then avoids any issues relating to punctuation being included in the tokens when splitting the searching string into an array.

###### Stemming and Lemmatisation

Because of grammatical intricacies, different words will have different forms, such as play, playing and played, and words with similar semantic meanings. It can be beneficial to have these types of words unified into one so that when searching for any one of them, all versions can be applied. Both stemming and lemmatization aim to “*reduce inflectional forms and sometimes derivationally related forms of a word to a common base form*” (Manning et al., 2009). This allows for a single search to target more relevant results and increase the chances of a correct result.

Stemming is a way to reduce the number of different word forms by removing suffixes that affect the meaning of the word (Manning et al., 2009). All the different stemming algorithms follow their own strict set of rules when stemming, an example of which is shown in Figure 29. Any variance from these rules can lead to a significant change in the performance, as it only works due to the unification of similar words, and therefore adding variation will impact recall instead of improving it. There are many different algorithms that already exist to perform this such as Porter’s algorithm (Porter, 2006), and each will provide different results, see Figure 30.

Text

Description automatically generatedText

Description automatically generated

*Figure 29 - The rules used in Porter’s algorithm (Porter, 2006)*

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 30 - A comparison of three stemming algorithms on a sample test (Manning et al., 2009)*

Lemmatization is another method which helps unify similar words with the similar semantic meaning. It uses a set vocabulary and analysis of word structure, with the goal of reducing the word to the base word (Manning et al., 2009). In a similar manner to stemming, this allows for a single word to be compared to multiple other words that have the same meaning. Unlike stemming, however, it does this semantically, which can allow the meaning of words and sentences to be kept the same, and therefore not altering the search.

For this system, it will only be included as an extension to the project. This is because it adds added complexity and a higher computational demand for every search. It may not improve search performance, however using the program in Appendix B, it does seem to improve the performance. The NLTK library in Python does have various stemmers and lemmatizer, so could be included, but a purpose-built solution may perform better. The added complexity is not necessary. For this reason, it will not be included.

###### Stop words.

Stop words are a list of words that are commonly used in a language (Ganesan, 2014). These are often removed so that the algorithm can focus on the important words in the document (Ganesan, 2014). Leaving these words in the data could skew the search results, and while this is more significant with BoW, it can still affect the results using TF-IDF, as if it does not appear in all titles, it will therefore be considered significant. With book titles, it is possible to have title made solely of stop words, such as *The How* by Yrsa Daley-Ward may end up being left blank. For this reason, these words should be left in the search data. It also increases the complexity.

###### Contractions, slang and numbers.

In the English language, there are different ways of abbreviating certain words, such as do not being abbreviated to don’t. These are known as contractions, and they can allow for different ways of representing words, and this can make comparisons harder, and increase the dimensionality of the data. Removing this can allow for more inputs to be matched to book titles if the entered word is contracted, or the title is contracted and the other is not. This can allow for more accuracy.

Slang words could also be searched. Converting these back into traditional words could mean that it avoids having the slang entered and the target title not using the same slang. This can help reduce the dimensionality. However, with books, the slang may be part of the title, and a defining way of identifying them. Removing this could make it harder to identify books correctly, so for this reason, the slang should not be removed. These can both be performed using the contractions library in python, using the fix method. When using this, the parameter slang must be set to False so that it is not converted for the reasons previously mentioned.

Numbers are another way that the variations in data can be achieved. For example twenty two, twenty-two and 22 all have the same meaning but would be tokenized into different words. To avoid this, these should be removed from the string and replaced with numeric characters. There are modules that can do this, such as words2num, wordtodigits and word2number, but these require the number to be separate from any punctuation, so could not be applied to Catch-twenty-two. This is not likely to be entered, and it is most likely that numerical digits would be entered anyway, so this is not essential. However, wordtodigits’s convert method would be the best as it can have an entire string passed, not just the number and convert it. This would have to be done before tokenization, however.

This adds significant complexity to the project, and, while it does improve the accuracy of the results, does not warrant the additional complexity that it brings. For this reason, it does not need to be included, except as an extension.

###### Spelling mistakes

Many people do make spelling mistakes when typing, and it would be a good feature of the search feature if it could correct these. There are many pre-existing algorithms that can do this, such as BK-Tree, and Norvig’s algorithm. They are not particularly fast, and through a website, a slow response time is noticeable. To avoid this another option is SymSpell, which is faster than the aforementioned algorithms. This is a free-to-use python library, so could be used. It could, however, reduce accuracy on titles with unusual names that are not in its dictionary. To avoid this, it would have to be run on both the query and the title. Furthermore, it would also need the website to be able to show the user what was corrected and give the option to search exactly for what they entered. Furthermore, every time the program starts, it needs to load the words to use for the corrections before being used. Loading these is slow, and on the Raspberry Pi, it took an average of 18.5526 seconds to load into memory, before being used.

Spell checking and corrections would be beneficial for the users, but as it is not part of the system requirements it is not necessary. It significantly increases the complexity, and for this reason, it should only be done as an extension.

##### Term Frequency

The term frequency is done in a similar way to the BoW algorithm. It is the frequency that a specific term appears in the document, in this case, the book title (Simha, 2021). There are different measures for this frequency, but in this case, it uses the term frequency which is adjusted according to the length of the document (Simha, 2021). As a result, all the book titles in the database will have their own TF values. Equation 1 shows how to calculate the TF value, where is the number of occurrences of that term, and is the total number of terms.

(1)

##### Inverse Document Frequency

The inverse Document Frequency (IDF) is part of TF-IDF calculations that differentiates it from BoW. It looks at how unique or otherwise, the word is within the entire corpus (Simha, 2021). Frequent words such are given significantly less importance. Furthermore, if a term appears in all documents, the IDF value will then be zero, assuming it is not an altered method, and the TF-IDF will also be zero. As a result, the word will be ignored entirely. This is why removing stop words from the corpus may not be necessary. There are different ways to calculate this. Equation 2 is the common way to calculate it and Equation 3 is the method used in the Scikit learn library and in this implementation, where is the number of documents, , in the corpus, .

(2)

(3)

In this case, the second Equation 3 will be used, as it negates some issues that are found with Equation 1. In Equation 1, it is possible for the word not to appear in any of the documents, so without adding one to this number, it would result in a division by zero error (Prakriti, 2022). Consequently, the numerator must be incremented by one, so the answer remains constant. Finally, an additional one is added after the division, within the log. This is Laplace smoothing, preventing any values from being zero, to help manage data sparsity (Prakriti, 2022). Consequently, the minimum value, assuming base ten is used, is approximately 0.301. It is important to note, however, that the base used is not important, as any base will eventually yield the same result after the cosine similarity operation if it is consistent.

##### Term Frequency – Inverse Document Frequency (TF-IDF)

Having calculated all the TF and IDF values, it is simple to calculate the TF-IDF values. TF-IDF is calculated by multiplying the TF values by the IDF values (Equation 4) (Simha, 2021).

(4)

##### Vectorising the search query

“*Cosine similarity is a measure that quantifies the similarity between two vectors*” (Alake, 2021). This allows the multi-dimensional vectors that are produced by the vectorisation of both the titles and search queries to be compared to each other. Equation 5 is the equation to calculate the cosine similarity, where and are the comparison vectors, and is the number of dimensions of the vector.

(5)

(6)

Taking of the result of the cosine similarity, will give the angle between the two vectors. However, this is not necessary for this, as the values provided without this are still usable. The values that can be obtained can be between 0 and 1 inclusive, and as the value tends towards one, the vectors are more like each other (Alake, 2021). Therefore, a vector with a cosine similarity of one is an exact match, whereas a value of 0 means that they are a 0% match Figure 27. By multiplying this by the cosine similarity by one hundred (Equation 6) (Alake, 2021).

Diagram

Description automatically generatedDiagram

Description automatically generated

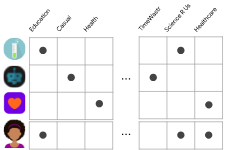
*Figure 31 - Representation of cosine similarity (Alake, 2021)*

### Recommendations

The main focus of this project is the recommender system, and without it, the project has little function. There are many different methods that could be used to achieve this, which are outlined below. However, this project does have a time limit, which should be taken into account when choosing the approach to take. Furthemore, the computational cost of the recommender system should also be considered, due to being limited to lower-end hardware. It is important to note that it is also possible to generate recommendations using deep neural networks, as explained by (Google for Developers, 2022c), however, this is very complicated, and may also not work very quickly on the available hardware. It would allow for easier use of implicit ratings and data, however, given the additional complexity that this provides, will not be considered for the project.

#### Content-Based filtering

The simplest approach to calculating the recommendations is to use content-based filtering. According to (Google for Developers, 2022b), it recommends items similar to what a specific likes based upon their previous interactions or explicit feedback. Each user has a rating for each feature, and each book has a rating for each feature in the same way, represented in the same feature space, see Figure 32.

**

*Figure 32 - Example feature space (Google for Developers, 2022b)*

To generate these recommendations, a similarity metric needs to be used, such as dot product. The system then must be set up to score each item according to the similarity metric (Google for Developers, 2022b). If the user embeddings are known as , where is a vector for a user’s embedding, and the item embeddings are known as , the dot product between the two, is the number of features that are in both vectors, assuming both vectors are binary vectors, with 1 indicating that the feature is active, and 0 indicating that it is not, consequently the higher the dot product the higher the similarity is (Google for Developers, 2022b).

(Google for Developers, 2022b) outlines multiple benefits with recommendation models using this approach. It does not need any additional data about other users to work correctly, as recommendations are user specific. This user-specificity also helps scalability with large user bases. The model also can recommend uncommon items to the users, which have very few other interested users. (Google for Developers, 2022b) further outlines the disadvantages with this approach. The feature embeddings for the items need to be hand-engineered, which requires domain knowledge. It therefore limits the model to the quality of the embeddings. Furthermore, the model cannot expand on the interests of the user, as it is limited to the user’s existing, known preferences.

This approach to the recommendations will not be taken, as it requires significant domain knowledge, which would require all the entered books to be read, and matched against a list of genres. This will be time consuming and would therefore limit the number of books that can be entered into the system. Furthermore, it cannot expand the user’s reading preferences, and may keep them within a certain author or genre. This then partially defeats the point of having the recommender system, if it only recommends books very close to their interests, and not necessarily introducing them to new authors or genres.

#### Collaborative Filtering and Matrix Factorisation

Collaborative filtering negates most of the limitations posed through content-based filtering. It allows for the learning of both the user embeddings and the item embeddings simultaneously, which removes the need for any domain knowledge (Google for Developers, 2022a). This would therefore allow for more books to be added to the system, as the genre weightings would not need to be known.

The recommendations can be found from a known feedback matrix, , where is the number of users and is the number of items that the model is meant to learn. From this matrix, the model learns both the user embedding matrix, , and the item embedding matrix , (Google for Developers, 2022a). These are learned so is an accurate approximation of the feedback matrix (Google for Developers, 2022a).

To perform this, an objective function needs to be chosen, and minimised in order to find the user and item embeddings that form the approximation of the feedback matrix (Google for Developers, 2022a). An example objective function is seen in Equation 7, where is the observed ratings in the feedback matrix (Rosenthal, 2016). is a hyperparameter which ensures that the two terms so that the objective is not dominated by either one, which is important as frequent queries, like heavy users may dominate the objective function (Google for Developers, 2022a).

(7)

This then needs to be minimised, which can be done using weighted alternating least squares (WALS). Explained by (Google for Developers, 2022a), this process involves first generating random values for both the user and item embeddings, assuming one is correct, and using this to calculate the other, using the derivative of the loss function. After that, fixing the previously calculated matrix and using this and calculating the other matrix. Iteratively doing this will ensure that the two matrices will converge to give the feedback matrix. Equation 8 gives the derivative for the user embeddings and Equation 9 gives the derivative for the item embeddings (Rosenthal, 2016).

(8)

(9)

This approach has multiple advantages, as outlined by (Google for Developers, 2022a). It does not require any domain knowledge, as both the user and item embeddings are learnt from the known feedback matrix. It is easy to start generating recommendations from this, as it only needs the feedback matrix to calculate both sets of factors. Finally, it also can generate recommendations outside the user’s existing preferences, as other similar users have expressed an interest in that item.

(Google for Developers, 2022a) also outlines some disadvantages with this approach. It is very difficult to add new users or include other factors within the recommendations, in this case whether a user is following an author. The difficulty with adding new users is not an issue, as each time new recommendations need to be generated, the model needs to be retrained, to include user’s new reviews and activity on the site. Combining this with initially asking new users for their preferences before generating recommendations will allow for new users to be added fairly easily.

This approach is the most suitable for this project, due to the fact that it can recommend user’s books outside of their existing preferences, and widen the range of genres or authors they read books from. It would therefore be able to introduce them to new genres and authors that they may have otherwise not found, which is better than the content-based filtering approach. Furthermore, not requiring any domain knowledge helps ensure that the project is feasible to get enough books with, as they do not have to be read, and have their genres ranked.

# Documented Design

This project can be split into two parts, frontend and backend. Each of these have their own requirements, and processes to use. For clarity, this section has been divided into the frontend components and backend components.

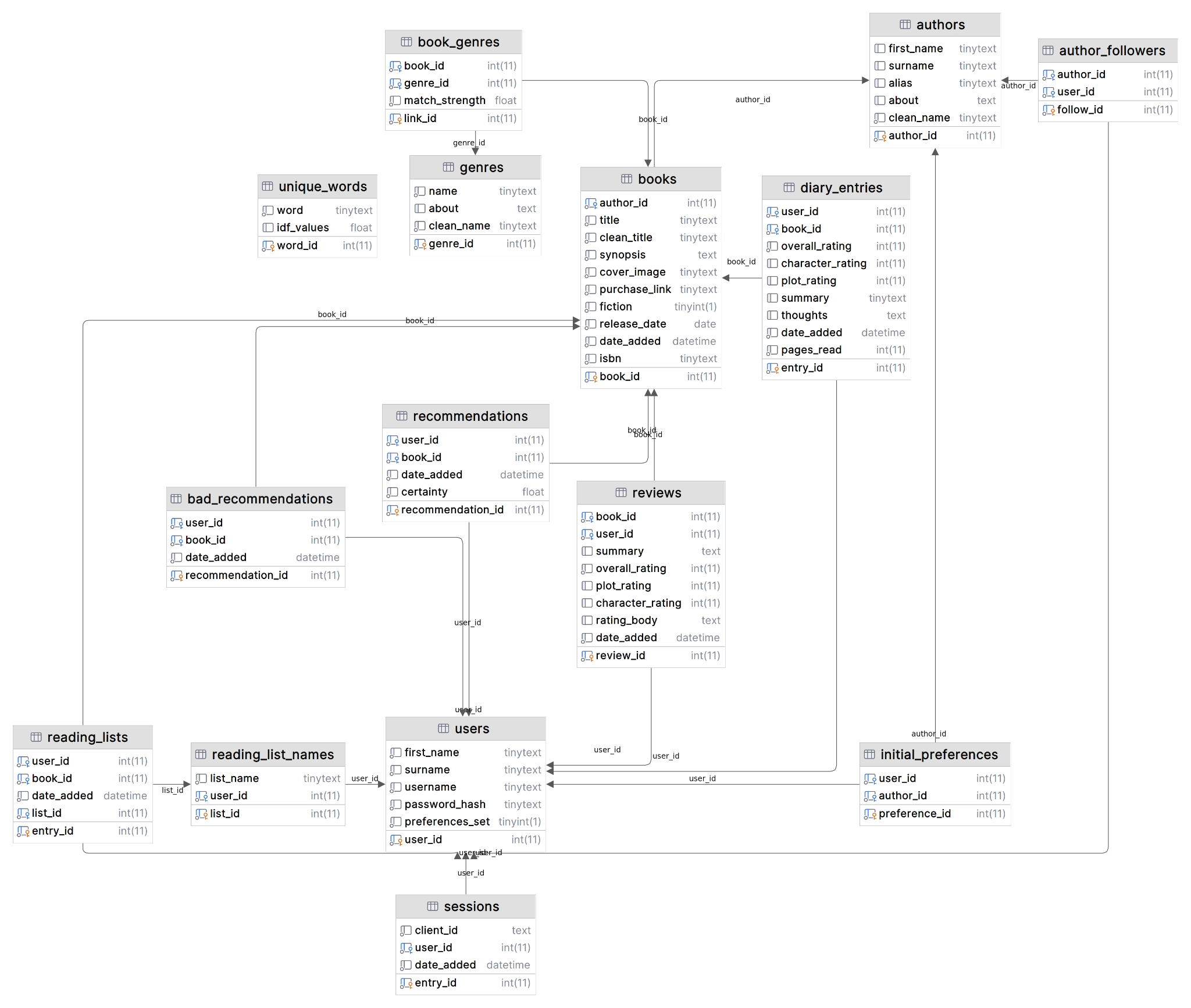
## Backend

This section explains how each of the sections of this project will be implemented. It gives flowcharts and pseudo code to explain the logic behind them. It also includes the expected setup for the lighttpd server, class diagrams and definitions, and the database structure, and some of the queries. The psuedocode, flowcharts and MySQL queries have been included for the more complex tasks. It has been excluded for simpler procedures, such as reordering the results from database queries into JSON to be sent to the client, as these are likely to be taking results from a query, and adding them into the corresponding fields. These functions will still be included in the class definitions

### MySQL Database

The database forms a significant part of the project. Without it there would not be any way to store user data, or data regarding the books safely, efficiently and reliably. It is important to note that this project was partially developed using pythonanywhere.com, which does not support the use of triggers. Being able to use triggers would have been able to simplify some of the queries, but could not have been used.

#### Table definitions



| SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0;  SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0;  DROP TABLE IF EXISTS authors;  DROP TABLE IF EXISTS author\_followers;  DROP TABLE IF EXISTS users;  DROP TABLE IF EXISTS genres;  DROP TABLE IF EXISTS book\_genres;  DROP TABLE IF EXISTS recommendations;  DROP TABLE IF EXISTS initial\_preferences;  DROP TABLE IF EXISTS bad\_recommendations;  DROP TABLE IF EXISTS books;  DROP TABLE IF EXISTS reading\_lists;  DROP TABLE IF EXISTS reading\_list\_names;  DROP TABLE IF EXISTS diary\_entries;  DROP TABLE IF EXISTS reviews;  DROP TABLE IF EXISTS sessions;  DROP TABLE IF EXISTS unique\_words;  CREATE TABLE authors (  author\_id INT NOT NULL AUTO\_INCREMENT,  first\_name TINYTEXT,  surname TINYTEXT,  alias TINYTEXT,  about TEXT,  clean\_name TINYTEXT NOT NULL,  PRIMARY KEY (author\_id)  );  CREATE TABLE author\_followers (  follow\_id INT NOT NULL AUTO\_INCREMENT,  author\_id INT NOT NULL,  user\_id INT NOT NULL,  PRIMARY KEY (follow\_id),  FOREIGN KEY (author\_id) REFERENCES authors(author\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id)  );  CREATE TABLE users (  user\_id INT NOT NULL AUTO\_INCREMENT,  first\_name TINYTEXT NOT NULL,  surname TINYTEXT NOT NULL,  username TINYTEXT NOT NULL,  password\_hash TINYTEXT NOT NULL,  preferences\_set BOOLEAN NOT NULL DEFAULT FALSE,  PRIMARY KEY (user\_id)  );  CREATE TABLE genres (  genre\_id INT NOT NULL AUTO\_INCREMENT,  name TINYTEXT NOT NULL,  about TEXT,  clean\_name TINYTEXT NOT NULL,  PRIMARY KEY (genre\_id)  );  CREATE TABLE book\_genres (  link\_id INT NOT NULL AUTO\_INCREMENT,  book\_id INT NOT NULL,  genre\_id INT NOT NULL,  match\_strength FLOAT NOT NULL,  PRIMARY KEY (link\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id),  FOREIGN KEY (genre\_id) REFERENCES genres(genre\_id)  );  CREATE TABLE recommendations (  recommendation\_id INT NOT NULL AUTO\_INCREMENT,  user\_id INT NOT NULL,  book\_id INT NOT NULL,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  certainty FLOAT NOT NULL,  PRIMARY KEY (recommendation\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id)  );  CREATE TABLE initial\_preferences (  preference\_id INT NOT NULL AUTO\_INCREMENT,  user\_id INT NOT NULL,  author\_id INT NOT NULL,  PRIMARY KEY (preference\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id),  FOREIGN KEY (author\_id) REFERENCES authors(author\_id)  );  CREATE TABLE bad\_recommendations (  recommendation\_id INT NOT NULL auto\_increment,  user\_id INT NOT NULL,  book\_id INT NOT NULL,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  PRIMARY KEY (recommendation\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id)  );  CREATE TABLE books (  book\_id INT NOT NULL AUTO\_INCREMENT,  author\_id INT NOT NULL,  title TINYTEXT NOT NULL,  clean\_title TINYTEXT NOT NULL,  synopsis TEXT NOT NULL,  cover\_image TINYTEXT NOT NULL,  purchase\_link TINYTEXT NOT NULL,  fiction BOOL NOT NULL,  release\_date DATE NOT NULL,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  isbn TINYTEXT NOT NULL,  PRIMARY KEY (book\_id),  FOREIGN KEY (author\_id) REFERENCES authors(author\_id)  );  CREATE TABLE reading\_list\_names (  list\_id INT NOT NULL AUTO\_INCREMENT,  list\_name TINYTEXT NOT NULL,  user\_id INT NOT NULL,  PRIMARY KEY (list\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id)  );  CREATE TABLE reading\_lists (  entry\_id INT NOT NULL AUTO\_INCREMENT,  user\_id INT NOT NULL,  book\_id INT NOT NULL,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  list\_id INT NOT NULL,  PRIMARY KEY (entry\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id),  FOREIGN KEY (list\_id) REFERENCES reading\_list\_names(list\_id)  );  CREATE TABLE diary\_entries (  entry\_id INT NOT NULL AUTO\_INCREMENT,  user\_id INT NOT NULL,  book\_id INT NOT NULL,  overall\_rating INT NOT NULL,  character\_rating INT,  plot\_rating INT,  summary TINYTEXT,  thoughts TEXT,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  pages\_read INT NOT NULL DEFAULT 0,  PRIMARY KEY (entry\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id)  );  CREATE TABLE reviews (  review\_id INT NOT NULL AUTO\_INCREMENT,  book\_id INT NOT NULL,  user\_id INT NOT NULL,  summary TEXT,  overall\_rating INT NOT NULL,  plot\_rating INT,  character\_rating INT,  rating\_body TEXT,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  PRIMARY KEY (review\_id),  FOREIGN KEY (book\_id) REFERENCES books(book\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id)  );  CREATE TABLE sessions (  entry\_id INT NOT NULL AUTO\_INCREMENT,  client\_id TEXT NOT NULL,  user\_id INT NOT NULL,  date\_added DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  PRIMARY KEY (entry\_id),  FOREIGN KEY (user\_id) REFERENCES users(user\_id)  );  CREATE TABLE unique\_words (  word\_id INT NOT NULL AUTO\_INCREMENT,  word TINYTEXT NOT NULL,  idf\_values FLOAT,  PRIMARY KEY (word\_id)  );  SET FOREIGN\_KEY\_CHECKS=@OLD\_FOREIGN\_KEY\_CHECKS;  SET UNIQUE\_CHECKS=@OLD\_UNIQUE\_CHECKS; |
| --- |
| The SET queries at the start of the query remove checking for foreign key constraints, and for unique constraints. This prevents the order of deleting the tables at the start not matter, as the foreign keys linking the entries in the tables are ignored. The final SET statements revert this change, so that it is not possible to create entries in other tables referencing a record that does not yet exist.  The DROP IF EXISTS statements delete all the tables used within the database. DROP IF EXISTS is used over DROP, so that if the database is being created for the first time, it does not raise any errors.  The authors table stores data about each of the authors that are stored within the database. The author\_id field is the identifying index for the table. The INT NOT NULL AUTO\_INCREMENT, in combination with the PRIMARY KEY constraint, means that each time a new record is added, the author\_id is automatically one greater than the previous, indexed from one. The PRIMARY KEY also applies a UNIQUE constraint, and INDEX constraint, which should also increase query speed.WWW The first\_name, surname and alias columns can be combined to generate the name to display. They can be null, as it is possible for only an alias to be given, or first name or surname only. They are TINYTEXT as they are not going to be very long as they are names. The about data can also be null, which stores a blurb about the author to be displayed on their about pages. TEXT is used as it is large enough to store a very long article about the author. The CLEAN NAME is used for the search feature, so it does not need to be cleaned every time it is loaded, again TINYTEXT as it is not going to be very long. It cannot be null, as it must be present for the search feature.  The author\_followers table stores the relationships between users and the authors that they are following. A user\_id corresponds to an author\_id, which indicates that they are following the author. An author\_id and user\_id can appear any number of times. The follow\_id is the PRIMARY KEY for the table, and with auto\_increment should increase automatically every time a new record is added. The author\_id is a FOREIGN KEY, referencing the author\_id column from the authors table. The user\_id is also a FOREIGN KEY, referencing the user\_id column from the users table. These FOREIGN KEY constraints ensure that a record cannot be inserted for a user or author that does not exist within the database. Both the author\_id and user\_id columns cannot be null, as it is illogical for records to exist within the table if a user\_id or author\_id is not specified.  The users table contains all the data about the users. It contains their first\_name, surname, and username, which are all TINYTEXT, as they are unlikely to be longer than 255 characters. They also cannot be null, as they must be known for the login. The password\_hash field contains the hash of the user’s password. The hash is stored as it cannot be reversed engineered to find the true password, which should be more secure. It is a TINYTEXT, as it will not be very long. It also cannot be null, as the user must have a password. The preferences\_set column is a boolean value, which indicates whether they have specified their initial preferences. If they have, the system should generate them recommendations, otherwise they should not, as they have not expressed any authors that they like. It defaults to FALSE, as when a new user first signs up they will have not immediately specified their preferences. Finally the PRIMARY KEY constraint is applied to the user\_id, which, in combination with the auto\_increment should ensure that it increases automatically, and should be faster to access.  The genre table stores the genre data about each of the genres stored in the database. The name column stores the name of the genre, which is a TINYTEXT as it will not exceed 255 characters, and cannot be null, as the name must be specified. The about column is a TEXT, which gives a description of the genre, but can be null, as it is not necessary for the user to understand what the genre is about. The clean\_name column stores a cleaned version of the genre name, which is used for the search feature. It cannot be null as it must be present for the search to work. It is also a TINYTEXT as the name will not exceed 255 characters, and should, therefore, reduce the required amount of storage space per entry. It is pre-computed and calculated so that the search feature does not need to perform it every time, which should improve search performance significantly. The genre\_id is the PRIMARY KEY for the table, which, in combination with the auto\_increment, should automatically increase for each entry added to the table.  The book\_genres table provides a link between a book and all the genres in the database. It gives a match\_strength as a floating point number as a rating of how good the match for each genre is for the book. The genre\_id column is a FOREIGN KEY, which references the genre table, and the book\_id is a FOREIGN KEY which references the book table.. This prevents a record being added for a book or genre which does not exist. None of the columns can be null, as if they were it would be an unnecessary record as it would not be useful data. A match\_strength of null or a missing record would be the same as a match\_strength of 0. The ratings for this are saved in the database, as it is required to list the genres for each book, or the books that apply to a specific genre. A table for the users is not included, as it is never displayed, and would be used only for the recommendations. The model needs to be re-trained before recommendations are generated, so the user preferences are already stored in memory, and do not need to be stored in the database. Minimising the amount of personal data in the database is important for privacy and security.  The recommendations table stores all of the user’s recommendations. The PRIMARY KEY on the table is the recommendation\_id, which in combination with auto\_increment, will increase automatically and provide a unique identifier for each recommendation. Each entry links a book\_id to a single user\_id, and there can be multiple of the same book\_id or user\_id in the table. However, there should only be a maximum of one instance of a book\_id per user\_id in the table, so there cannot be the same book recommended to the same user twice. The user\_id is a FOREIGN KEY, which references the users table, and the book\_id is a FOREIGN KEY, which references the books table. This prevents a user or a book being inserted which does not exist within the database. The user\_id and book\_id cannot be null, as a recommendation must have a specified user and book, otherwise it does not have a purpose in the table. Furthermore, a certainty is stored, which is a floating point value between 0 and 1, to represent how good a recommendation each is for the user. This cannot be null, as it must be displayed to the user, as specified in the requirements for the project. Finally, the date\_added column stores that date that the recommendation was made, and is a DATETIME data type. It defaults to the time that the recommendation was inserted, so it does not need to be included explicitly in the INSERT statement. It is included in the table so that it can be displayed to the user, and recommendations can be deleted after a certain number of days, which ensures that the table does not grow uncontrollably as more recommendations are generated.  The initial\_preferences table stores a list of authors that the user has explicitly stated that they like, by linking an author\_id to a user\_id. The author\_id column is a FOREIGN KEY which references the authors table, and the user\_id is a FOREIGN KEY which references the users table. This prevents any records being inserted that reference a user or an author that do not exist within the database. The fields also cannot be null, as that would make the data stored in the record pointless. Finally, the preference\_id column is the identifying value for each relationship given, which, between the PRIMARY KEY constraint and auto\_incremenet, will remain a unique integer value.  The bad\_recommendation table stores a list of recommendations that the user has explicitly marked as a bad recommendation, so it can be included in the recommendations model, and avoid being generated as a recommendation in the future. It links a user\_id to a book\_id, which is not a good recommendation for the user. The user\_id is a FOREIGN KEY, referencing the user table, and the book\_id is a FOREIGN KEY, referencing the books table, which prevents a record being inserted for a user or book that does not exist. The date\_added column includes the date that the record was added, so that it can expire after a while. It defaults to the time that the record was created, so it does not need to be explicitly stated in the INSERT statement. The expiry ensures that books can be re-recommended to users after a long period of time as their preferences may have changed, so it could be a good recommendation again. The recommendation\_id is the PRIMARY KEY for this table, which should automatically increase due to the auto\_increment on the column.  The books table stores the data about all the books in the database. The book\_id column is the PRIMARY KEY for the table, which automatically increases, indexed from one, every time a new record is added. The author\_id is a FOREIGN KEY which references the author table. It links the author who wrote the book to the book. The title column stores the title for the book, as a TINYTEXT as it is unlikely to exceed 255 characters long. It cannot be null, as the book must have a title. The clean\_title column is also a TINYTEXT as it would be the same length, or shorter than the normal title. It is stored so it can be used by the searching algorithm, and not needing to be calculated every time, which should help to improve searching performance. Again, it cannot be null as it must be present for the algorithm. The synopsis column stores the blurb for the book. It cannot be null, as the book must have a synopsis. It is a TEXT datatype, as it could be significantly longer than 255 characters, and less than 65,535. The cover\_image column holds an address to the cover\_image for the book. This could be stored locally, or a link to an external image outside of the web server. It stores the link instead of the image as it is simpler and less likely to harm query performance than large image files. It is a TINYTEXT as it is unlikely to exceed 255 characters, and cannot be null, as there must be an image to display to the user. Similarly, the purchase link holds a URL to an external website where the user can buy the book. This is also a TINYTEXT as it is unlikely to exceed 255 characters, and is not null, as it must be present as specified in the objectives. The release\_date column stores a DATE, which stores the date that the book was published, so it can be displayed to the user. Again, it cannot be null, as it must be displayed. The date\_added field stores a DATETIME value, which defaults to when the record was added to the database. Finally the isbn column stores the ISBN number of the book. It is stored as a TINYTEXT as it will not exceed 255 characters, and being text avoids issues with losing leading zeros. It also cannot be null, as it must also be displayed to the user, as specified by the requirements.  The reading\_list\_names table stores each of the names for the users reading lists. This allows for the user to have custom lists. The list\_id column is the PRIMARY KEY for the table, which identifies the list name of an entry into a reading list. The list\_name is a TINYTEXT as it is unnecessary for list names to exceed 255 characters, and cannot be null, as any list must have a name. It also has a user\_id field which is a FOREIGN KEY, referencing the users table, which ensures that the user exists. It also links the list name to a specific user, preventing users from viewing each other’s lists. The user\_id cannot be null, as each list name needs to be associated with a user. When a new user is created, there should be a ‘Currently Reading’, ‘Want to Read’, and ‘Have Read’ automatically added. This could be done with triggers, but as previously mentioned, these cannot be used.  The reading\_lists table stores all of the entries into all of the user’s lists. The entry\_id field is the PRIMARY KEY for the table, and identifies each entry into a reading list, which, in combination with the auto\_increment, should automatically increase an remain a unique identifier for each record. The user\_id column which links each entry to a specific user, and although it is not strictly necessary, it can be used as a further safeguard to prevent access from users who’s do not own the list, as both the list id and user id must be known. The list\_id column is a FOREIGN KEY which references the reading\_list\_names tables, which links the list to a specific user’s list. The date\_added column is a DATETIME value, which defaults to the time the entry was added. This is the time that the entry was added to the list. The book\_id column is a FOREIGN KEY which links to the books table. It links the entry to a specific book. Finally, the user\_id column is a FOREIGN KEY which links to a specific user in the user tables. The FOREGIN KEYS prevent entries being made referenceing records that do not exist in the database.  The diary\_entries table stores the diary entries for all of the users. The entry\_id column is the PRIMARY KEY for the table, which forms the unique identifier for each of the diary entries. The user\_id column is a FOREIGN KEY which references the user\_id. It links a diary entry to a specific user. The book\_id column is also a FOREIGN KEY that references the book table, and links a diary entry to a specific book entry. These FOREIGN KEYs prevent entries being added to the table without the the user or book existing in the database. The overall\_rating column is a INTEGER, as the project does not need to support fractional ratings. It also cannot be null, as the diary entry must have a specified overall rating. The character\_rating and plot\_rating are both integers for the same reason, however can be null, as they are not required for the diary entry. The summary column stores a brief overview of the diary entry. It is a TEXT datatype, however, as it could be difficult to write a summary that is useful, and is under 255 characters. The thoughts column stores the detailed entry information, and is therefore a TEXT datatype, so the user is not realistically limited in the length of the entry. Both the summary and thoughts columns can be null, as the user is not required to provide these to add an entry. The pages\_read column is an integer, which stores the number of pages that the user has read. Finally the date\_added column defaults to the time that the record was inserted into the table. This is so that the entries can be ordered by the time that they were made. The default value makes the INSERT query simpler.  The reviews table is very similar to the diary\_entries table. The review\_id is the PRIMARY KEY which is used as a unique identifier for each entry. The book\_id and user\_id are the same as in the diary\_entries table, as well as the summary, overall\_rating, plot\_rating and character\_rating. The rating\_body is the detailed main body of the review, which contains all of the users beliefs about the book. The date\_added is the same as the diary\_entry table, which is a record for when the review is left. This can then be used to order the reviews for when they are displayed on the about page.  The sessions table keeps a record of all the active connections to the server. It ensures that the user ids do not need to be sent to clients, which should increase the security. The entry\_id is the PRIMARY KEY, which should therefore be a unique identifier of the session. The client\_id column stores the client id, which is the value that the user should have access to. It is a TEXT as the length would have to vary depending on the number of users of the users, so cannot be assumed to be less than 255 characters long. The user\_id is a FOREIGN KEY, referencing the users table, which links each session to a specific user. It also ensures that a session cannot be created for a user which does not exist within the users table. The date\_added column is a DATETIME object that defaults to the time the record was added. This is used so that the sessions can be expired after a period of time, so users cannot keep reusing sessions forever. Removing expired sessions would also ensure that the table keeps a reasonable amount of entries, reducing required size.  The unique\_words table stores the list of unique words. The words are are from the clean names and titles from the genres and titles. They are stored for the TF-IDF search, which should improve speeds of the search. The word is a TINYTEXT, which stores the word in the table. It is a TINYTEXT as a single word is very unlikely to exceed 255 characters. The idf\_values column is a FLOAT, which stores the IDF values for the TF-IDF search. It uses a float over a decimal, as it should be able to offer a greater range of storable values, with a high enough level of precision. Any floating point errors would be small enough not to have any noticeable impact. This should help reduce search times as it does not need to be calculated every time. It can be null, as the word needs to be inserted before the idf values can be calculated. The word\_id is the PRIMARY KEY for the table, which gives each word a unique identifier. |

#### Database queries

This section contains some of the queries that are likely to be required in this project, limited to ones that are the most complex, or unique.. They have been grouped into loose sections, although there can be significant overlap between sections as they often require access to the data of other tables. The top row of the table gives the purpose of the query, the middle section gives the actual query, and the bottom section gives a description of how the query works, and any applicable reasoning and explanation of the techniques used.

##### Accounts

| **Get password hash and user\_id from a username** |
| --- |
| SELECT password\_hash, user\_id FROM users  WHERE username="<username>"; |
| This query gets the password hash and the user id from the users table. The WHERE clause gives the result for a single username. This should only ever give one result, as usernames must be unique. |

### 

| **Check whether username is already used** |
| --- |
| SELECT username FROM users  WHERE username="<username>"; |
| Gets the username from the users table, for a specific user as specified by the WHERE clause. This can therefore be used to check whether the username already exists in the database. This must be done, as a UNIQUE constraint cannot be applied to the username column in the user table, as it is a TINYTEXT. If there are any results, the target username is taken. If there are not any results, the username given is usable, and can therefore be used by the new user. |

### 

| **Update the expiry time of a session** |
| --- |
| UPDATE sessions  SET  date\_added=NOW()  WHERE client\_id="<session id>"; |
| This query updates the date\_added time to the current time for a specific session, specified by the WHERE clause. This can be used for when a session id is used, so it is known that the user is still active, and therefore the time before it expires can be reset. |

##### Authors

| **Add a follower for an author** |
| --- |
| INSERT INTO author\_followers (user\_id, author\_id) VALUES (<user id>, <author id>); |
| This query adds a new record into the author\_followers table, linking a user id and an author id together. This marks that the user, specified by the user id now follows the author specified by the author id. |

### 

| **Unfollow an author by user id and author id** |
| --- |
| DELETE FROM author\_followers  WHERE user\_id=<user id>  AND author\_id=<author id>; |
| This query unfollows a specific author, by a user id. It deletes a record from the author\_followers table, which removes the following link. It deletes by user id and author id, as there can be multiple users following the same author, and the same user following multiple authors. The follow id is probably not known, as it is shouldn’t be passed to the client, so it must be done by user id and author id. |

### 

| **Get the number of followers for a specific author by author id** |
| --- |
| SELECT COUNT(author\_id) FROM author\_followers  WHERE author\_id=<author id>; |
| This query gets the number of followers a specific author has, specified by the author id in the WHERE clause. It counts the number of occurrences of the specific author id, which would then give the number of followers an author has. |

### 

| **Get average rating, number of reviews, number of followers and all data from authors table by author id** |
| --- |
| SELECT authors.first\_name,  authors.surname,  authors.alias,  authors.about,  (SELECT count(author\_followers.user\_id) FROM author\_followers  WHERE author\_followers.author\_id=authors.author\_id) AS followers,  ROUND(AVG(reviews.overall\_rating), 2) AS average,  COUNT(reviews.overall\_rating) AS number  FROM authors  LEFT OUTER JOIN books  ON authors.author\_id=books.author\_id  LEFT OUTER JOIN reviews  ON reviews.book\_id=books.book\_id  WHERE authors.author\_id=<author id>; |
| This query most of the data required for the about page. It gets the author’s first name, surname, writing alias, their about from the authors table. It then has a subquery that calculates the number of followers, which is the same as the previous query to get the number of followers for a specific author. This does not require any joins as it is part of a subquery.  It also gets the average rating for all of their books. This is done by using the average function, built into MySQL to average the overall ratings for all the reviews that have been left for that specific book. This is limited by the joins and the WHERE clause limiting the results to a single author. It then rounds this result to two decimal places using the round function. This is done so that the result is accurate enough for the user, but is a sufficient number of decimal places to be displayed. The query also gets the number of ratings that have been left for the user. It also gets the number of reviews that have been left for all of the books that the author has written. This is done by counting the number of overall ratings present for the books that the specified author has written. It uses overall ratings as the column used has no effect on the results, and is already used so may, therefore, be faster. To be able access these values, the query must be joined with the books table. This is done using a LEFT OUTER JOIN as it is possible for an author to exist within the database and not have any books written for it. It is on the the author id matching the author id of the book, which limits the results to only the books that have been written by the specified author. Using an INNER JOIN for this, and the author not having any books would mean that the query does not yield any results, which would cause errors as it is still a valid author. It also needs to be joined to the review table, again using a LEFT OUTER JOIN. This type of join is used as a book may have not had any reviews left for it, but should still be considered. It is joined on the book id for the review matching the book id of the books table.  The results are limited to a single author by the WHERE clause, matching the author id. This means that this query can only ever yield one or zero rows, one if the author does exist in the database, and zero if it does not. The averages and counts mean that, even though they are applied to many different rows, they give a single value, and therefore the query will only contain one row per author. The number of authors is limited to one by the where clause, as the author id will be unique, so the query gives a maximum of one row. |

### 

| **Get genres that an author has written books for** |
| --- |
| SELECT genres.name  FROM genres  INNER JOIN book\_genres  ON book\_genres.genre\_id=genres.genre\_id  INNER JOIN books  ON books.book\_id=book\_genres.book\_id  INNER JOIN authors  ON authors.author\_id=books.author\_id  WHERE authors.author\_id=<author id>  LIMIT <maximum number genres>; |
| This query gets all the genres that an author has written for. It does this by selecting the genre name from the genre table. It has to be joined to the book\_genres table, as it needs to be known which genres apply to each book. This uses an inner join, as if the genre is not in the book\_genres table for that specific book, the genre has a match strength of 0, and the book does not belong to that genre. It is then joined to the books table, using an inner join, as a record cannot exist in the book\_genres table if the book does not exist. Finally it is joined to the authors table. This allows for the WHERE clause to specify all books that the author has written. It again uses an inner join, as a book cannot exist without an author. The WHERE clause limits the results to all books written by the specified author id. Finally, the LIMIT statement limits the number of genres displayed to a specified number, which prevents many genres being outputted. |

### 

| **Get highest rated books from a user’s followed authors** |
| --- |
| SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias,  ROUND(AVG(reviews.overall\_rating), 2) AS average\_rating  FROM books  INNER JOIN authors  ON books.author\_id=authors.author\_id  LEFT OUTER JOIN reviews  ON reviews.book\_id=books.book\_id  WHERE books.author\_id IN (SELECT author\_followers.author\_id  FROM author\_followers  WHERE author\_followers.user\_id=<user id>)  GROUP BY books.book\_id  ORDER BY average\_rating DESC  LIMIT <number of books>; |
| This query gets the summary data for all of the highest rated books from all of the author’s a specific user is following. It selects the book titles, book ids, cover image links from the books table.  It also gets the author’s first name, surname and writing alias from the authors table. This is done by joining the query to the authors table, with the author id from the authors table matching the author id from the books table. It uses an INNER JOIN as the book cannot exist without an author so it will not lose any data.  Additionally, it gets the average rating from the reviews table. It does this by using the average function to average the overall rating from the reviews table, followed by the round function to round the result to two decimal places. It does not try to use the plot rating and character ratings in the average as they are unlikely to have a noticeable impact on the results, but would increase the time the query takes. To be able to get data from the reviews table, it is joined to the reviews table, where the book id in the review entry matches the book id from the books table. It uses a LEFT OUTER JOIN, as it is possible for books not to have any reviews, and these should still be included in the results. The GROUP BY clause ensures that each row is a specific book, rather than the average of all the books in the query.  The results of the query are limited to authors of a specific user through the WHERE clause. It limits the results to books, whose author ids are in the author\_followers table for the specific user. These author ids are found by performing a subquery, getting all the author ids that a specific user follows from the author\_followers table. All of the author ids in this result will therefore be all the author follows, and therefore the outer where clause gets the results for all the books from all of the authors that they follow.  The result is ordered by the average rating of all the books, and then the LIMIT clause limits this to a specified number of books. This then means that the result is the highest rated books from the user’s followed authors. |

##### Books

| **Get genres for all books in the database** |
| --- |
| SELECT book\_genres.book\_id,  GROUP\_CONCAT(book\_genres.genre\_id) as genres,  books.author\_id  FROM book\_genres  INNER JOIN books ON books.book\_id=book\_genres.book\_id  GROUP BY book\_id; |
| This query gets all the genres that a book matches. GROUP\_CONCAT gives these as a comma separated list of the ids. The GROUP BY is also required for the concatenate function to give the correct result. It should be concatenated, as this means one query is able to get all the data for all of the books, instead of using a query per item. This approach is likely faster. |

### 

| **Get about data for a specific book** |
| --- |
| SELECT books.title,  books.cover\_image,  books.synopsis,  books.purchase\_link,  books.release\_date,  books.isbn,  authors.first\_name,  authors.surname,  authors.alias,  authors.about,  (SELECT COUNT(author\_followers.author\_id) FROM author\_followers  WHERE author\_followers.author\_id=books.author\_id) AS num\_followers,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Want to Read"  AND reading\_lists.book\_id=books.book\_id) AS num\_want\_read,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  AND reading\_lists.book\_id=books.book\_id) AS num\_reading,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Have Read"  AND reading\_lists.book\_id=books.book\_id) AS num\_read  FROM books  INNER JOIN authors  ON authors.author\_id=books.author\_id  WHERE books.book\_id=<book id>; |
| This query gets all of the relevant data for a specific book. The book is selected through the WHERE clause, which matches a specified book id. The results are then also limited to a maximum of one result, as the book ids will be unique due to the PRIMARY KEY constraint and the auto\_increment on the table.  It gets the book title, cover image, synopsis, purchase link, release date and the ISBN from the books table. It also gets the information for the author from the authors table (their first name, surname, writing alias and their about paragraph). To be able to get this, the query must be joined with the authors table, on the author id for the book, matching the author id from the authors table. An INNER JOIN is used, as it can offer faster results, and as it is not possible for a book to not have an author due to the not null and foreign key constraints, can be used as it would not prevent results appearing.  The query also gets the number of followers the author who wrote the book has. This is done by performing a subquery by selecting all the author ids from the author\_followers table, limiting it matching the author id in the table to the author id of the book. The COUNT function then calculates the number of the entries in the table, which is the number of followers. This does not need a join, as the data is fetched through a subquery.  It also gets the number of entires that the book has into any users ‘Have Read’, ‘Want to Read’ or ‘Currently Reading’ lists, as separate rows. This is done by performing a subquery, counting the book id from the reading lists table. It is then joined, using an INNER JOIN to the reading\_list\_names table, so the entries can be selected by list name not id, which avoids needing to know list ids or user ids. The WHERE clause within the subquery specifies which list the number is for, specifying the list name, as either ‘Have Read’, ‘Want to Read’, or ‘Currently Reading’. |

### 

| **Leave a review** |
| --- |
| INSERT INTO reviews (user\_id, book\_id, overall\_rating, plot\_rating, character\_rating, summary, rating\_body) VALUES (<user id>, <book id>, <overall rating>, <plot rating>, <character rating>, <summary>, <rating body>); |
| This query adds a new review from the reviews table. It specifies the user id, book id, overall rating, plot rating, character rating, summary and rating body. The plot rating and character rating, if not given, should be inserted as null. The other fields in the table, such as review\_id and date\_added are set automatically when the record is added. |

### 

##### Diaries

| **Get diary entries for a specific user by id** |
| --- |
| SELECT diary\_entries.entry\_id,  diary\_entries.book\_id,  diary\_entries.overall\_rating,  diary\_entries.character\_rating,  diary\_entries.plot\_rating,  diary\_entries.summary,  diary\_entries.thoughts,  diary\_entries.date\_added,  diary\_entries.pages\_read,  books.cover\_image,  books.title,  authors.author\_id,  authors.first\_name,  authors.surname,  authors.alias,  (SELECT IFNULL(ROUND(AVG(reviews.overall\_rating), 2), 0)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_rating  FROM diary\_entries  INNER JOIN books  ON books.book\_id=diary\_entries.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  WHERE diary\_entries.user\_id=<user id>; |
| This query gets the information for all the diary entries for a specific user. It is limited to the results of a specific user through the WHERE clause, limiting it to a single user id. This therefore ensures that all the results are for the single user. Each row in the result therefore represents a single diary entry.  It gets the entry\_id identifying the diary entry, the book id of the target book, the overall rating, character rating, plot rating, diary entry summary, the user’s thoughts and feelings about what they read, the date it was added and the number of pages read from the diary\_entries table.  It also gets the cover image and title of the book the diary entry is about from the books table. This is done by joining the query to the books table, where the book id of the entry matches the book id from the books table. It also uses an INNER JOIN, as a diary entry cannot exist without a corresponding book due to the foreign key constraints applied to the book id column.  It also gets the author's name information (the first name, surname and writing alias). It also gets the author id as well. It does this by joining the query to the authors table, where the author id from the books table matches the author id from the authors table. This means that the join must happen after the join to the books table, as data from the books table is required to perform the join. It can safely use an INNER JOIN as, due to the foreign key constraints on the author id column in the books table, a book cannot exist without an author, so there isn’t any risk of results being lost.  The query also gets the average rating for the book that the entry refers to. This is done by performing a subquery, selecting the overall rating from the reviews table. It limits the results to the current book through the WHERE clause limiting the book to the same as the row from the outer query. This is averaged using the AVG method, and then rounded to two decimal places using the ROUND method. However, if there has not been any reviews left for that book, the average will be given as null. The IFNULL method ensures that it is 0 if there is not any reviews for that book.  Finally, it gets the number of reviews that have been left for the book. This is done through another subquery, selecting the overall\_rating from the reviews table, and using a WHERE clause to limit the results to the same book as the outer query. It uses the built in COUNT method to count the number of overall ratings, which is synonymous with the number of reviews for the book. If there isn’t any reviews for the book, it will give 0. |

### 

| **Create a diary entry** |
| --- |
| INSERT INTO diary\_entries (user\_id, book\_id, overall\_rating, character\_rating, plot\_rating, summary, thoughts, pages\_read) VALUES (<user id>, <book id>, <overall rating>, <character rating>, <plot rating>, <summary>, <thoughts>, <pages read>); |
| This query creates a new diary entry value. It requires the user id, book id, overall rating, character rating, plot rating, summary, the user's thoughts about the book, and the number of pages read. If the user has not specified the plot rating or character rating, they should be inserted as null. |

### 

##### Genres

| **Get about information** |
| --- |
| SELECT genre\_id,  name,  about FROM genres  WHERE name="<genre name>"; |
| This query gets the genre id, genre name and the description of the genre from the genre table. It is specified using the genre name through the WHERE clause, which limits the result to a maximum of one row, as the genre name needs to be unique. |

### 

| **Get book summary information for all books within a specific genre** |
| --- |
| SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM books  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN book\_genres  ON books.book\_id=book\_genres.book\_id  WHERE book\_genres.genre\_id=<genre id>; |
| This query gets all the information required for a book summary for all of the books that match a specified genre. The genre is specified through the WHERE clause matching the genre id. This means that there will only ever be a maximum of one result from the query, as the genre ids are unique, due to the primary key and auto\_increment constraints.  It selects the data from the books table as that makes the most sense semantically, as it is book data, and would not reduce the number of joins to use another table. It therefore needs to join the query to the book\_genres table, so it can use the genre id within the WHERE clause to limit the results to all the books within that genre. It can use an inner join safely, as if the book matches the genre it must be in the book\_genres table, and if it is not, it can safely be ignored. This will, therefore, not lead to any loss in results. From the books table it gets the title, cover image and id.  The query also gets the author’s name information (first name, surname and writing alias) from the authors table. To be able to do this, the query must be joined to the authors table, matching the author id specified within the book table to the author id in the authors table. It uses an INNER JOIN, as a book cannot exist without a corresponding author, so will not lose any results from this. |

### 

##### Database Searching

| **Get cleaned book titles** |
| --- |
| SELECT books.clean\_title,  books.book\_id,  authors.clean\_name  FROM books  INNER JOIN authors  ON authors.author\_id=books.author\_id; |
| This query gets the clean titles for the books. It also gets the authors name, so that it can be added so searching for an author also gives the results for the books. It gets the clean title and the book id from the books table, and the authors clean name from the authors table. To be able to get data from the authors table, the query is joined to the authors table, matching the author id in the book table with the author id from the authors table. It uses an inner join as a book cannot exist without an author, and therefore results cannot be lost due to the inner join. It gets all of the clean titles, as the results of the query are not limited using a WHERE clause. |

### 

| **Get clean names for genres/authors** |
| --- |
| SELECT clean\_name, genre\_id FROM genres;  SELECT clean\_name, author\_id FROM authors; |
| These queries get all of the clean names for either the authors or the genres. It also gets the identifying id for results, the genre id for genres and author id for authors. The queries give the same data and in the same format, but get it from different tables and column names, which is why they are grouped together. |

### 

| **Delete all stored unique words** |
| --- |
| DELETE FROM unique\_words; |
| This query deletes all the records from the unique\_words table. This removes all the data from it, which could be used if they needed to be recalculated, such as if a new book, author or genre is added to system. If a specific record needs to be deleted from the table, a WHERE clause should be used to match the word id. |

### 

| **Add a new unique words record** |
| --- |
| INSERT INTO unique\_words (word) VALUES <word>; |
| This query adds a unique word record. It only inserts the word value, as it may not be known what the idf values are yet. The additional values for the record, such as the word id and the idf values have default values, so do not need to be inserted into the table. |

### 

| **Change the idf value of a word** |
| --- |
| UPDATE unique\_words  SET idf\_values=<idf value>  WHERE word\_id=<word id>; |
| This query updates the idf value of a word. This is needed as the idf values are not known when the records are first added, or if new unique words are added to the database. It changes the idf\_values column to a specified column for all entries matched by the WHERE clause. The WHERE clause specifies the word by word id, which, because of the primary key and auto\_increment constraints, is unique, and therefore changes only one entry. |

### 

| **Get unique words** |
| --- |
| SELECT word, idf\_values FROM unique\_words; |
| This query gets the word and the corresponding inverse document frequency (idf) value for it. Each row in the result is a new unique word. It gets all the values stored as there is not a WHERE clause to limit results. |

### 

##### Reading lists

| **Get the popular books (most people currently reading)** |
| --- |
| SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias,  COUNT(books.book\_id) as num  FROM books  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_lists  ON reading\_lists.book\_id=books.book\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  GROUP BY books.book\_id  ORDER BY num DESC  LIMIT <number summaries>; |
| This query gets the books that appear the most in ‘Currently Reading’ lists. This is therefore gives the most popular books in the system at the point the query has been performed. It gets the book id, title and cover image from the books table. It gets the data from the books table, as semantically this is most logical as the data is regarding books, mostly from the books table. It does not require any additional joins to select from this table, and therefore should not have a noticeable impact on the query speed. The query also selects the author’s name information, namely their first name, surname and writing alias, from the authors table. To be able to do this, the query must be joined to the authors table, matching the author id from the authors table to the author id in the books table. This can use an inner join, as a book must have an author due to the foreign key constraints, so their will not be any books lost.  It also joins the query to the reading lists table, matching the book id in the reading lists table to the book id in the books table. This uses an inner join, as if the book is not present in the reading lists table, it should not be included in the results. It is required, as it is needed to join to the reading lists table, and therefore required to sort by the list name. Following the join to the reading\_lists table, it is joined to the reading\_list\_names table, so the where clause can use the list name. It joins the reading\_list\_names list id to the list id from the reading\_lists table, using an inner join, as the foreign key constraints prevent the list entry existing without the name. This join must come after the join to the reading\_lists table, as it uses fields from that table within the join statement. This table is used in the where clause, to limit the results of the query to only books that are in reading lists called ‘Currently Reading’. This includes all users, as it is sorting by list name not list id, which makes it independent of the user.  It counts the number of book\_ids that are found in the query. The group by book id statement ensures that there is only one row per book, and that the count is representative of the number of lists that the book is in. It is also ordered by this number, descending, so that the first results are the most popular items. Finally the limit clause limits results to a specific number of books, which will be the books that are being read by the most people. |

### 

| **Get list id from list name and user id** |
| --- |
| SELECT list\_id  FROM reading\_list\_names  WHERE user\_id=<user id>  AND list\_name="<list name>"; |
| This query gets the list id that corresponds to a user id and a list name. It gets the list id from the reading\_list\_names table, and specified through the where clause, matching both user id and list name. Both user id and list name are required as users may have multiple lists and multiple lists may share the same name within the table. |

### 

| **Get currently reading/want to read summaries** |
| --- |
| SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM reading\_lists  INNER JOIN books  ON reading\_lists.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  AND reading\_list\_names.user\_id=<user id>;  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM reading\_lists  INNER JOIN books  ON reading\_lists.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Want to Read"  AND reading\_list\_names.user\_id=<user id>; |
| These queries get the book summary data for all the entries in a specific reading list for a specific user by user id. These queries are for the summaries that the user is currently reading and want to read in that order. These are both here as they are the ones that are most likely going to be used, however changing the value of the list\_name within the where clause would make this applicable to any reading list.  The gets the data from the reading lists table, as semantically the data is regarding reading lists. It does not require any more joins this way, as it would be required regardless for the join to the reading\_list\_names table, so that the where clause can specify the list name. The where clause limits results to a specific reading list and user. This requires the query to be joined to the reading\_list\_names table, matching the list id from the reading list entry to the list id for the reading\_list\_names table. It is done using an inner join as any reading list entry cannot exist without a corresponding list name, so no data will be lost.  The query gets the book id, title and cover image from the books table. To do this it must be joined to the books table, matching the book id in the reading list entry to the book id in the books table, again using an inner join, as the foreign key constraints prevent any data loss because of the join. It also gets the authors name information (their first name, surname and writing alias) from the authors table. This requires the query to be joined to the authors table, matching the author id specified in the books table with the author id in the authors table. This join must happen after the join to the books table, as data from the books table is required to make the join to the authors table. |

### 

| **Get the contents of a list** |
| --- |
| SELECT books.book\_id,  books.cover\_image,  books.title,  books.synopsis,  authors.first\_name,  authors.surname,  authors.alias,  authors.author\_id,  reading\_lists.date\_added,  (SELECT GROUP\_CONCAT(genres.name)  FROM book\_genres  INNER JOIN genres  ON genres.genre\_id=book\_genres.genre\_id  WHERE book\_genres.book\_id=reading\_lists.book\_id  GROUP by book\_genres.book\_id) AS genres,  (SELECT CAST(IFNULL(AVG(reviews.overall\_rating), 0) as FLOAT)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_ratings  FROM reading\_lists  INNER JOIN books  ON books.book\_id=reading\_lists.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_lists.list\_id=<list id>  AND reading\_lists.user\_id=<user id>  ORDER BY reading\_lists.date\_added DESC, books.title ASC; |
| This query gets all the data required for all of the entries in a user’s reading list. The results are ordered by the date the entries were added, however if there are two entries with the same date, they are sorted alphabetically. This is done using the order by clause at the end of the query.  It gets the date added from the reading lists table. This is taken initially from the reading lists table, as semantically the information retrieved is about the reading lists. If it were taken from the books table, it would still require the same number of queries, so performance is likely to be the same.  The query gets the book id, cover image, title and the book’s blurb from the books table. To perform this, the query needs to be joined to the books table, matching the book id in the entry to the book id in the books table. It uses an inner join as it is not possible for a list entry to exist without the corresponding book in the books table, due to the foreign key constraints on the column. The query also gets the author’s first name, surname, alias and author id from the authors table. To do this, it must also be joined to the authors table, matching the author id specified in the books table with the author id in the authors table. It can use a inner join as the foreign key constraints prevent a book or author without a corresponding author or book. This join must happen after the join to the books table, as the author id column from the books table is needed to perform the query.  It also gets a list of all the genres that the book applies to. It gives the result as a comma separated list of names, as this prevents the need for an additional query to get this data, which would likely be slower. To do this, it performs a subquery that selects data from the book\_genres table, and joins it to the genres table so the genre names can be accessed. It matches the genre id from the book genres table to the genre id from the genres table. This name column from this table then has the group\_concat function applied to it to make it into a comma separated list of the names. The group by clause within the subquery ensures that the group\_concat only joins the results for the one book, rather than all of the books. Finally, the where clause ensures that the results are for the same book as in the outer query.  The query also gets the average rating for the book that the entry refers to, through a subquery, selecting the overall rating from the reviews table. It limits the results to the current book through the where clause limiting the book to the same as the row from the outer query. This is averaged using the AVG method, and if there has not been any reviews left for that book, the average will be given as null. The IFNULL method ensures that it is 0 if there are not any reviews for that book. The result of this is then cast to a float.  Finally, it gets the number of reviews that have been left for the book. This is done through another subquery, selecting the overall\_rating from the reviews table, and using a WHERE clause to limit the results to the same book as the outer query. It uses the count method to count the number of overall ratings. |

### 

| **Delete a reading list entry** |
| --- |
| DELETE FROM reading\_lists  WHERE user\_id=<user id>  AND book\_id=<book id>  AND list\_id=<list id>; |
| This query deletes a single entry from the reading lists table. It takes the user id, list id as it might make it more difficult to delete an entry if it was not done by the correct user. It specifies the list id, as each user could have the same book appearing in multiple reading lists. The book id can then be used to specify a single entry in a specific list, as a list can only have a book appearing once in it. |

### 

| **Create a reading list entry** |
| --- |
| INSERT INTO reading\_lists (user\_id, book\_id, list\_id) VALUES (<user id>, <book id>, <list id>); |
| This query creates a new reading list entry. It specifies the user id of which the entry belongs to, the book id of the book to be added in the new entry, and the list id of the list that the entry is to be added into. The other fields are filled automatically on insertion, such as the entry\_id which is filled because of the auto\_increment on the column, and the date\_added which defaults to the date and time at which the query was performed. |

### 

| **Delete a reading list** |
| --- |
| DELETE FROM reading\_lists  WHERE list\_id=<list id>  AND user\_id=<user id>;  DELETE FROM reading\_list\_names  WHERE list\_id=<list id>  AND user\_id=<user id>; |
| These queries delete a reading list. This requires two queries, as the values in the reading list need to be deleted, then the list itself needs to be deleted, and as they are in different tables, they need to be deleted within separate queries.  The first query deletes the entries in the specific list, which is specified by the list id and user id within the where clause. The second deletes the actual list name from the reading\_list\_names table, which is the same as deleting the list. The foreign key constraints mean that the contents of the list need to be deleted first, and the constraints cannot be safely removed for the operation, as the same user could be accessing the database at the same time, which would then apply to the user as well, so could then have unintended impacts of deleting items that should not have been possible to delete. |

### 

| **Create a new reading list** |
| --- |
| INSERT INTO reading\_list\_names (user\_id, list\_name) VALUES (<user id>, "<list name>"); |
| This query creates a new reading list, by adding a new record to the reading\_list\_names table. It specifies the user id of the user the new list belongs to, and the name of the new list. The list\_id is automatically filled because of the auto\_increment on the column |

### 

| **Get most recent book read** |
| --- |
| SELECT books.book\_id,  Books.title  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  INNER JOIN books  ON books.book\_id=reading\_lists.book\_id  WHERE reading\_lists.user\_id=<user id>  AND reading\_list\_names.list\_name="Have Read"  ORDER BY reading\_lists.date\_added DESC  LIMIT 1; |
| This query gets the most recent book a user has read. The query is selected from the reading lists table, as it is regarding the reading lists, although retrieves data from the book table. The query needs columns from the reading\_lists table, so selecting from this table over the books table does not have any performance implications.  The query is joined to the books table, matching the book id of the list entry to the book id from the books table. It is done using an inner join, as the foreign key constraint linking the two columns together, ensures that there cannot be a reading list entry without the corresponding book, so the inner join cannot lose any results from the query. This then allows the book id and title to be retrieved from the books table.  It is also joined to the reading\_list\_names table, matching the list id of the list entry to the list id of the reading list name. This again can use an inner join as the list entries cannot exist, so it cannot lead to data loss. This is then used within the where clause to limit the results to results in a list called ‘Have Read’. It also limits it by user id, which would be the same as limiting by list id, but this approach does not require a separate query to find the list id.  Finally, the results are ordered by the date added, with the most recent first. Finally, it is limited to one result, which is therefore the most recent book that has been read. |

### 

| **Get newest list entry that has not been read** |
| --- |
| SELECT books.book\_id,  books.title  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  INNER JOIN books  ON books.book\_id=reading\_lists.book\_id  WHERE reading\_lists.user\_id=<user id>  AND reading\_list\_names.list\_name!="Have Read"  ORDER BY reading\_lists.date\_added DESC  LIMIT 1; |
| This query works the same as the previous one. However, in the where clause it checks that the reading list name is not ‘Have Read’, which means that the query will get the most recent addition to a reading list which was not added to the ‘Have Read’ list. This means it gets the most recent book added into any of a user’s reading lists, which has not been read. |

### 

##### Recommendations

| **Get users without known preferences** |
| --- |
| SELECT user\_id FROM users WHERE preferences\_set=FALSE; |
| This query gets the user id from the users table, which have not expressed any explicit data to generate recommendations. This is done by limiting the results to users where there preferences\_set is false, through the where clause |

### 

| **Get review ratings average for all of a user’s reviews** |
| --- |
| SELECT book\_id,  (overall\_rating + IFNULL(character\_rating, overall\_rating) +  IFNULL(plot\_rating, overall\_rating)) / 3  FROM reviews  WHERE user\_id=<user id>  GROUP BY review\_id; |
| This query gets all the book ids and the average rating for it for all of a users ratings. It gets the book id from the reviews table. It also sums the overall\_rating, character\_rating and plot\_rating together. If the character rating or plot rating is not set, it is set to the overall rating, so that the average is not effected by not being given. It is then divided by three to get the average of the three ratings for each review. The group by clause is needed to ensure that the three columns are only for a specific review. Finally, the results are limited to reviews from a specific user through the where clause matching user id. |

### 

| **Get books from authors in user’s initial preferences** |
| --- |
| SELECT books.book\_id  FROM initial\_preferences  INNER JOIN books  ON books.author\_id=initial\_preferences.author\_id  WHERE initial\_preferences.user\_id=<user id>  GROUP BY books.book\_id; |
| This query gets all of the book ids written by the authors specified in a user’s initial preferences. It selects first from the initial\_preferences table, which is then inner joined to the books table, matching the author id in the preferences table to the author id in the books table. The group by clause is required to ensure that each book is in a single row, not one row per author. It is limited to results for a single user by matching the user id in the where clause. This should only give a results for a single user, as user ids are unique because of the primary key constraint on the user id column in the users table. |

### 

| **Get book ids from authors that a user is following** |
| --- |
| SELECT books.book\_id  FROM author\_followers  INNER JOIN books  ON books.author\_id=author\_followers.author\_id  WHERE user\_id=<user id>; |
| This query gets all of the book ids that have been written by the authors that a user follows. It joins the query to the books table, matching the author id in the author\_followers table, which the query is initially selected from, to the author id from the book table. The inner join means that book ids for all books written by all the authors within the table are returned, one row for each book. The where clause limits all the results to a single user, by user id. The user id is unique, due to the primary key constraint on the user id column in the users table. |

### 

| **Get average rating from diary entries for each book** |
| --- |
| SELECT book\_id,  (SUM(overall\_rating) + SUM(IFNULL(character\_rating, overall\_rating)) +  SUM(IFNULL(plot\_rating, overall\_rating))) / (COUNT(entry\_id) \* 3)  FROM diary\_entries  WHERE user\_id=<user id>  GROUP BY book\_id; |
| This query gets the average rating for the diary entry for all of the books the user has created diary entries for. If the character rating or plot rating is null, so has not been specified, the overall rating is used, so that the average is not affected by them not being present. These are then summed for all of the diary entries of that book, and divided by the number of entries, multiplied by three, to get the average rating. The group by statement ensures that the sum and count is only for entries with the specific book id. Finally, the where clause limits results to a single user, which is specified by id. Each row in the resulting query should be for a single book that an entry has been left for, only for a single user. |

### 

| **Get book ids of books in system-defined reading lists** |
| --- |
| SELECT reading\_lists.book\_id  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_lists.user\_id=<user id>  AND reading\_list\_names.list\_name IN (  "Currently Reading",  "Have Read",  "Want To Read"  ); |
| This query gets all of the book ids from all of the entries a user has made into any of the system-created lists, namely the ‘Have Read’, ‘Currently Reading’, and ‘Want to Read’. It selects the book ids from the reading\_lists table. To be able to limit by the list name, it must be joined to the reading\_list\_names table, matching the list id in each of the tables together. The where clause limits results to a specific user by matching the user id, and to the system-defined lists by ensuring that the list name is in the correct form. It cannot use list ids for this, as each user would have separate list ids, which would require an additional query beforehand, which would likely be slower than the join. |

### 

| **Delete old recommendations** |
| --- |
| DELETE FROM recommendations  WHERE date\_added<=DATE\_SUB(NOW(), INTERVAL 2 DAY); |
| This query deletes all the old recommendations from the recommendations table, for all of the users. It is limited to old results using the where clause. It checks whether the date the recommendation was made was more than two days ago, and if it was it is deleted. This is done by subtracting two days from the current time, using the DATE\_SUB method, and then matching all of the dates to this. Any values before this should be removed from the table. |

### 

| **Delete a specific recommendation** |
| --- |
| DELETE FROM recommendations  WHERE user\_id=<user id>  AND book\_id=<book id>; |
| This deletes a specific recommendation. Each user can only have a book recommended to them once, so this therefore will only specify a single entry. This will then only delete a single item. |

### 

| **Get user’s recommendations** |
| --- |
| SELECT recommendations.book\_id,  ROUND(recommendations.certainty \* 100, 1) as certainty,  recommendations.date\_added,  books.cover\_image,  books.synopsis,  books.title,  authors.first\_name,  authors.surname,  authors.alias,  authors.author\_id,  (SELECT GROUP\_CONCAT(genres.name) FROM book\_genres  INNER JOIN genres  ON book\_genres.genre\_id=genres.genre\_id  WHERE book\_genres.book\_id=recommendations.book\_id  GROUP BY books.book\_id) AS genres,  (SELECT ROUND(CAST(IFNULL(AVG(reviews.overall\_rating), 0) as FLOAT), 2)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_ratings  FROM recommendations  INNER JOIN books  ON recommendations.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  WHERE recommendations.user\_id=<user id>  ORDER BY recommendations.certainty DESC; |
| This query gets all of a users recommendations. It gets the book id and the date the recommendation was generated from the recommendations table. It also gets the certainty of the recommendations. It multiplies the certainty by 100, to convert it to a percentage, and rounds it to one decimal place.  It also gets the book information, namely the cover image, title and synopsis. To do this, the query is joined to the the books table, matching the book id of the recommendation to the book id in the books table. This uses an inner join, as the foreign key constraint on the book id column within the recommendations table prevents a recommendation from being generated which is not in the books table. In addition to this, it also gets the author’s name from the authors table, including their first name, surname and writing alias, as well as the author id. This also requires the query to be joined to the authors table, matching the author id from the authors table, to the author id specified within the books table. It is done using an inner join, as the foreign key constraints on the author id column in the books table prevents a book having an author that is not in the authors table. The join to the authors table must happen after the join to the books table, as it requires a column from the books table to perform the join.  Finally the query gets all the genres that are applicable to that book, the average rating, and the number of ratings. This is done using subqueries, in the same way as for getting the entries into a reading list.  The results are limited to a single user, by specifying the user id through the where clause. This limits the results to a single user, with each record in the result being a separate recommendation. The results are also ordered by certainty, with the closest book being the first result. |

### 

| **Mark a user’s preferences as set** |
| --- |
| UPDATE users  SET preferences\_set=TRUE  WHERE user\_id=<user id>; |
| This changes the flag that specifies whether a user has left initial data for recommendations. It changes the preferences\_set column to true, which would be used for when a user has specified their favourite authors before recommendations are generated for them. It is limited to a single user, by limiting to a single user, using the where clause. This will only change a single user, as the user id column is unique, due to the primary key constraint, and the auto\_increment applied to the column. |

### File system

| .  ├── ./assets  │ ├── ./assets/favicon.ico  │ ├── ./assets/logo.png  │ └── ./assets/book\_covers  │ ├── ./assets/book\_covers/book1.png  │ ├── ./assets/book\_covers/book2.png  │ └── ./assets/book\_covers/book3.png  ├── ./backend  │ ├── ./backend/wsgi.py  │ ├── ./backend/maintenance.py  │ └── ./backend/configuration.py  ├── ./css  │ └── ./css/styles.css  ├── ./html  │ ├── ./html/author.html  │ ├── ./html/book.html  │ ├── ./html/browse.html  │ ├── ./html/diary.html  │ ├── ./html/genre.html  │ ├── ./html/home.html  │ ├── ./html/my\_books.html  │ ├── ./html/recommendations.html  │ └── ./html/search.html  ├── ./javascript  │ ├── ./javascript/jquery.js  │ └── ./javascript/script.js  ├── ./default\_config.json  ├── ./index.html  └── ./project\_config.conf |
| --- |

This outlines the expected layout for the file system of the project. This is likely to change, with additional files as the the code will likely be modularised, and additional files added for simplicity. However, the basic file structure is unlikely to change from this. The full file paths have been given, relative to the root folder, as these are what would be used to access the file from the HTML. All the folders also use lowercase, and underscores over spaces, so they are easier to access throught the HTML, as well as through the command line.

The assets folder contains any images that are required by the project. The favicon.ico file is the icon to be displayed in the tab, and the logo.png file is the logo to be displayed at the top of the webpage. The book\_covers folder stores any locally stored book cover images. The path to that image would then be stored in the cover\_image column in the books table, within the database, so it can be accessed in a separate get request.

The backend folder contains any of the scripts required for the project, including the WSGI file, the maintenance file, and the custom configuration module. These have been put into a separate folder, as this is likely easier to prevent users accessing through the Lighttpd configuration.

The css folder is contains any css files. This is likely to only be one file in this folder, as the other css that is likely to be the font-awsome stylesheet. However, there is little benefit outside of faster loading times, so is not essential for the project, given that there is also links available to be used instead.

The html folder stores each of the individual html files for the pages. These are then inserted into the index.html file as needed, which should improve appered responsiveness of the site, as the page does not need to be refreshed to switch between them. These also, therefore, do not need to be complete files, as they do not need to have the javascript or the stylesheets added.

The javascript folder contains all of the javascript files used. This includes any additional modules used, including JQuery. JQuery should be stored locally, so it can be requested faster, and therefore reduce access times.

The index.html file needs to be outside of the html folder, as the webserver will look for it in the root folder. This could be configured differently, however it is convention for it to be in the root folder. The configuration options are also in the root folder, as this is logical.

### Handlers and Middleware

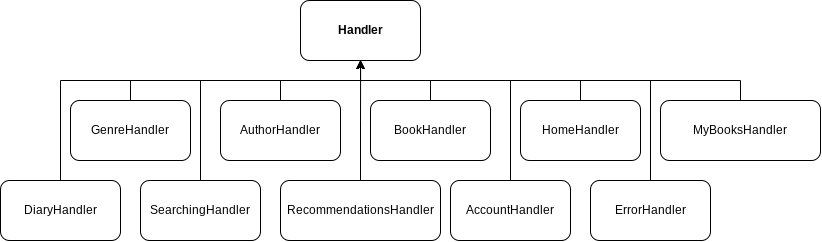
All the requests made to the WSGI script can be catagorised into:

* Reading lists
* Genres
* Books
* Authors
* Home
* Recommendations
* Search
* Accounts
* Diaries
* Errors

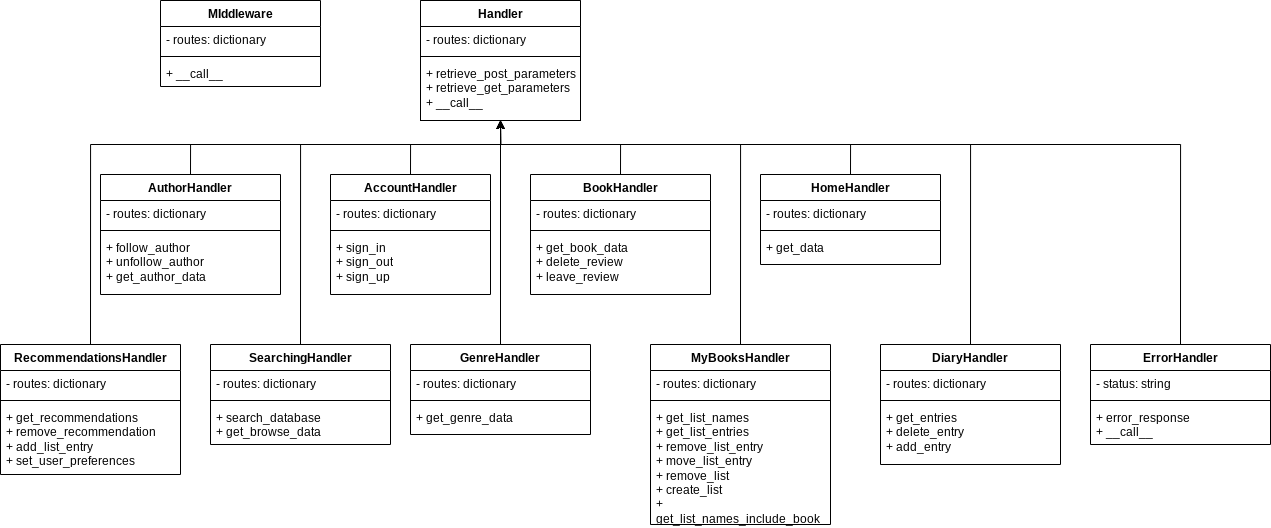
Breaking down the project into these individual parts should allow each section to be developed in isolation, making development simpler. It should also make development easier, as well as tracking requests as they are handled, which should further aid debugging. Furthermore, it offers a simple way to compartmentalise requests, and make creating request URLs easier. Each of the handlers then can handle a single type of URLs. Forming the URLs out of ’/cgi-bin/handler-name/process’. The ‘cgi-bin’ means that the request is passed to the WSGI script. The ‘handler-name’ then specifies which handler should be used, and then the process specifies which function, within the handler that should be used. This then means that there should be nine handlers (with an additional one for errors).

| **Handler Name** | **Purpose** | **Example URL** |
| --- | --- | --- |
| My Books | Handling any reading list processes | /cgi-bin/my\_books/create\_list |
| Genres | Handling any genre requests | /cgi-bin/genres/get\_about |
| Books | Handling any book requests, including reviews | /cgi-bin/books/get\_about |
| Authors | Handling any author’s related requests | /cgi-bin/authors/get\_about |
| Home | Handling any requests relating to home screen data | /cgi-bin/home/get\_data |
| Recommendations | Handling any recommendations requests | /cgi-bin/recommendations/get\_recommendations |
| Search | Handling any browsing or searching requests | /cgi-bin/search/get\_browse\_data |
| Accounts | Handling any account-related requests | /cgi-bin/accounts/sign\_out |
| Diaries | Handling diary related requests | /cgi-bin/diary/get\_entries |
|  |  |  |
| Errors | Errors that have occurred during a request, such as 404 | /cgi-bin/abc/asd |

These should each be classes, with their own methods, all inherited from the Handler superclass. The following inheritance diagram shows the relationship between them all.



This second diagram shows the inheritance relationship between all of the classes, and the methods associated with each of them. The \_\_call\_\_ method is the method that would be invoked if the object was called like a function, such as ‘objectName()’.



The purpose of the middleware is to direct the requests to the correct handler, based on the URL, after it has been passed to the WSGI file. This can then also log all the requests that pass through it, and the handler that is used to handle it. The following table shows the mapping between the request URL, excluding the ‘cgi-bin’ prefix, as it must be present to be passed to the middleware. It also includes all of the request processes each handler performs, and the corresponding function to be executed within the handler

| **Handler** | **URL handler name** | **URL process name** | **Executed Method** |
| --- | --- | --- | --- |
| AccountsHandler | …/account | /sign\_in | sign\_in |
| /sign\_out | sign\_out |
| /sign\_up | sign\_up |
| MyBooksHandler | …/my\_books | /get\_lists | get\_list\_names |
| /get\_list\_entries | get\_list\_entries |
| /remove\_list\_entry | remove\_list\_entry |
| /add\_list\_entry | add\_list\_entry |
| /move\_list\_entry | move\_list\_entry |
| /remove\_list | remove\_list |
| /create\_list | create\_list |
| /get\_lists\_book\_target | get\_list\_names\_include\_book |
| GenreHandler | …/genres | /about\_data | get\_genre\_data |
| BookHandler | …/books | /about\_data | get\_book\_data |
| /delete\_review | delete\_review |
| /add\_review | leave\_review |
| DiaryHandler | …/diary | /get\_entries | get\_entries |
| /delete\_entry | delete\_entry |
| /add\_entry | add\_entry |
| HomeHandler | …/home | /get\_data | get \_data |
| RecommendationsHandler | …/recommendations | /get\_recommendations | get\_user\_recommendations |
| /remove\_recommendation | remove\_user\_recommendation |
| /add\_list\_entry | add\_to\_want\_read |
| /set\_user\_preferences | set\_new\_user\_preferences |
| SearchingHandler | …/search | /search | search\_database |
| /get\_browse\_data | get\_browse\_data |

### Project Maintenance

For the project to work correctly, the database needs to be maintained, to keep the number of records in some of the tables down to a manageable size. This includes closing any expired sessions, removing old recommendations, whilst also generating new ones. Removing expired sessions prevents the table from growing in size indefinitely, which should reduce the amount of required storage space, as well as increasing the query speed. When new recommendations are generated, old ones should be removed, so each user only ever has a fixed number of recommendations. This prevents the table from growing over time, as well as allowing recommendations to remain accurate to the user. The following pseudocode can be used to maintain the project.

| ARRAY sessions = sessions.GET\_SESSIONS()  FOREACH session IN sessions:  IF session.IS\_EXPIRED() THEN:  session.CLOSE()  ENDIF  ENDFOR  recommendations.FIT()  recommendations.DELETE\_OLD()  recommendations.GENERATE\_RECOMMENDATIONS() |
| --- |

This first gets a list of all of the active sessions. It then iterates through them all, checking whether they have expired or not. If they have expired, they are then closed, and therefore removed from the table. This ensures that only the necessary sessions, which have recently been used, remain active. Following this, it re-fits the recommendations model, so that it accounts for any new user activities. It deletes old recommendations, so the table does not keep growing, and then generates the new recommendations for all users. This process would need to be carried out frequently, such as every day, at a point of low usage, as this would likely be a computationally expensive process. This could be achieved by setting up a cron-job, which runs the maintenance script every day at 1am, using the command below. This assumes that the script is under the backend folder, and is called maintenance.py, and is in python. However, the part that specifies the cronjob would be the same, regardless of the required command to run the script.

| 0 1 \* \* \* python3 /absolute/path/to/project/backend/maintenance.py |
| --- |

### Lighttpd Configuration

Lighttpd needs to be configured to allow for the dynamic content loading, redirecting all the URLs to the index.html file, and logging the website activity.

| server.modules = (  "mod\_indexfile",  "mod\_access",  "mod\_alias",  "mod\_redirect",  )  server.document-root = "</absolute/path/to/project>"  server.upload-dirs = ( "/var/cache/lighttpd/uploads" )  server.errorlog = "/var/log/lighttpd/error.log"  server.pid-file = "/run/lighttpd.pid"  server.username = "www-data"  server.groupname = "www-data"  server.port = <port>  #https://redmine.lighttpd.net/projects/lighttpd/wiki/Server\_feature-flagsDetails  server.feature-flags += ("server.h2proto" => "enable")  server.feature-flags += ("server.h2c" => "enable")  server.feature-flags += ("server.graceful-shutdown-timeout" => 5)  server.http-parseopts = (  "header-strict" => "enable",  "host-strict" => "enable",  "host-normalize" => "enable",  "url-normalize-unreserved"=> "enable",  "url-normalize-required" => "enable",  "url-ctrls-reject" => "enable",  "url-path-2f-decode" => "enable",  "url-path-dotseg-remove" => "enable",  )  index-file.names = ( "index.html" )  url.access-deny = ( "~", ".inc" )  static-file.exclude-extensions = ( ".php", ".pl", ".fcgi", ".py" )  include\_shell "/usr/share/lighttpd/use-ipv6.pl " + server.port  include\_shell "/usr/share/lighttpd/create-mime.conf.pl"  include "/etc/lighttpd/conf-enabled/\*.conf"  server.modules += (  "mod\_dirlisting",  "mod\_staticfile",  ) |
| --- |

### 

This configuration is the main lighttpd configuration file. It first includes all of the additional modules and modifications that are required for the project to work. It then sets the location of the error log, upload directory, the PID file, the username and the groupname. It also sets the port for the webserver to use, and the document root for the project. It also adds support for HTTP/2 and cleartext support, and sets the number of seconds to wait for existing requests to complete.

It limits the characters allowed in HTTP request headers, Host header, and normallieses the HTTP host header. It also decodes reserved unreserved characters if percent encoded. It also rejects any control characters that are encoded using percents, translates ‘%2F’ to a forward slash, and reject ‘.’ and ‘..’ path segments.

It sets the index file to index.html, which is the file that should be loaded as the home page for the site. It also denies access to any files ending in ‘.inc’, which is often used for code and ‘~’ for any backup files. The exclude-extensions ensures that the file extensions are not required for the URL’s to still be able to get the correct file.

This configuration is based upon the default configuration that comes when lighttpd is installed, and the rest is based upon the recommendations given within the documentation for (Lighttpd, n.d.-c), and their example configurations.

#### Enabled modules

| server.modules += ( "mod\_rewrite" )  url.rewrite-once += (  "^/recommendations(\?[^?]\*)?$" => "/index.html$1",  "^/diary(\?[^?]\*)?$" => "/index.html$1",  "^/my-books(\?[^?]\*)?$" => "/index.html$1",  "^/browse(\?[^?]\*)?$" => "/index.html$1",  "^/genre/[\w-]+(\?[^?]\*)?$" => "/index.html$1",  "^/book/.+(\?[^?]\*)?$" => "/index.html$1",  "^/author/.+(\?[^?]\*)?$" => "/index.html$1",  "^/search(\?[^?]\*)?$" => "/index.html$1"  ) |
| --- |

This configuration file should be included in the conf-enabled folder. It first adds the rewrite module to the used modules. It then maps any url which is used within the project back to the home page, so content can be loaded dynamically, without refreshing the page so it seems more seamless, and also allow for the URL to be changed with the javascript, so it is clearer as to what page the user is looking at. The regular expressions allow for the URL to be updated with any information specific to the current page, such as search query, genre name, book id, or author id. This is explained by (Lighttpd, n.d.-b).

| server.modules += ( "mod\_accesslog" )  accesslog.filename = "/var/log/lighttpd/access.log" |
| --- |

This adds the access log module to the project, so any requests made to the server are logged, and written to the file. This should make it easier to debug and use the project.

| server.modules += ( "mod\_fastcgi" )  fastcgi.debug = 1  fastcgi.server = ("/wsgi.py" =>  ((  "socket" => "/tmp/fsgi.sock",  "bin-path" => "/path/to/project/backend/NEA/flup.server.fcgi",  "check-local" => "disable",  "max-procs" => 1  ))  )  url.rewrite-once += (  "^/cgi-bin/(.\*)$" => "/wsgi.py$1"  ) |
| --- |

This adds the fastcgi module to the project. This passes any url starting with ‘/cgi-bin’ to the FastCGI script, in this case the WSGI file. The debug option is enabled, and should be changed to a 0 when it is deployed. This is based upon the documentation given by (Lighttpd, n.d.-a).

### Class diagrams and definitions

This section includes all of the class definitions, and the relationships between all of the classes. Not all of the methods that are in these definitions may not be explained in this section, as they are making a database query, and packaging the results into a JSON. Some classes are likely to just be these methods, so will not be explained. It will still be included within these diagrams however.

#### Class diagrams

#### Class Definitions

Accounts

* Hashing algorithm
* Hashing\_salt
* Number hash passes
* Database connection
* reading\_lists
* Hash\_password
* Check credentials
* Create user
* Get user id
* Get user id list

Sessions

* Database connection
* Token size
* Create session
* Update time
* Get uder id
* Close
* Get session id list

Authors

* Number summaries home
* Number genres
* Database connection
* Follow
* Unfollow
* Get number followers
* Get about data
* Id to name
* Get author id list
* Get author favourite data

### Session IDs

The session IDs that are used to identify users should be unique. Having duplicate session IDs would make it impossible to know which user they correspond to. Generating random strings is relatively expensive, and for a large user base, the length would have to be very long. The required length of this could be reduced by appending a second field that frequently changes. This could be done by generating a secure random string, and then appending the current time to it, which would mean that the number of sessions available from the string resets every second.

A random string of length n, would have 26n different possible permutations. Appending this to the current time, means that it can have 26n users sign in every second. Extending this to include uppercase and lowercase letters, and numbers this becomes 62n. However, increasing the string length, the cost of generating the strings will increase too, so should be the lowest it can reasonably be. This means that it should be configurable, so it can vary with the size of the user-base. Below is an example session ID using these rules, generating a 10 character long random string, and appending the time in seconds since the epoch. This would therefore give a possible possible permutations, and therefore could have that many new sessions created every second.

| zWIwXcSqy7N7eg | 1700738771 |
| --- | --- |
| 10 Character long random string | Number of seconds since the epoch |

### Books

The main aspect of the project for the books is calculating books that are similar to it. The other associated functions are database queries, and packaging the results into dictionaries to be sent to the client. These are not listed here, as they were given within the AJAX section.

First, the genres and the corresponding weighting, and the author id for all of the books needs to be retrieved from the database. This can be done using the previously explained query, which is also shown below.

| SELECT book\_genres.book\_id,  GROUP\_CONCAT(book\_genres.genre\_id) as genres,  books.author\_id  FROM book\_genres  INNER JOIN books  ON books.book\_id=book\_genres.book\_id  GROUP BY book\_id; |
| --- |

This pseudocode can be used to find the most similar books to a specified book. The variable ‘results’ is a two dimensional array, with an array for each result, and each of the 3 items in the inner array corresponds to each column in the query result. The number\_similarities variable is the number of similar books that the algorithm is meant to find. This should be configurable, as the more it needs to find, the more data that is required to be sent to the client, and therefore the higher the latency. It also is likely to be more computationally expensive to get more results.

| DICTIONARY genre\_dict = {}  FOREACH book IN results:  SET items = {}  FOREACH genre IN SPLIT( ",", book[1] ):  items.ADD( FLOAT( genre ) )  ENDFOR  items.ADD( -book[2] )  genre\_dict[i[0]] = items  target\_genres = genre\_dict[book\_id]  genre\_dict.POP( book\_id )  ARRAY results = []  FOREACH book IN genre\_dict:  DICTIONARY value\_dict = {  "book\_id": book,  "distance": 1 - JACCARD\_SIMILARITY( target\_genres, genre\_dict[book] )  }  results.APPEND( value\_dict )  ENDFOR  output = SORT( results BY "distance" DESCENDING ).FIRST( number\_similarities ) |
| --- |

The pseudocode creates an empty dictionary to store the sets of all the factors that are being considered in calculating the similarity between the books. For each of the books in the results (results is an array containing the results of the previous MySQL query), it converts the comma-separated string of values to floats and appends them to the set. It also adds the negative of the author id, to the set. This can be done, as the jaccard similarity uses the union of sets, and they will be counted as separate, so this allows the author id to be factored in. It removes the book id of the book that the algorithm is finding similarities with, and sets it as the target. Then it iterates through all of the books in the genre dictionary, assigning a book id and distance to a dictionary, and adding it to the results array. Finally, the results array is sorted, by the jaccard distance, descending. These are the most similar books, and taking the first n of these, the n closest matches are found.

### Information Retrieval

The information retrieval aspect of the project is responsible for the search feature. It should be able to take a search string entered by the user, and return the best matches in the entire database. This should encompass books, authors and genres. Furthermore, searching for a book should give the author, and searching for an author should also give books that they have written. This could be achieved by adding the author’s name to the end of the clean\_title stored for the book, as this would then mean it is treated as part of the title, and would therefore appear in search results. However, this also means that there is a chance a search for an author will give book results above authors. This therefore means that there should be a hierarchy for the order of results, with authors coming first, followed by books, then genres last.

#### Getting clean document names

Document titles should be cleaned before being used, so that punctuation and capitalisation does not matter. This should therefore be applied to all of the searchable items in the project. This process is not expensive, however it is likely to be significantly quicker to store these, then query them from the database, which is why the clean\_name/clean\_title columns are used in the books, authors and genres tables for this reason.

The cleaning process should remove any punctuation and make all characters lowercase. The following pseudocode can be used to clean a string. The result of this can then be stored when the item is. Applying this to all of the searchable terms in the database will then give all of the clean terms to use for the TF-IDF search.

| FUNCTION clean\_string ( target\_string ):  target\_string = target\_string.LOWER()  STRING results\_string = “”  FOREACH character IN target\_string:  IF character.IS\_ALPHA\_NUMERIC() THEN:  results\_string = results\_string + character  ELSE:  results\_string = results\_string + “ ”  ENDIF  ENDFOR  RETURN results\_string  ENDFUNCTION |
| --- |

This pseudocode first converts the inputted string to lowercase, to remove any capital letters. It then also creates an empty string for the result. It then iterates through all of the characters within the string, and if they are letters or numbers, they are added to the result string, otherwise a space is added. This therefore converts the input string into lowercase and removes any punctuation.

#### Loading documents

TF-IDF needs to have a list of all the documents in the database. This is made up of all the cleaned names, or titles for all of the genres, books and authors stored in the database. These clean items should be then added to an array of clean items that the search is finding the best match for a search query from. These should be kept in memory, as this prevents it needing to be loaded every time, which should then increase the speed of using the search feature, as less processing is required for it.

The following pseudocode can generate the dictionary of all cleaned titles to be used by the implementation. The documents\_dictionary array contains a dictionary that contains a string linking the cleaned name, id of the entry, and a character that explains whether it is a book, genre or an author, which are given in the table below. The documents array stores all of the words, without any of the associated data with them. Having the two arrays is quicker, as traversing and accessing the words from the dictionary will be slower, as there would need to be a dictionary lookup for each item as well. The array does not have this limitation. The subsequent pseudocode uses previously explained queries to create the dictionary and the array. The 3 different queries have been duplicated below, and assigned to result\_books, result\_genres, and result\_authors, each being a 2D array, with an array per result from the query, and the subarray having an index per column.

| **Type** | **Identifier** |
| --- | --- |
| Author | a |
| Book | b |
| Genre | g |

| **result\_books** |
| --- |
| SELECT books.clean\_title,  books.book\_id,  authors.clean\_name  FROM books  INNER JOIN authors  ON authors.author\_id=books.author\_id; |

| **result\_genres** |
| --- |
| SELECT clean\_name,  genre\_id  FROM genres; |

| **result\_authors** |
| --- |
| SELECT clean\_name,  author\_id  FROM authors; |

| ARRAY documents\_dictionary = []  ARRAY documents = []  STRING new\_title = “”  FOREACH record IN result\_books:  new\_title = record[0] + “ “ + record[2]  DICTIONARY dictionary = {  “type”: “b”,  “words”: new\_title,  “id”: record[1],  “similarity”: 0  }  documents\_dictionary.APPEND( dictionary )  documents.APPEND( new\_title )  ENDFOR  FOREACH record IN result\_genres:  DICTIONARY dictionary = {  “type”: “g”,  “words”: record[0],  “id”: record[1],  “similarity”: 0  }  documents\_dictionary.APPEND( dictionary )  documents.APPEND( record[0] )  ENDFOR  FOREACH record IN result\_authors:  DICTIONARY dictionary = {  “type”: “a”,  “words”: record[0],  “id”: record[1],  “similarity”: 0  }  documents\_dictionary.APPEND( dictionary )  documents.APPEND( record[0] )  ENDFOR |
| --- |

This pseudocode constructs both the documents\_dictionary and documents, using the results of the MySQL queries. The type attribute in each dictionary in the documents\_dictionary contains a single letter, as given in the previous table. The words attribute contains all of the words in the clean title or name of the document. The ID contains the ID used within the database for the record, which can then be narrowed down to a table using the type field. Finally, the similarity is set to 0, so it is marked as a bad match initially. When adding the books, it appends the author’s name to the clean title, so that when the database is searched using an author’s name it will also give the books that the author has written.

#### Generating TF values

The term frequency (TF) is the number of occurrences of a specific term, divided by the number of terms, as explained in the analysis section. The equation for this is below, where is the number of occurrences, and is the total number of terms.

This can be rewritten as:

Re-writing the formula this way makes it simpler to calculate these terms, as it can then be done using a for loop, adding the reciprocal of the number of terms each time. The following pseudocode performs this process, to calculate the term frequency (TF) values. For the pseudocode, both the documents\_dictionary and documents need to have been loaded into memory.

| FUNCTION generate\_term\_frequency\_values (term DEFAULT NULL):  IF term == NULL THEN:  INTEGER count = 0  FOREACH document IN documents\_dict:  DICTIONARY tf = {}  ARRAY words = document[“words”].split( “ “ )  FLOAT reciprocal = LENGTH( words ) \*\* -1  FOREACH w IN words:  IF w != “ “ THEN:  IF tf.CONTAINS( w ) THEN:  tf[w] = tf[w] + reciprocal  ELSE:  tf[w] = reciprocal  ENDIF  ENDIF  ENDFOR  documents\_dict[count][“tf”] = tf  count = count + 1  ENDFOR  RETURN NULL  ELSE:  DICTIONARY tf = {}  ARRAY words = term.split( “ “ )  FLOAT reciprocal = LENGTH( words ) \*\* -1  FOREACH w IN words:  IF w != ““ THEN:  IF tf.CONTAINS( w ) THEN:  tf[w] = tf[w] + reciprocal  ELSE:  tf[w] = reciprocal  ENDIF  ENDIF  ENDFOR  RETURN tf  ENDIF  ENDFUNCTION |
| --- |

This function calculates the term frequency. It takes an optional parameter, term. It defaults to null, if it is not specified, as this makes it simple to check whether it has been given. If it is not given, it calculates the term frequency values for all of the documents stored in the documents dictionary, and adds a new field to each of the dictionaries in the array of the term frequencies. To do this, the pseudocode iterates through all the dictionaries stored in the documents\_dict array. For each of these, it splits the clean name into the separate words, to tokenize the string. It then iterates through all of the tokenized words. It also checks for any empty strings, in case there are multiple consecutive spaces within the string, so these are not included in the term frequency values. For every time a word repeats, it increases its value by the reciprocal of the number of words in the document, as explained in the equation above. However, if the terms were specified, it carries out the same process for only that string, and generates a singular dictionary for the term frequency for the string, and returns it instead.

These cannot easily be stored, as they will change whenever the database is searched, so must be calculated when needed.

#### Generating IDF value

The inverse document frequencies (IDF) values can be calculated using the formula, as explained in the analysis section. The formula has been repeated here, where is the number of documents, , in the corpus, . In this project, the documents array is the corpus.

To be able to calculate the IDF values, it needs to have a list of unique words. This MySQL query will get these words from the database, as previously explained in the database queries section. It is repeated below, and the results of which are assumed to be stored in a 2D array, containing an array for each result, with an item in it for each column in the results. It is stored in an array called unique\_words.

| SELECT word\_id,  word  FROM unique\_words; |
| --- |

The IDF values can be stored, as they will not change when a search query is made. This also means that the process to generate them needs to store these values. The query below, which was also explained in the database queries section updates the value stored in the unique\_words table. The query below can be used to update this, and is assigned to a function, UPDATE\_STORED\_IDF(idf\_value, word\_id), which inserts the values into the query, and executes it.

| UPDATE\_STORED\_IDF(idf\_value, word\_id) |
| --- |
| UPDATE unique\_words  SET idf\_values=<idf value>  WHERE word\_id=<word id>; |

| FUNCTION generate\_idf\_values():  INTEGER number\_documents = LENGTH( documents )  DICTIONARY idf\_values = {}  FOREACH record IN unique\_words:  INTEGER word\_id = record[0]  STRING word = record[1]  FLOAT idf = LOG( number\_documents / documents.COUNT( word ) )  UPDATE\_STORED\_IDF( idf, word\_id )  idf\_values[word] = idf  ENDFOR  RETURN idf\_values  ENDFUNCTION |
| --- |

This pseudocode generates a dictionary of the IDF values, and updates them in the database. It creates a dictionary of values as well, as this is faster to look up from memory, than loading from the database. It iterates through the list of unique words from the query, and calculates the IDF value using the formula. It then stores the values to the database, and adds it to the dictionary, with the key being a lookup for the idf value.

Accessing the items from the database will be very slow, so these values should be kept in memory to try and improve the query performance. This means the values should be loaded into memory when the wsgi script starts. The following query, previously explained in the database queries section, can retrieve the words and the IDF values from the database. The results are in a 2D array, and stored in the query\_result variable.

| SELECT word,  idf\_values  FROM unique\_words; |
| --- |

| FUNCTION load\_idf\_values():  DICTIONARY idf\_values = {}  FOREACH record IN query\_result:  idf\_values[record[0]] = record[1]  ENDFOR  RETURN idf\_values  ENDFUNCTION |
| --- |

This pseudocode creates an empty dictionary, and for every record within the unique\_words table, creates a new item within the dictionary, with the word as a key. It the returns this dictionary.

#### Generating TF-IDF values

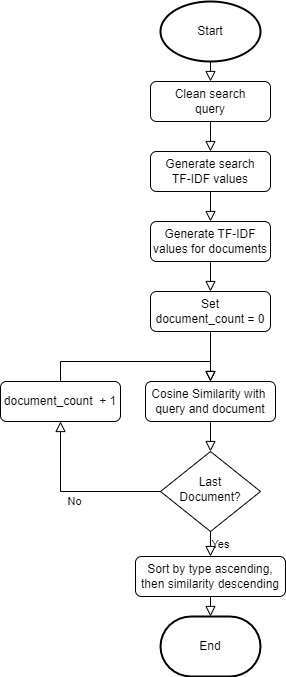
The TF-IDF values need to be generated for the search. This is dependent on the search terms, so cannot be be pre-calculated, and will need to be run every time a search is performed. This needs to either be for all of the documents, or the search query. The following pseucode can perform this, assuming that the documents\_dict and documents array have been created.

| FUNCTION generate\_tfidf\_values(document DEFAULT NULL, search\_terms DEFAULT NULL):  IF document == NULL THEN:  INTEGER count = 0  FOREACH document IN documents\_dict:  document\_words = document["words"].split( " " )  IF search\_terms == NULL THEN:  ARRAY new\_search\_terms = document\_words  ELSE:  ARRAY new\_search\_terms = search\_terms  ENDIF  DICTIONARY res = {}  FOREACH i IN new\_search\_terms:  res[i] = 0  ENDFOR  FOREACH i IN res.KEYS():  IF idf\_values.CONTAINS(i) AND document\_words.CONTAINS(i)  THEN:  res[i] = document["tf"][i] \* idf\_values[i]  ENDIF  ENDFOR  documents\_dict[count]["tfidf"] = res  count = count + 1  ENDFOR  RETURN NULL  ELSE:  ARRAY document\_words = document.split( " " )  DICTIONARY tf = generate\_term\_frequency\_values()  DICTIONARY res = {}  FOREACH i IN document\_words:  res[i] = 0  ENDFOR  FOREACH i IN res.KEYS():  res[i] = tf[i] \* idf\_values[i]  ENDFOR  RETURN res  ENDIF  ENDFUNCTION |
| --- |

This function generates the TF-IDF values, either for a given search query, or for all of the titles. It takes two parameters, the document and search terms. If the document is not specified, it generates all of the tf-idf values for all of the documents in the array. It uses the words in the documents, if the search\_terms are not specified, otherwise it uses the document words. Following this, it then multiples the term frequency and IDF value to give the TF-IDF values. Any entries that are not in both the IDF values and document\_words are excluded, and default to 0, as the dictionary is created with 0s for all the values, initially. Finally, it updates the documents\_dict value to add the TF-IDF value to it. However, if the document is specified, it generates the terms for the search query. The dictionary lookup for the search query is then returned, with the word as key, and the TF-IDF as the value.

#### Search

To search the database, the search query given by the user needs to be cleaned, the TF-IDF values for all the documents, and the search query need to be generated. For each of the TF-IDF values they then need to be compared to each other using cosine similarity. Finally, the results need to be sorted by similarity, but with authors first, then books then genres, prioritised above similarity strengths. The following flowchart outlines the logic for this process.



As previously mentioned, the type of each element is either ‘*a*’, ‘*b*’, or ‘*g*’, for author, book and genre. The results must have authors being first in the results, as by appending their name to the clean title for the book, books may come up first if the user is specifically looking for an author, as the match strength could be 100%. For this reason, authors must be the first results, regardless of whether books are included in the search. Books could also share the same name as a genre, but a user is more likely to be searching for a book than a genre. For this reason, books should come before the genres in the results. This is where the order of authors, books, then genres comes from. Sorting by the type (the representing letter of the result), in ascending order, will give the results in the required order. Sorting by the match strength, in descending order, then gives the results ordered by match strength, but with the authors first, then books, and finally genres. The following pseudocode can generate a list of the IDs of the results to a search query.

| FUNCTION tfidf\_search( terms ):  STRING clean\_terms = CLEAN\_DATA( terms )  ARRAY term\_arr = clean\_terms.split( " " )  DICTIONARY search\_tfidf = GENERATE\_TFIDF\_VALUES( document = clean\_terms )  ARRAY results = []  GENERATE\_TFIDF\_VALUES( search\_terms = term\_arr )  FOREACH document IN documents\_dict:  FLOAT similarity = 0  FLOAT a\_total = 0  FLOAT b\_total = 0  tfidf = document["tfidf"]  FOREACH k in term\_arr:  similarity = similarity + (search\_tfidf[k] \* tfidf[k])  a\_total = a\_total + (search\_tfidf[k] \*\* 2)  b\_total = b\_total + (tfidf[k] \*\* 2)  ENDFOR  IF similarity > 0 THEN:  similarity = similarity / ((a\_total \* b\_total) \*\* 0.5)  document["similarity"] = similarity  DICTIONARY temp\_dict = {  "type": document["type"],  "similarity": document["similarity"],  "id": document["id"]  }  result.APPEND( temp\_dict )  ENDIF  ENDFOR  results = SORT( results BY "type" ASCENDING, "similarity" DESCENDING )  RETURN results  ENDFUNCTION |
| --- |

Another requirement for the search is to be able to search for a book by the ISBN number. If this is the case, it should always be the first item that appears in the search results. This can be done by first checking whether the search query is a numeric value, as if it is it is most likely a ISBN number. The database should then be queried for this value, and if it is not present, it is not in the database. Regardless of whether a result is found, more results should still be generated. This allows for other matches to be found, especially if it was not a search for an ISBN.

The following pseudocode can perform a database search, including limiting the results to a specified number, including the data to send to the client, and searching by ISBN as well. Like above, this requires the documents\_dict and documents to have been created. It also uses books.GET\_SUMMARY\_ISBN() to get the cover image, title and authors name by ISBN, and books.GET\_SUMMARY\_ID() to get the data by book ID, genres.ID\_TO\_NAME() and authors.ID\_TO\_NAME() convert an ID to the name of the corresponding value, and IS\_NUMERIC() to check whether the string it is applied to can be cast to an integer. It assumes that an error will be thrown if a book is not found in the database, which is called BookNotFoundError. Finally, result\_limit is a variable that specifies the number of results that should be given in response to a search query.

| FUNCTION database\_search( query ):  DICTIONARY output\_dict()  INTEGER addition = 0  IF search.IS\_NUMERIC() THEN:  TRY:  output\_dict[0] = books.GET\_SUMMARY\_ISBN( query )  output\_dict[0]["type"] = "b"  output\_dict[0]["certainty"] = 100.0  addition = 1  CATCH (BookNotFoundError):  PASS  ENDTRY  ENDIF  ARRAY search\_result = tfidf\_search( query )  search\_result = search\_result.FIRST( result\_limit - addition )    INTEGER count = 0  FOREACH result IN search\_result:  IF result["type"] == "b" THEN:  DICTIONARY temp = books.GET\_SUMMARY\_ID( result["id"] )  temp["type"] = "b"  output\_dict[count + addition] = temp  ELSEIF result["type"] == "a" THEN:  output\_dict[count + addition] = {  "name": authors.ID\_TO\_NAME( res["id"] ),  "type": "a",  "author\_id": res["id"]  }  ELSE:  output\_dict[count + addition] = {  "name": genres.ID\_TO\_NAME( res["id"] ),  "type": "g"  }  output\_dict[count + addition]["certainty"] = round( result["similarity"] \* 100, 1 )  count = count + 1  ENDFOR    RETURN output\_dict  ENDFUNCTION |
| --- |

This pseudocode first generates a dictionary to store the output in. It then checks whether the search string is numeric, as if it is a number, it is most likely an ISBN. If it is numeric, it then gets the summary data by the ISBN number, and adds it to the first location in the dictionary. If it does exist, and is added, the addition variable is increased by 1, so the result limit can still be obeyed and not exceeded. Furthermore, if it does not exist, it will throw a BookNotFoundError, which then means that the book with the ISBN does not exist, and so can be skipped. After checking for ISBN numbers, it performs the basic search, using the previous method. The results of this are restricted to the specified number in the result\_limit constant. The addition variable is subtracted from this, so if an ISBN was found the number included from the subsequent search would be one less, so the total number of the results still obeys the constant. As the addition variable is set to 0 initially, if an ISBN is not found, it has no effect on the number of results. Finally, it converts these into dictionaries that can be used by the client, with the cover image, title, author and book ID for books, name and author ID for authors, and name for genres. Each of these dictionaries also has their corresponding type and certainty, converted into a percentage to one decimal place.

### Recommendations

#### Fitting the model

To fit the model, it should use weighted alternating least squares to determine the book factors and user factors, the factors being the weightings for each genre in the database. It should iterate many times, so that these two matrices converge, to give a good approximation of the real matrix of the explicit data. The difference between the calculated approximation and real matrix should be as low as possible, which can be measured using the mean squared error, which should be as low as possible. This should maximise the following objective function, which was given earlier in the analysis.

The weighted alternating least squares (WALS) should continue to iterate until the mean squared error is very low. There are two approaches to this, either repeating until the mean squared error is below a certain threshold, or running iterations, and assume that it has converged sufficiently. The risk with using the ‘repeat until’ method is that the mean squared error does not go below the required threshold, either at all, or for a large number of iterations. This would therefore limit when the model can be refit, and therefore when new recommendations can be generated. Conversely, specifying a number of iterations is more of a heuristic approach, may mean that they may converge further, and the accuracy is then likely to be lower, and more inconsistent. It does guarantee the length of time required to fit the model will be consistent, and the number of iterations can also then be specified to performance of the available hardware. This would also allow the recommendations to be comfortably retrained and recalculated overnight.

Using the limited number of iterations approach, also has the advantage of not needing to calculate the mean squared errors each time. This is likely to be very expensive, so, therefore, given seconds, it is likely to be able to perform significantly more WALS iterations. However, the mean squared errors would still need to be calculated for when the hyperparameter is being chosen, as it needs to be minimised as much as possible, and as fast as possible.

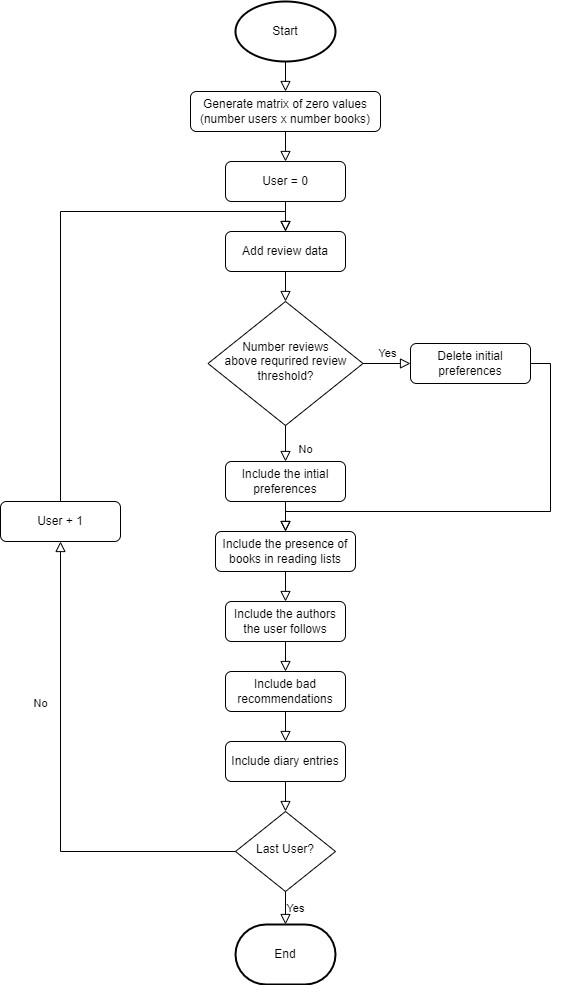
| book\_factors = MATRIX( number\_books, number\_factors )  user\_factors = MATRIX( number\_users, number\_factors )  IF debug THEN:  ARRAY test\_mse\_record = []  ARRAY train\_mse\_record = []  FOR iter = 0 TO number\_convergence\_iterations:  PRINT( "Iteration " + (iter + 1) + " of " +  number\_convergence\_iterations + "start.")  user\_factors = wals\_step( train, book\_factors )  book\_factors = wals\_step( train.transpose(), user\_factors )  predicted = predict()  train\_mse\_record.APPEND( mean\_squared\_error( train, predicted ) )  test\_mse\_record.APPEND( mean\_squared\_error( test, predicted ) )  PRINT( "Iteration " + (iter + 1) + " of " +  number\_convergence\_iterations + "end.")  ENDFOR  ELSE:  FOR iter = 0 TO number\_convergence\_iterations:  user\_factors = wals\_step( train, book\_factors )  book\_factors = wals\_step( train.transpose(), user\_factors )  ENDFOR  save\_book\_genres()  ENDIF |
| --- |

This pseudocode uses the constants number\_books, number\_users, and number\_factors, which are the number of books, users and genres respectively. The method save\_book\_genres is a procedure which stores the book factors matrix to the database. The constant number\_convergence\_iterations is an integer which stores the number of converge iterations to be performed. The function predict calculates the review matrix approximation that is used to generate the recommendations, using the equation , where is the user’s factor matrix, and is the book factor matrix. Finally, the function wals\_step calculates the other matrix, having fixed the first. The following equation is the derivative of the objective function, rearranged to find one of the factors. Substituting either matrix into this equation will give the other, as given in the analysis.

The pseudocode generates two random matrices for the book and user factors. It then performs iterations, to allow the two to become a good approximation of the actual matrix. Each iteration it performs a WALS step, to give both new matrices. If it is being debugged/tested, it also creates two arrays to store the mean squared errors between the actual and the calculated matrix, so the hyperparameter can be trained, so this is as low as possible, and as quickly as possible. It adds these results to the records. If it is not being tested, it saves the book factors to the database, so they can be used later. It does not do this if it is being tested, as this is expensive, and would also overwrite the current data, which is not ideal if the hyperparameter is not chosen correctly.

#### Generating review matrices

The following flowchart shows how the actual review matrix with the explicit data is generated. It includes using the explicit data given by the user (the reviews and the diary entries), and the implicit data, such as adding an item to a reading list, having read an item or following an author.



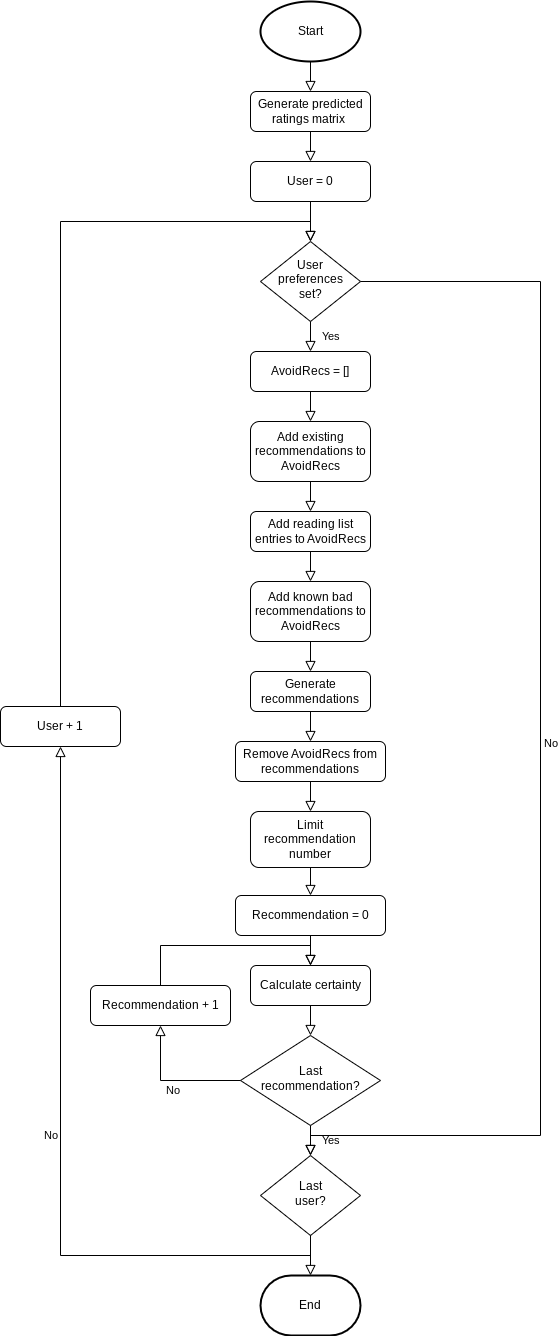
#### Generating recommendations

Generating the recommendations is fairly simple to perform. The estimated matrix to be used is given by multiplying the user factor matrix with the transposed book factors matrix. The best recommendations for each user are then the columns with the highest value in them for each user (row). A simple example of what would be expected is shown below, using 3 users, 6 books and 2 genres (factors). The recommendations that should be chosen are then colour coded, with the best being green, middle being yellow, and the worst being red. Note that this is not the best example, and is not representative of the actual model to be used, as there needs to be a significant number of books and genres for it to work effectively. This means that in this example most of the results are very similar, and all the users would be recommended the two top books.

|  |  | |  | **Book 1** | **Books 2** | **Book 3** | **Book 4** | **Book 5** | **Book 6** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  | 0.3741 | 0.0595 | 0.4911 | 0.6361 | 0.3268 | 0.3471 |
| 0.3741 | 0.5034 | 0.94 | 0.8347 | 0.8002 | 0.3741 |
|  |  | | | | | | | | |
| **User 1** | 0.8348 | 0.6199 |  | 0.54420327 | 0.36172826 | 0.99267628 | 1.04844681 | 0.76885662 | 0.52166367 |
| **User 2** | 0.2111 | 0.7839 | 0.3722295 | 0.40717571 | 0.84053721 | 0.78860204 | 0.69626426 | 0.3665298 |
| **User 3** | 0.2134 | 0.3618 | 0.2158232 | 0.19482742 | 0.44489274 | 0.4377382 | 0.35925148 | 0.20942052 |

In this example, books 3 and 4 should be recommended to all of the users as the top choices. This is because the dot product between the book 3 and book 4 vectors is the greatest for all of the users. However, there are situations where it would not be appropriate to recommend a user a book. These are when it has been read, is currently reading, or the user wants to read it, when the user has explicitly marked it as a bad recommendation previously, or it is already in the user’s list of recommended books. The following flowchart shows the logic for this process.

The certainty for each recommendation needs to be calculated separately, as the value given within the matrix cannot be assumed to be between 0 and 1, as shown in the table above. This also means that the certainty of the recommendation needs to be calculated separately. This is done after limiting all of the recommendations to a specified number, as it will be expensive, so should be done for the fewest number of recommendations possible.



#### Calculating recommendation certainty

As mentioned above, the value in the review matrix calculated cannot be assumed to be between 0 and 1. This means that it cannot be multiplied by 100 to convert to a percentage. Multiplying two matrices together is similar to dot product, in fact the dot product between vectors formed from a single user’s factors and a single book’s factors will be the same as that found in the corresponding cell in the resulting matrix of the multiplication of the overall factors. This cannot be reversed to find the magnitude of each vector. The following formula can be used to calculate the certainty of recommendation of book to user . is the user factor matrix and is the book factor matrix.

The following pseudcode takes the book vector and the user vector. These can be easily retrieved when the recommendations are calculated. It then calculates the dot product between the two vectors. This is done by multiplying each of the corresponding terms together, and summing the results. It also calculates their magnitudes, by adding the squares of all the terms, and then raising it to the power of a half to square root it. This is then used to calculate the certainty of the recommendation, between 0 and 1.

| FUNCTION calculate\_certainty( book\_vector, user\_vector ):  INTEGER number = LENGTH( book\_vector ) - 1  FLOAT dot\_prod = 0  FOR count = 0 TO number:  dot\_prod = dot\_prod + (book\_vector[count] \* user\_vector[count])  ENDFOR  FLOAT abs\_book\_vector = 0  FOREACH i IN book\_vector:  abs\_book\_vector = abs\_book\_vector + (i \*\* 2)  ENDFOR  abs\_book\_vector = abs\_book\_vector \*\* 0.5  FLOAT abs\_user\_vector = 0  FOREACH i IN user\_vector:  abs\_user\_vector = abs\_user\_vector + (i \*\* 2)  ENDFOR  abs\_user\_vector = abs\_user\_vector \*\* 0.5  FLOAT similarity = dot\_prod / (abs\_book\_vector \* abs\_user\_vector)  RETURN similarity  ENDFUNCTION |
| --- |

### Configuration

For multiple aspects of this project, there are different factors that can be configured. These can be categorised into nice-to-have features, and essential ones. They can impact the speed of the response, and therefore should be suited to the system performance, number of users of the system, number of concurrent users and number of hits. They could also impact the formatting and appearance of the site, which could then be configured to suit the tastes of the website manager. Most importantly, however, parameters for recommendations need to be configured depending on the user’s. This must be easy to achieve, so a configuration file is essential, especially one that is intuitive to use.

#### Existing configuration methods

A configuration file could be created using YAML or JSON. YAML (YAML, 2021), is a “complete data serialization language”, which is often used for software configuration. It contains many features that are not necessary for this project, and would likely come with a performance hit that is not justifiable. Furthermore, it does not have clear support for binary strings, which would be required for the salt for the hashing of passwords. JSON does not have support for binary as well, and is much less readable than YAML. For ease of managing the system, a custom system should be used. This therefore allows only the required data types to be included, and only the required level of specificity to be used, which should help improve performance. Furthermore, it also allows some of the features of YAML, which are not as clear, namely the headings, to be improved, and made more readable.

#### Project configuration specification

| header1:  field1 str: abc  field2 int: 5  header2:  field3 bool: true  field4 float: 1.1  miscField5 bin-str: def | header1 field1 >> “abc”  header1 field2 >> 5  header2 field3 >> True  header2 field4 >> 1.1  miscField5 >> b“def” |
| --- | --- |

*Figure 33 (Left): Example configuration file Figure 33 (Right): Example configuration queries*

Figure 33 (left) shows the proposed configuration file format. All fields must be one word, including headers. Headers must be followed by a colon, and any subsequent fields that fall under the heading must be indented, either using four spaces or tabs. This differs to YAML, as it does not need dashes to act as bullet points, which help improve the clarity, and how easy it is to write. Any lines that do not have the colon, and are not indented do not belong to a heading, and are miscellaneous items. Finally, each item is made up of three parts. The variable name, followed by the datatype and a colon and space, and then the value. This is clear, and allows for all the necessary datatypes, which cannot be provided by either YAML or JSON. To then query the configuration file, a query string, which consists of the header, then the variable name is used. This then returns the corresponding value, as shown in Figure 33 (right).

The project does have a range of different parameters, lots of which do not necessarily need to be changed. For this reason, there should be a way of storing default values for each of the items. This file does not need to be user-serviceable, and also does not require custom datatypes, so can use a JSON file to set these. This also is unlikely to have the same overhead as using the custom configuration method, as it will require significantly less processing.

#### Configurable Parameters

##### MySQL

The MySQL parameters control the database, and store all the required information to connect to the project database. They fit under the ‘mysql’ header.

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| username | String | wsgi | mysql username |
| schema | String | OpenBook | mysql schema |
| host | String | localhost (127.0.0.1) | mysql host |
| password | String | - | mysql password |

Username: Used as the connection identifier to the database. It can be assumed to be wsgi, as this is a logical name for the connection, as it is the name of the script that will be used.

Schema: Used to identify the database to use on the specified machine. It can be assumed to be OpenBook, as this is the name of the project.

Host: Used to identify the machine that can be used, takes an IP address. It can be assumed to be localhost (127.0.0.1), as it is most likely to be on the same machine as the web server.

Password: Used to authenticate the database connection. It cannot be assumed, as it should be project specific, for security.

##### Passwords

The passwords parameters specify how passwords should be hashed, before they are stored. They fit under the ‘passwords’ header..

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| hashing\_algorithm | String | sha256 | passwords hashing\_algorithm |
| number\_hash\_passes | Integer | 100,000 | passwords number\_hash\_passes |
| salt | Binary String | - | passwords salt |

hashing\_algorithm: Used to specify the hashing algorithm to be used. It can be assumed, as SHA256 is secure enough, and offers good performance on a Raspberry Pi

number\_hash\_passes: Used to specify the number of passes to be made by the hashing algorithm. It can be assumed, as 100,000 is as high as is reasonably possible on a Raspberry Pi, and according to (Meltem et al., 2010), it should be as high as possible. On higher performance hardware, this should be changed, but this offers a good baseline.

salt: Used to improve security of the password hashes. It is a binary string, as the hashlib method that is likely to be used needs it to be in binary. It cannot be assumed, as it could be a security risk if it was.

##### Home

The home parameters includes the configurable options for the home page, and any other similar pages. They fit under the ‘home’ header.

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| number\_home\_summaries | Integer | 8 | home number\_home\_summaries |
| number\_about\_similarities | Integer | 10 | home number\_home\_summaries |

number\_home\_summaries: Used to specify the number of summaries sent to the client to be displayed on the home page, or any other similar page, and will limit it to a maximum of this value even if the viewport is wide enough to accommodate more. It affects the performance, as more data needs to be retrieved from the database, and more data to be transmitted and requested, which increases response times. This can be assumed to be 8, as it is a reasonable trade-off between appearance and performance.

number\_about\_similarities: Used to specify the number of similar book summaries that are listed on a book’s about page. This will impact performance, as more data needs to be fetched and transmitted and requested, which will increase response times. This can be assumed to be 10, as it is a good trade-off between functionality and performance.

##### Recommendations

The recommendations parameters include the configurable parameters for the recommendations model, as well as any associated parameters which may affect the performance of the system.

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| number\_converge\_iterations | Integer | 100 | recommendations number\_converge\_iterations |
| hyperparameter | Float | 0.1 | recommendations hyperparameter |
| inital\_recommendation\_matrix\_value | Float | 0.5 | recommendations inital\_recommendation\_matrix\_value |
| reading\_list\_percentage\_increase | Float | 0.5 | recommendations reading\_list\_percentage\_increase |
| author\_following\_percentage\_increase | Float | 0.5 | recommendations author\_following\_percentage\_increase |
| bad\_recommendations\_matrix\_value | Float | 0.5 | recommendations bad\_recommendations\_matrix\_value |
| minimum\_required\_reviews | Integer | 10 | recommendations minimum\_required\_reviews |
| number\_recommendations | Integer | 10 | recommendations number\_recommendations |

number\_converge\_iterations: Used to specify the number of iterations of the weighted alternating least squares used to find the user and book factors. Increasing this increases the accuracy, but also increases the time required. It can be assumed, as 100 is a good trade-off between performance and accuracy on a Raspberry Pi. It can be tuned depending on the hardware.

hyperparameter: Used to prevent very frequent items or frequent users dominating the model, and needs to be tuned to the users. It can be assumed, as 0.1 works well with the training data set.

inital\_recommendation\_matrix\_value: Used to specify the starting weighting for initial specified recommendations, which will impact how long it takes for them to be removed from the recommendations. It can be assumed to be 0.5, as that will change quickly as more data is gained about the user’s preferences.

reading\_list\_percentage\_increase: Used to specify the importance of an item being present in a reading list for the recommendations. This is a value between 0 and 1, which is the percentage increase of being in a reading list. It can be assumed to be 0.5, as it has a significant effect, but would not overwhelm the recommendations.

author\_following\_percentage\_increase: Used to specify the importance of an item being present in a written by an author the user is following. This is a value between 0 and 1, which is the percentage increase of the book being written by an author the user has followed. It can be assumed to be 0.5, as it has a significant effect, but would not overwhelm the recommendations.

bad\_recommendations\_matrix\_value: Used to specify the importance of an item being marked as a bad recommendation. This should not be 0, to ensure that the preferences are not completely removed. This can be assumed to be 0.5, as explicit data will be considerable higher.

minimum\_required\_reviews: Used to specify the number of reviews required before initial preferences are removed and disregarded. Increasing this takes longer to remove the user’s initial recommendations, but will increase the initial accuracy. It can be assumed to be 10.

number\_recommendations: Used to specify the number of recommendations to be generated for each user. Increasing this number will significantly increase the time required to generate recommendations. It can be assumed to be 10, as this is a reasonable compromise with a good number of recommendations and performance.

##### Searching

These configuration options affect how the search feature works. They come under the ‘search’ header.

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| number\_results | Integer | 50 | search number\_results |

number\_results: Used to specify the number of results given when the search bar is used. Increasing this result gives more results, but requires more processing time, so response will be slower. It can be reasonably assumed to be 10, as it gives a wide range of options, and is unlikely to be too slow to be used with a Raspberry Pi.

##### Miscellaneous

These configuration options also affect the performance, or options of the project, however, they do not fit into a meaningful category, so therefore do not come under a header.

| **Variable Name** | **Data Type** | **Default Value** | **Query String** |
| --- | --- | --- | --- |
| session\_id\_length | Integer | 4 | session\_id\_length |
| debugging | Boolean | false | debugging |
| number\_display\_genres | Integer | 8 | number\_display\_genres |

session\_id\_length: Used to specify the length of the sessions generated. This needs to be adjusted depending on the number of users, and the system performance. It can be assumed to be 4 bytes, as this gives a large number of users, and is fast enough to provide fast responses to the user.

debugging: Used to specify whether debug logs should be left. It can be assumed to be false, as it is unlikely that when in use, debug logs would be necessary.

number\_display\_genres: Used to specify the number of genres that should be shown regarding a book. This includes on book about pages, and summaries for reading lists, diary entries, etc. It is likely to have a slight impact on performance, but is mostly for aesthetic preferences. It can be assumed to be 8, as this is a reasonable number for aesthetics and functionality.

#### Design

##### Exceptions

| **Exception** | **Usage** |
| --- | --- |
| ConfigInvalidDataForType | The value given for an item in the configuration file cannot be converted to the specified data type, like an integer being given a value of ‘a’. |
| ConfigInvalidDataTypeError | The datatype specified for an item in the configuration file is not known. |
| ConfigIndentationError | The item is indented when it should not be, such as when there has not been a header. |
| ConfigVariableNotFound | A configuration variable is accessed, but does not exist in the configuration file |

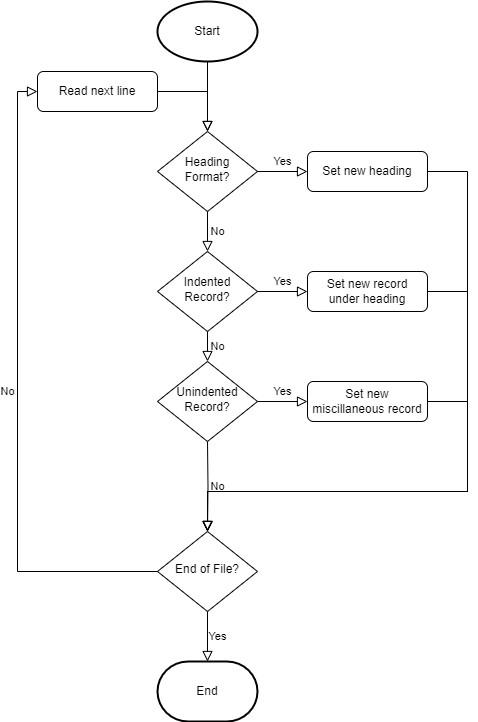
##### Regular Expressions

The headers, indented and unindented fields can be recognised and separated into their component parts using regular expressions. The syntax for this is the same as used by python, where values in brackets are taken as groups, and can be accessed individually. For clarity, these have been highlighted in bold

| **Regular Expression** | **Useage** | **How it works** |
| --- | --- | --- |
| (**\w+**):\s\* | Finding headers | Matches one or more alphanumeric characters or underscores, followed by a column, then any number of spaces.  The group is the header. |
| \t|\s{4}(**\w+**)\s+(**[\w-]+**)\s\*:\s\*(**.+**) | Finding indented items | Matches either four spaces, or a tab. It then matches one or more alphanumeric characters or underscores, then one or more spaces. Then it matches one or more alphanumeric characters, underscores and dashes. This is followed by any number of spaces, a colon, then any number of spaces. Finally it matches one or more characters.  The first group is the variable name  The second group is the datatype  The third group is the value |
| (**\w+**)\s+(**[\w-]+**)\s\*:\s\*(**.+**) | Finding unindented items | Matches one or more alphanumeric characters or underscores, then one or more spaces. Then it matches one or more alphanumeric characters, underscores and dashes. This is followed by any number of spaces, a colon, then any number of spaces. Finally it matches one or more characters.  The first group is the variable name  The second group is the datatype  The third group is the value |
| \s\* | Finding blank lines | Matches any number of space characters. |

### 

##### Processing and data access



This flowchart represents the basic logic of loading the configuration. First it needs to check whether the line is a header, and therefore it should go to the next line, which should be a member of it. If it isn’t, it should check whether it is indented, in which case it should belong to the previous heading. Finally, if it is not, it should therefore be an unindented record, and be a miscellaneous entry. This process should repeat for every line in the file.

| IF defaultConfigurationFilename != NULL THEN  defaultConfiguration = OPEN\_READ( "default\_config.conf" )  DICTIONARY hierarchy = jsonDictConvert(  defaultConfiguration.READ\_ALL()  )  ELSE:  DICTIONARY hierarchy = {}  ENDIF  configurationFile = OPEN\_READ( "project\_config.conf" )  lineNum = 1  heading = “”  WHILE NOT configurationFile.END\_OF\_FILE():  line = configurationFile.READ\_LINE()  headingRegEx = RegEx.MATCH( "(\w+):\s\*", line )  IF headingRegEx.EXISTS() THEN:  heading = headingRegEx.GROUP( 1 ).LOWER() + " "  ELSE:  IF heading != "" THEN:  entryRegEx = RegEx.MATCH(  "\t|\s{4}(\w+)\s+([\w-]+)\s\*:\s\*(.+)", line  )    IF entryRegEx.EXISTS() THEN:  string = heading + entryRegEx.GROUP( 1 )  hierarchy[string.LOWER()] = CAST\_TO\_TYPE(  entryRegEx.GROUP( 2 ).LOWER(),  entryRegEx.group( 3 ),  lineNum  )  ELSE:  heading = ""  ENDIF  ENDIF  IF heading == “” THEN:  externalRegEx = RegEx.MATCH(  "(\w+)\s+([\w-]+)\s\*:\s\*(.+)", line  )  IF externalRegEx.EXISTS() THEN:  hierarchy[externalRegEx.GROUP(1).LOWER()] = CAST\_TO\_TYPE(  externalRegEx.GROUP( 2 ).LOWER(),  externalRegEx.GROUP( 3 ),  line\_num  )  ELIF NOT RegEx.MATCH( "\s\*", line ).EXISTS() THEN:  RAISE ConfigIndentationError( line\_num )  ENDIF  ENDIF  ENDIF  lineNum = lineNum + 1  ENDWHILE |
| --- |

This pseudocode can be used to load a configuration from a file, including a default configuration JSON file. It produces a dictionary, where the key is the query string, the value is the corresponding value from the configuration file , and the datatype is also the same as that specified in the configuration file. To perform the look-up for configuration variables, this can therefore be O(1), as it is just a dictionary lookup.

This assumes that there is a pre-existing method to convert a JSON file into a dictionary. It also assumes there is a class for matching regular expressions to a string, and extracting groups from it. FInally, it assumes that the project configuration is in a file called project\_config.conf and the default JSON is in default\_config.conf. The CAST\_TO\_TYPE method used is a function that takes a value and a datatype and converts it to the corresponding data type. It also takes the line number, so it can be included in the error message if there is an issue with it.

Initially, it assumes there is not a header, so any records are first assumed to be miscellaneous. This ensures that there are no errors with records being added to headers that do not yet exist. It then loads the default configuration JSON object from the file and converts it into a dictionary. Any additional values in the configuration file will then be added to the dictionary, and any overwritten values will also be updated. This will ensure that the default configuration is included, but the user-specified one is prioritised.

The pseudocode loops for every line in the file, as well as having a counter for the line number which increases each iteration. It first checks whether the current line is a heading, using the previously explained regular expression. If the line matches this, it is therefore a heading, and the heading variable must be overwritten with this. If it does not match the heading regular expression, it must either be a record that is miscellaneous, a record that belongs to the previous heading, a blank line, or an invalid option.

If the heading is not an empty string, there was a previous heading. This means that it should check for an indented record, using the previously explained regular expression. If the line matches this form, the record exists, and should be added under the heading. This is then added to the dictionary, with the variable name in that line being appended to the previous heading, which forms the query string. The value is then stored in the hierarchy dictionary. If there is not a match, however, the section must have ended. To mark this, the header is then set back to an empty string.

If there has not been a heading, including if the heading was removed by the previous check, it checks whether it matches the miscellaneous record. It must also include if the previous check removed the heading, as otherwise an indented record followed immediately an unindented one, the unindented one will have been missed. If it matches the regular expression, it can be added to the dictionary in the same way as previously. If it does not match, it also checks whether it is a blank line using another regular expression, as previously explained, as if it is, it is valid. However, if the line does not fit any of the regular expressions, including that for a blank line, it is invalid.

The hierarchy dictionary is a dictionary of all the records in the configuration file. The key is the query string for the look-up, and the value is the corresponding value in the dictionary. This means that the time complexity of accessing the records is O(1). Calculating the query strings when the configuration is loaded will increase the amount of processing required beforehand, and will increase the time it takes to load. However, the project load time is not the most important factor, whereas having quick access to the records is significantly more important, and therefore the initial processing time is worthwhile.

##### Class definition

The class definition for the configurations class is as follows. It should use an object rather than a collection of functions and methods, as it can be viewed as an object, with its own associated data. The cast\_to\_type function should take a value, and a datatype and convert the value into that datatype. The load method should load the configuration from the file(s). Finally, the get method should get the requested value, as explained previously.

| **Configuration** |
| --- |
| * filepath * default\_file\_path * file\_config |
| * get * cast\_to\_type * load |

#### 

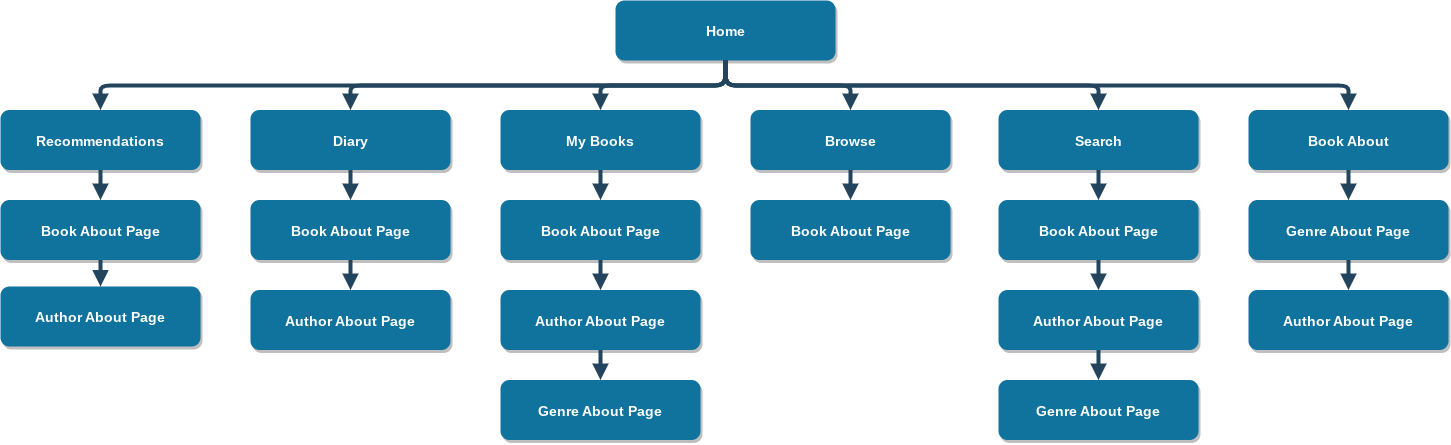
## Frontend

From here, whenever a (book) summary is mentioned, it refers to the book cover, book title, and author’s name. They are used to link from pages, such as the home screen, to a book about page.

### Website

In order to design the website, both the page hierarchy must already be designed, and the design for each of the pages.

#### Site hierarchy

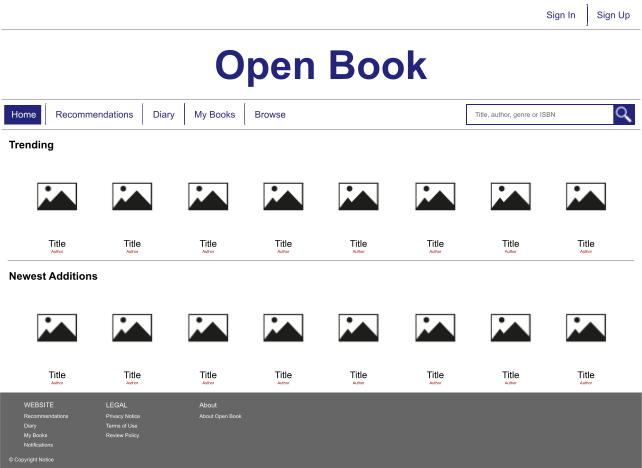


#### UI design

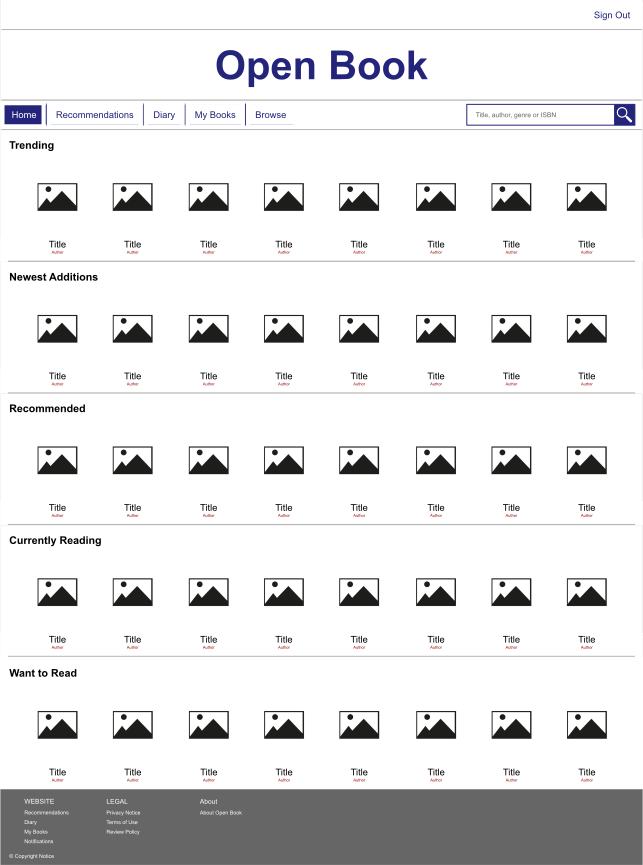
Each of these different pages show the footer at the bottom of the page content. However, on the real website, this would look wrong, with the footer floating if the content does not fill the entire viewport (the portion of the webpage that is being displayed). Therefore, it needs to be at the bottom of the viewport, unless the content exceeds this, in which case, it should be at the bottom of the content, outside of the viewport. For each of the page designs, however, it is depicted as being at the bottom of the page for simplicity.

##### Home Page

The home page is what the user will normally see first when visiting the website. Therefore, it should contain information that they would consider useful, which should vary depending on the user, or whether there is a signed in user. Each user should be shown the trending/popular books and the newest additions to the site. If the site was to be maintained effectively, this would then also include the newest publications as well.



If there is a user signed in, it should also show some information related to their interests and activities on the site, so it should show them their recommendations, currently reading and want to read.



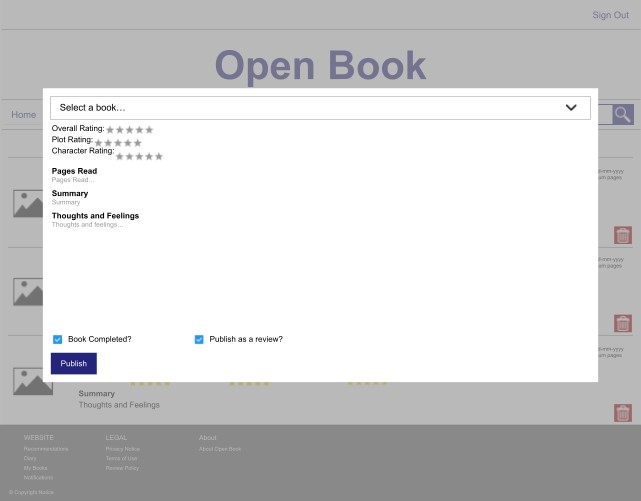
##### Recommendations Page

##### Diary Page

The diary page needs to be able to clearly show the users all their entries, and the data given in each clearly. The book can be shown using the cover, and title, and the ratings can be shown with stars. The summary should also be a higher contrast to the thoughts and feelings, so it stands out more, and gives the user an overview of their entry quickly.

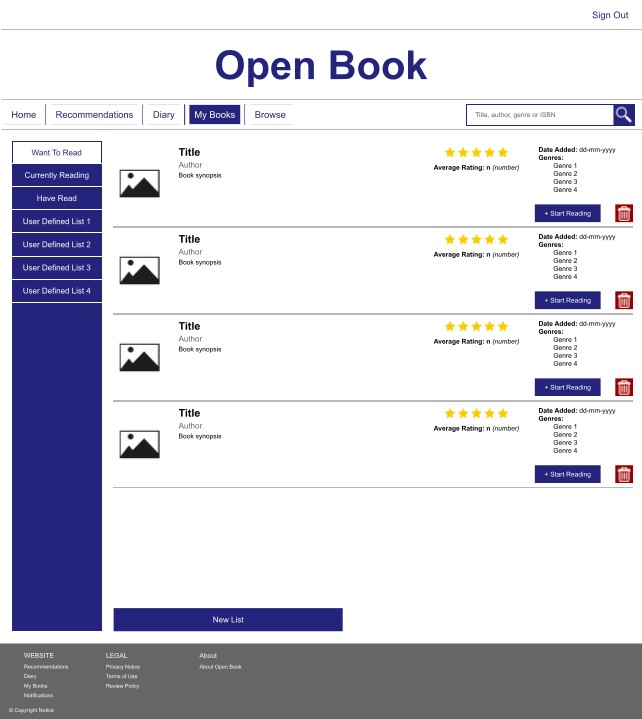
### 

When creating a new diary entry, the user needs to be able to select a book for the entry. This should be done using a dropdown menu, and limited to the items stored in their ‘Currently Reading’ list, which should help reduce the number of books needing to be displayed to a manageable amount. The rating stars for each section should be selected by clicking on the desired star, with all stars before it turning yellow on hover, and after it has been selected. The book completed checkbox should always be displayed, but the ‘publish as a review’ checkbox should only show when the ‘book completed’ box is selected, as the user should only be able to leave a review once they have read the book. Furthermore selecting the ‘book completed’ checkbox should also move the book from ‘Currently Reading’ to ‘Have Read’. The form should also appear as a popup, that can be cancelled by clicking outside of it, and the background greyed out to make the form clearly defined.



##### My Books Page

The My Books page should show the user all of their lists, automatically generated, and user designated, and all the entries in the lists. To change the list, there needs to be a sidebar, which shows all of the list names, and enables navigation between them. This navigation bar was inspired by that by (Waterstones, 2019), although the function is different and the displayed informWhen clicked, all the entries in that list should be shown, which should include the title, author, date added, genres, average rating, and book synopsis. There also needs to be a button to remove an entry from the list. Furthermore if it is the ‘Want to Read’ list, there should be a button to move the entry to the ‘Currently Reading’ list, with the text ‘Start Reading’. If it is the ‘Currently Reading’ list, there should be a button to move the entry to the ‘Have Read’ list, with the text ‘Mark as Read’. On the bottom of all the entries, irrespective of the list, there needs to be a button to create a new list, and on click, become a form, with an add button, as shown in the second image.



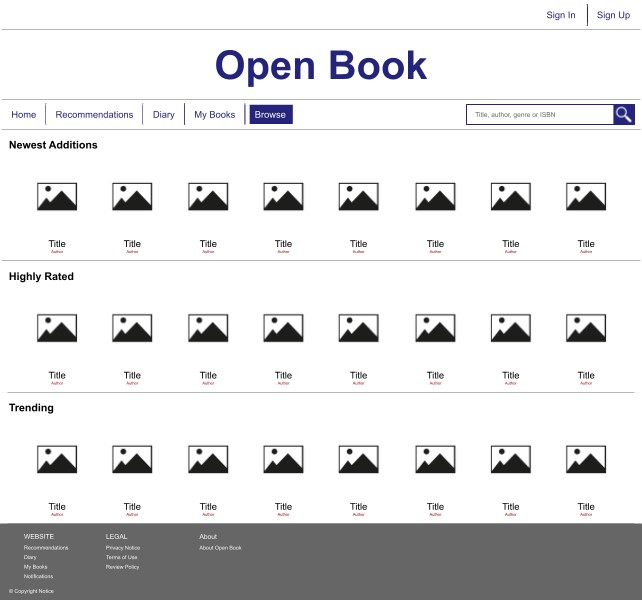
If the selected list is a user defined list, there should also be a ‘delete list’ button. The ‘Want to Read’, ‘Currently Reading’, and ‘Have Read’ lists are excluded from this as they are defined by the system, and are required for diary entries and leaving reviews.



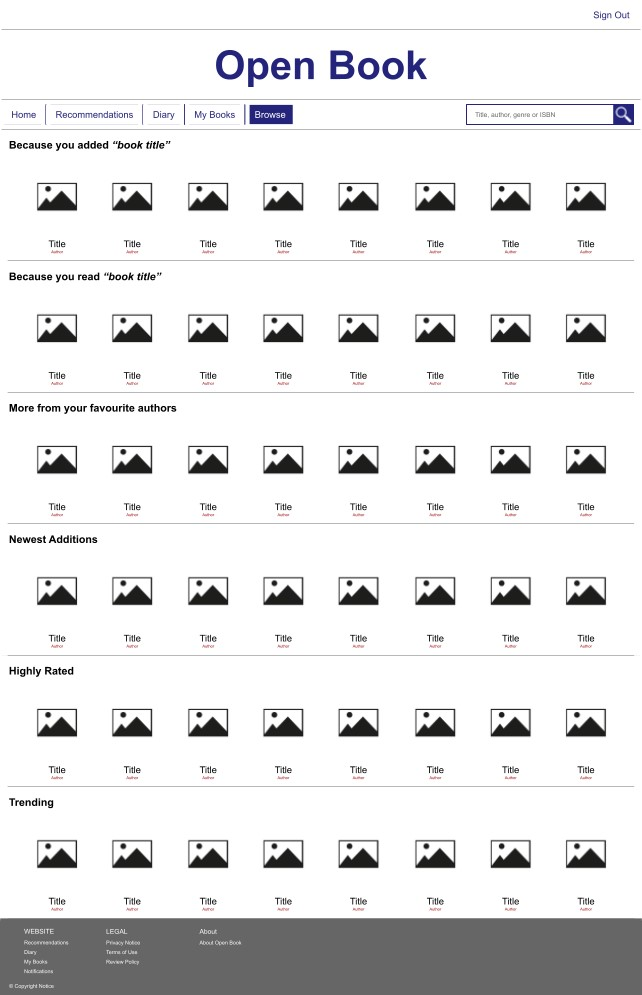
##### Browse Page

The browse button in the navigation bar is used for this page, genre about pages, author about pages, book about pages and the results of using the search bar. However, clicking on it directly will show the browse page.

The browse page should vary depending on whether the user is signed in. If the user is signed out, it is limited to displaying information similar to the home page. In this case, it should show the newest additions to the site, the highest rated books, and the trending books.



If the user is signed in, however there is more information that can be displayed. This information should be different from the data that is displayed on the home screen when they are signed in, and can also be less general than the home page, as the user has to look for this information, unlike the home screen, where it is the first thing that the user sees. Furthermore, the information should be related to the books that they would like to based off of their previous activity on the site. For this reason, it should include related books to their most recent books read, related books to the newest book they added to a reading list, and more books written by some of the authors they are following.

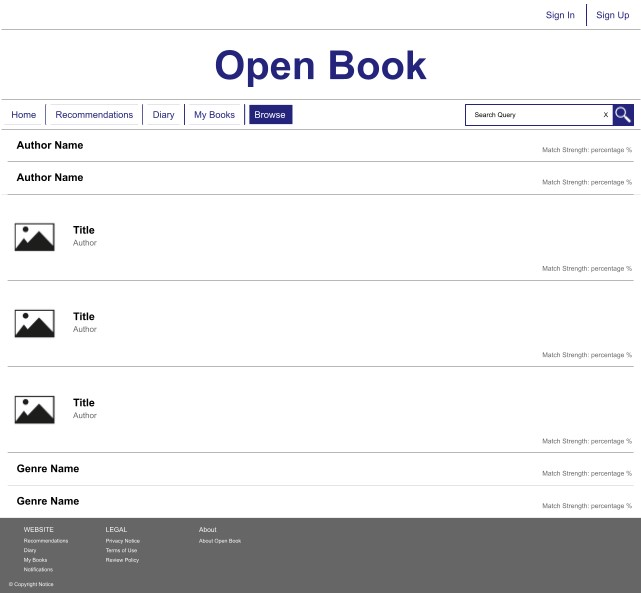


##### Search Page

The search page is found when a user uses the search bar to search for a book (using title or ISBN), genre or author. The browse link in the navigation bar is used, as it is browsing the database, and links to genre, author or book about pages, depending on the query and what the user searched. If there are no results to the search query, a message needs to be displayed to indicate that there are no items found, and possible fixes to the issue.

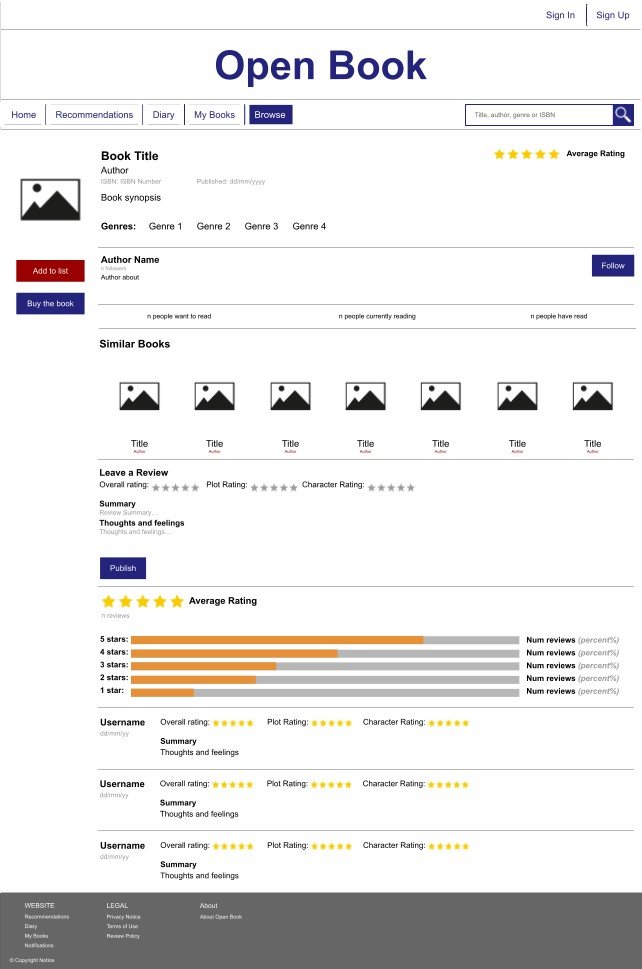


If there are results, these should be shown in rows, for clarity. Each of the items does not need to have much information about it, only enough to make it recognisable, as it should be easy for the user to identify the item they were looking for, as it should match, or be similar to the search query. Having less information displayed, helps reduce clutter, and therefore make it easier for users to find the desired item.

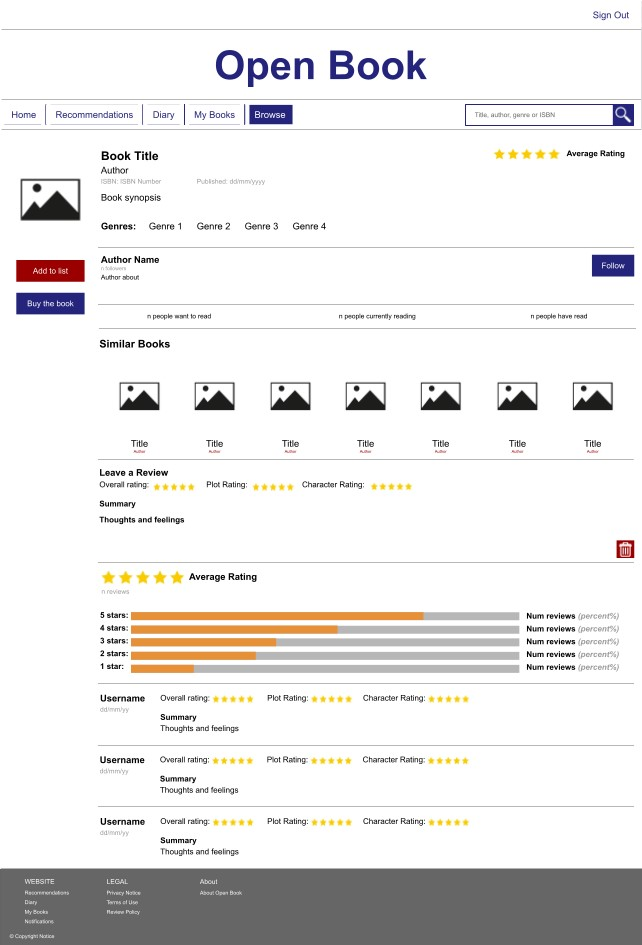


##### Book About Pages

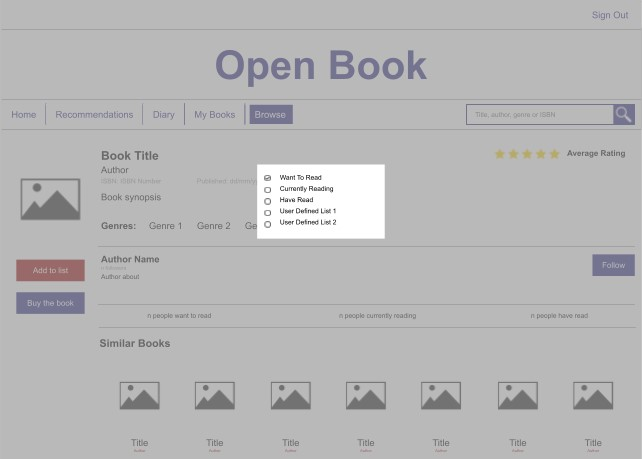
Each book that is stored within the database has lots of details relating to it, which should all be displayed. The cover image, title, author, synopsis, ISBN, genres and publish date are all important to list. However, the database also stores other information about the book, including the reviews for the book, and therefore a review breakdown (showing the percentage of reviews are for each star) and average rating. Consequently, there also should be a way to leave reviews or view and delete the current user’s existing review, if applicable. It also stores reading list data, and while, for privacy reasons, the users cannot be shown, a total of all the people wanting to read, currently reading, and have read, can be used. Furthermore the authors also have an about, and followers, so their about and number of followers can also be shown. Consequently, there also needs to be a button to follow/unfollow an author. Finally, a link to buy the book is also stored in the database, so there should also be a link to open that page in a new tab. Finally, a list of similar books can be generated, which should also be shown to the user, as it might give them suggestions on what they would like to read next. This design is based on the one by (Goodreads, 2018), however, it has fewer features, and less data.



If the user has left a review, this should be displayed instead, with an option to delete the review. It should be in a similar format to the reviews below, except the username and date added is excluded.

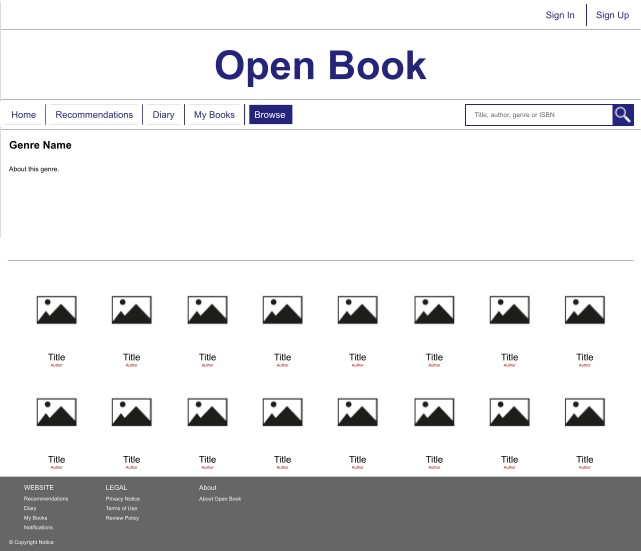


There also needs to be an option to add a book to a reading list. A book can only exist in one of ‘Want to Read’, ‘Currently Reading’ or ‘Have Read’ lists at one time, however, it can be in one of these lists, and any number of user defined lists at one time. The users need to be able to add it to a list. This should be done with a popup.



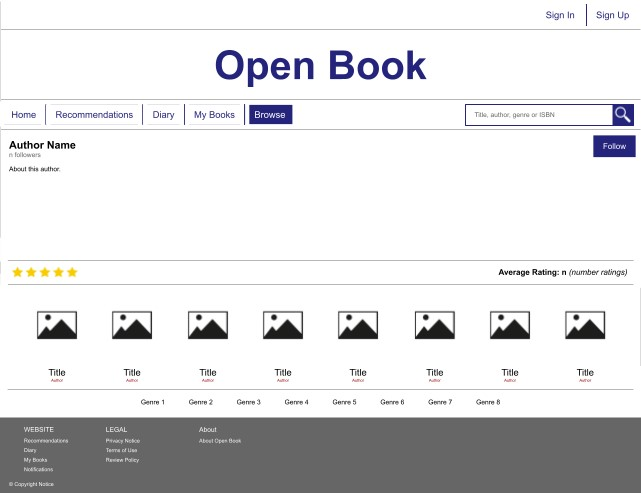
##### Genre About Pages

Each genre has a stored about, and a name, which should be clearly displayed. The books that have these genres should also be displayed, so the user can find other books that might interest them, if they like that genre.



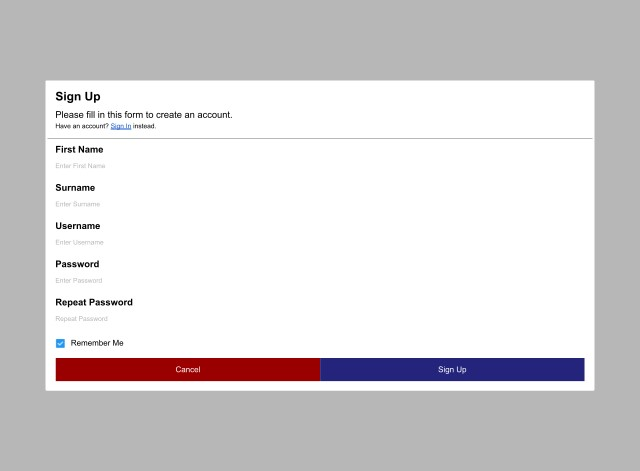
##### Author About Pages

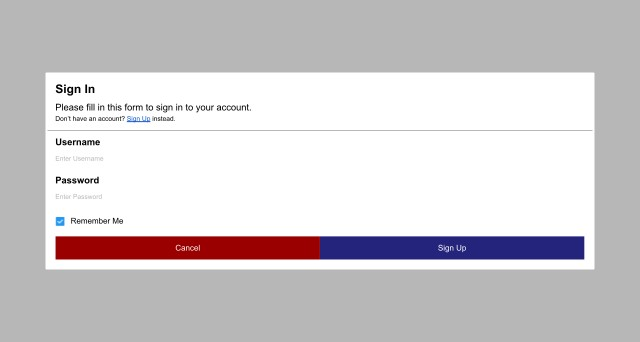
Each author, similar to the genres, has a stored name, and about. They also have a number of followers which can be displayed, and, consequently, a follow/unfollow button should be provided so a signed in user can follow or unfollow the author if they wish. Furthermore, the author’s books can be shown, as if a user likes one of their books, they can then look to find any others that they have written. The average review of the author, across all of these books can then be shown as well. Finally, the top genres that they have written for can be shown, which, in combination with the average rating, can be used to build a picture of what the author’s writing is like, and therefore the user can assess if they want to read any of their other books.



##### Sign In and Sign Up Pages

These pages are based on some examples from (w3schools, 2019), with different colours, and styles. These pop ups are shown when either the sign in, or sign up buttons are clicked, or when the link to the other is clicked on a popup. On the sign up form, the first name and surname fields are optional, but the username, password and repeat password fields are required. On the sign in form, both the username and password are required. This popup should be closed by clicking on the cancel button or outside of the popup.





## 

### AJAX Requests

All the project’s requests are either GET or POST requests. These work differently, and therefore require different fields, so are split into two different tables to accommodate this. Example responses to the requests have not been included, as would require many different fields.

Each URI is composed of up to three separate parts. URI’s that begin with ‘/cgi-bin’ should be redirected to the WSGI script for processing. The next section within the URI is the handler within the WSGI file that the required function can be processed by. The final part of the URI refers to the function that needs to be executed for the response.

#### GET requests

| **URL** | **Purpose** | **Asynchronous** | **Get parameters** | **Received Data Type** |
| --- | --- | --- | --- | --- |
| /html/page-name.html | Get a specific page html file to insert into the index.html | Yes |  | HTML |
| /cgi-bin/my\_books/get\_lists | Get users reading list names | Yes | session\_id (string) | JSON |
| /cgi-bin/my\_books/get\_list\_entries | Get entries into a specific reading list | Yes | session\_id (string)  list\_id (integer) | JSON |
| /cgi-bin/genres/about\_data | Get genre about data | Yes | genre\_name (string) | JSON |
| /cgi-bin/books/about\_data | Get book about data | Yes | session\_id (string)  book\_id (integer) | JSON |
| /cgi-bin/my\_books/get\_lists\_book\_target | Get reading lists with specific book in | Yes | session\_id (string)  book\_id (integer) | JSON |
| /cgi-bin/authors/about\_data | Get authors about data | Yes | author\_id (integer) | JSON |
| /cgi-bin/diary/get\_entries | Get user’s diary entries | Yes | session\_id (string) | JSON |
| /cgi-bin/home/get\_data | Get data to display on the home page (dynamically generated) | Yes | session\_id (string) | JSON |
| /cgi-bin/recommendations/get\_recommendations | Get user’s recommendations | Yes | session\_id (string) | JSON |
| /cgi-bin/search/search | Get results for a user’s search | Yes | query (string) | JSON |
| /cgi-bin/search/get\_browse\_data | Get data for browse page (dynamically generated) | Yes | session\_id (string) | JSON |

#### POST requests

| **URL** | **Purpose** | **Asynchronous** | **Data Type (out)** | **Data Fields (Out)** | **Received Data Type** |
| --- | --- | --- | --- | --- | --- |
| /cgi-bin/account/sign\_up | Create a new user account | Yes | JSON | first\_name (string)  surname (string)  username (string)  password (string) | JSON |
| /cgi-bin/account/sign\_in | Sign in to an existing account | Yes | JSON | username (string)  password (string) | JSON |
| /cgi-bin/account/sign\_out | Sign out and close session | Yes | String | session\_id (string) | text |
| /cgi-bin/my\_books/create\_list | Create a new, custom reading list | Yes | JSON | session\_id (string)  list\_name (string) | text |
| /cgi-bin/my\_books/remove\_list | Delete a custom reading list | Yes | JSON | session\_id (string)  list\_id (integer) | text |
| /cgi-bin/my\_books/remove\_list\_entry | Delete an entry into a reading list | Yes | JSON | list\_id (integer)  book\_id (integer)  session\_id (string) | text |
| /cgi-bin/my\_books/move\_list\_entry | Move an reading list entry to another list | Yes | JSON | list\_id (integer)  book\_id (integer)  session\_id (string) | text |
| /cgi-bin/books/delete\_review | Delete a book review | Yes | JSON | session\_id (integer)  review\_id (integer) | text |
| /cgi-bin/my\_books/add\_list\_entry | Add a new entry into reading list | Yes | JSON | list\_id (integer)  book\_id (integer)  session\_id (string) | text |
| /cgi-bin/authors/follow\_author | Follow an author | Yes | JSON | author\_id (integer)  session\_id (string) | text |
| /cgi-bin/authors/unfollow\_author | Unfollow an author | Yes | JSON | author\_id (integer)  session\_id (string) | text |
| /cgi-bin/books/add\_review | Leave a review for a book | Yes | JSON | session\_id (string)  book\_id (integer)  overall\_rating (integer)  plot\_rating (integer)  character\_rating (integer)  summary (string)  thoughts (string) | text |
| cgi-bin/diary/delete\_entry | Delete a diary entry | Yes | JSON | session\_id (string)  entry\_id (integer) | text |
| /cgi-bin/diary/add\_entry | Leave a new diary entry | Yes | JSON | session\_id (string)  book\_id (integer)  overall\_rating (integer)  plot\_rating (integer)  character\_rating (integer)  pages\_read (integer)  summary (string)  thoughts (string)  book\_completed (boolean)  as\_review (boolean) | text |
| /cgi-bin/recommendations/set\_user\_preferences | Set initial author preferences for recommendations | No | JSON | authors (array of integers)  session\_id (string) | text |
| /cgi-bin/recommendations/remove\_recommendation | Remove a recommendation, by marking it as bad | Yes | JSON | book\_id (integer)  session\_id (string) | text |
| /cgi-bin/recommendations/add\_list\_entry | Mark a recommendation as ‘Want to Read’ | Yes | JSON | book\_id (integer)  session\_id (string)  list\_id (integer) | text |

# Technical Solution

As done in the documented design section, the code can be broken down into frontend and backend. The frontend consists of HTML, CSS and Javascript only. The backend however, can be broken down further into the component classes for authors, genres, books, etc. It also includes the utility modules, like the configuration module, and the logging module. The final part is the WSGI file, which is the actual script that responds to the requests made through the javascript. This section will be arranged in these groups. All of the code will be available in Appendix C. It should be arranged, according to the following file structure.

| .  ├── MySQL  │ └── create\_tables.sql  ├── assets  │ ├── favicon.ico  │ └── logo.png  ├── backend  │ ├── components  │ │ ├── \_\_init\_\_.py  │ │ ├── accounts.py  │ │ ├── authors.py  │ │ ├── books.py  │ │ ├── diaries.py  │ │ ├── genres.py  │ │ ├── information\_retrieval.py  │ │ ├── reading\_lists.py  │ │ └── recommendations.py  │ ├── configuration.py  │ ├── data\_generation.py  │ ├── data\_structures.py  │ ├── environ\_manipulation.py  │ ├── flup.server.fcgi  │ ├── logger.py  │ ├── maintenance.py  │ ├── ml\_utilities.py  │ ├── mysql\_handler.py  │ ├── searching\_algorithms.py  │ └── wsgi.py  ├── css  │ └── sitewide.css  ├── html  │ ├── author.html  │ ├── book.html  │ ├── browse.html  │ ├── diary.html  │ ├── genre.html  │ ├── home.html  │ ├── my\_books.html  │ ├── recommendations.html  │ └── search.html  ├── javascript  │ ├── jquery.debounce.js  │ ├── jquery.js  │ ├── js.cookie.js  │ └── script.js  ├── default\_config.json  ├── index.html  └── project\_config.conf |
| --- |

During coding, line lengths were kept below 75 characters wherever possible. This is lower than the recommended 80, as it is roughly the number of characters that can fit on single line on an A4 page. This helps to keep the code readable within this document. However, in some scenarios this was impractical, and would significantly harm the readability of the code. In these scenarios, the code wraps around the lines, which is why there is occasionally strange indentation. Docstrings where also added to the python functions. These were removed for this document to try and keep the file size down, and would not be necessary, as all the functions are explained here, with their purpose and there parameters anyway.

## Requirements

| **Software** | **Version** | **Usage** |
| --- | --- | --- |
| Python | 3.9.2 | Handling requests made to the WSGI part of the project |
| Lighttpd | 1.4.59 | Runs the web server |
| MariaDB | 10.5.21 | Runs the database server |

| **Python Package** | **Version** | **Usage** |
| --- | --- | --- |
| numpy | 1.25.2 | More efficient matrix operations on large amounts of datasets. |
| sklearn | 1.3.1 | More efficient way of calculating mean squared error. |
| Gensim.summarization | 3.6.0 | Summarising reviews for generating test data |
| Flup.server.fcgi |  | Running the WSGI script and handling requests |
| mysql.connector | 2.2.9 | Connecting to the database and making queries |

| **JavaScript Package** | **Version** | **Usage** |
| --- | --- | --- |
| JQuery | 1.25.2 | Simplifying AJAX requests and changing formatting |
| JQuery debounce | 1.1 | Debouncing functions to reduce the number of executions of functions |
| JavaScript Cookie | 1.5.0 | Setting and managing cookies |

## Installation

This project was designed for use with a linux server, and therefore these installation instructions are for linux. Both the machines that this was tested on use Debian GNU/Linux distributions. These were a Raspberry Pi 4, and a Chromebook. They both require the same configuration steps, although the chromebook requires an additional step to allow other devices on the network to access the webserver. These instructions assumes that the code files are already present, and the current directory is the project’s root folder. This can be changed using cd path/to/project, where the path is the absolute path to the project’s root directory.

### Software installation

sudo apt update

sudo apt upgrade

sudo apt install lighttpd

sudo apt install mariadb-server

sudo apt install pip

sudo apt-get install cron -y

### Python package installation

sudo pip install mysql.connector

sudo pip install numpy

sudo pip install scikit-learn

sudo pip install sklearn

sudo pip install matplotlib

### MySQL Database setup

sudo mysql\_secure\_installation

sudo mysql -u root -p

CREATE DATABASE OpenBook;

CREATE USER "username"@"database ip" IDENTIFIED BY "user password";

GRANT ALL PRIVILEGES ON OpenBook.\* TO "username"@"database ip";

FLUSH PRIVILEGES;

QUIT;

### Lighttpd setup

Edit files to update port, and file paths.

cd lighttpd

sudo cp lighttpd.conf /etc/lighttpd/lighttpd.conf

sudo mv /etc/lighttpd/conf-enabled/90-javascript-alias.conf /etc/lighttpd/conf-available/90-javascript-alias.conf

sudo cp -r conf-enabled/ /etc/lighttpd/

cd -

chmod a+r /absolute/path/to/project

chmod a+x /backend/flup.server.fcgi

### Scheduled Jobs

This is given later in this chapter, but the commands are given here for convinience.

sudo crontab -e

0 1 \* \* \* python3 /absolute/path/to/project/backend/maintenance.py

### Configuration

After setting up the rest of the project, the configuration file needs to be updated. The mysql password field needs to be set to that used earlier when adding a user to the database. If the database name is also different to that given in the the schema field needs to be set. The password salt also needs to be set. It is important to note here that this salt cannot be changed, or any of the fields relating to creating passwords, after the first user is added, as it then changes how the passwords are hashed, and users will not be able to access their accounts.

If the database is created from the dump.sql, the password salt must be “+%E!mKZ(5%Z}k#pi(cPW!US8TU-J87”, as that was what was used to create the passwords for the user accounts (where the username is the user, followed by the number that the user was created, and the password is just password). If the generation script is used, it can be set to anything, assuming the configuration is set before running the generation script. If the script was used, and different data was used within it, the recommendations parameters will need to be tuned to the dataset, and added into the configuration file. The debugging field can also be set to true to enable debug logs, which can then be accessed using tail -f /tmp/output.log to access the end of the file which updates as new data is added, or cat /tmp/output.log to access the entire log. This can be enabled by adding the debugging field, or removed by excluding it.

mysql:

password str: 1qwerty7

passwords:

salt bin-str: +%E!mKZ(5%Z}k#pi(cPW!US8TU-J87

debugging bool: true

### Test data

In the downloaded files, within the MySQL folder, there is a dump.sql file. This contains test data for the project, which was generated using the data\_generation.py script, then dumping the database. This is given as a file, as it allows a reasonable amount of data to be used, whilst also being reasonable to process. This test data was taken from <https://grouplens.org/datasets/book-genome>.

sudo mysql -u root -p

USE OpenBook;

SET GLOBAL max\_allowed\_packet=1073741824;

SOURCE /absolute/path/to/project/MySQL/dump.sql;

EXIT;

sudo systemctl restart mariadb

Data can also be generated from <https://grouplens.org/datasets/book-genome> for larger datasets. To do this, the ZIP should be downloaded, and the contents of the raw folder moved to a new folder in the database called data/Orignal. Then running the data\_generation.py file then processes this and adds it to the database. This does take a long time to run. Truncating the reviews file can improve this, but will reduce the number of books that are then stored in the database. It can be done using truncate -s size filename, with the size being a value, such as 200K. If this is done instead of using the MySQL dump, the recommendations parameters need to be tuned to the data, and it may also perform poorly, as their may be too little data to generate accurate recommendations, or too much data for the server to process correctly.

sudo mysql -u root -p

USE OpenBook;

SET GLOBAL max\_allowed\_packet=1073741824;

EXIT;

sudo python3 data\_generation.py

sudo systemctl restart mariadb

### Web server control

Whenever any changes are made to the server, it should be restarted. This includes any chagnes to the configuration files. It should be started automatically when the server starts though, so it does not need to be started explicitly.

sudo systemctl restart lighttpd

sudo systemctl start lighttpd

sudo systemctl stop lighttpd

sudo systemctl status lighttpd

## Objectives

This following table gives all of the requirements that were outlined at the start of the project. They have also been labelled as to whether they were met, partially met, or not met. It is important to note that some of these objectives are not easily measurable, and are subjective. Furthermore, some of these requirements became invalid due to some design decisions, or where it was impractical. These were met to the extent that was reasonable for within this project. These decisions are then explained in the footnotes.

| **It must use a website** | | |
| --- | --- | --- |
| The website should have a modern design. | Met |  |
| The website must be easy and intuitive to navigate for third parties. | Met |  |
| Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system. | Met |  |
| Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| Each author stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| The users’ reading lists must be available on a dedicated webpage on the website. | Met |  |
| The users’ recommendations must be available on a dedicated webpage on the website. | Met |  |
| The users’ diary entries must be available on a dedicated webpage on the website. | Met |  |
| The website must use a hierarchical website structure, with all pages being able to be navigated to from the home page, or further pages in a logical format. | Met |  |
| The website’s colour theme should be white, black, navy, grey and red colour scheme. | Met |  |
| The website should be able to support at least five concurrent users. | Met |  |
| Pages should be switched and navigated between without refreshing the page | Met |  |
| **The website’s home page should present users with a list of frequently accessed information.** | | |
| The home page should give users a summary of their recommended books. | Met |  |
| The home page should give users a summary of what is in their want to read list. | Met |  |
| The home page should give users a summary of what is in their currently reading list. | Met |  |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. | Met |  |
| Book summaries should include the book’s cover image, author, and title. | Met |  |
| Each book summary should link the specific books about the page. | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
| **Users should have their own accounts on the website.** | | |
| Users must be able to create their own accounts on the website. | Met |  |
| Users must be able to log in to the website to access their personalised information. | Met |  |
| Recommendations and reading list entries must be kept private. | Met |  |
| Each user’s diary entries must be kept private | Met |  |
| The website should be able to save a user’s login | Met |  |
| **Personal information that is stored within the database should use appropriate data protection methods.** | | |
| All passwords should be hashed before being stored in the database. | Met |  |
| **Users should be able to create collections of books called reading lists** | | |
| Users should be able to record the books that they have read in the past. | Met |  |
| Users should be able to record the books that they are currently reading. | Met |  |
| Users should be able to record the books that they would like to read in the future. | Met |  |
| *Have Read*, *Currently Reading*, and *Want to Read* reading lists should be automatically created when a user signs up. | Met |  |
| Users should be able to create their own custom reading lists | Met |  |
| The user should not be able to delete the default *Have Read*, *Currently Reading*, and *Want to Read* reading lists | Met |  |
| Users should be able to delete their own custom lists | Met |  |
| Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list | Met |  |
| Each entry into a reading list should allow the user to delete the entry. | Met |  |
| Each entry into the *have read* reading list should also supply a link to all the user’s diary entries for that book. | Not Met |  |
| Each entry into a reading list should supply a link to the books about page. | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
| **The system should generate recommendations for the users.** | | |
| The system should generate recommendations that are unique to each individual user’s tastes. | Met |  |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met[[1]](#footnote-0) |  |
| Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre. | Met |  |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. | Met |  |
| Recommendation entries should have a method to move it to the *want to read* reading list. | Met |  |
| Recommendations should include whether the user follows an author | Met |  |
| Recommendations should include whether the book is in a reading list | Met |  |
| Recommendations should include the user’s diary entries | Met |  |
| Recommendations should be regenerated daily | Met |  |
| Recommendation entries should show the certainty of the recommendation | Met |  |
| **Users should be able to have a diary to record what they have read.** | | |
| Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating. | Met |  |
| Users must specify the number of pages that they read. | Met |  |
| When the user finishes the book, the user should be given the option to make the diary entry available as a review. | Met |  |
| Each diary entry should supply a link to the corresponding book about page. | Met |  |
| Users should be able to delete diary entries | Met |  |
| **Each book should have an about page, which gives all the related information about the book.** | | |
| The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book | Met |  |
| A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website. | Partially Met |  |
| It should have a method to add the book to the have read, currently reading, and want to read lists. | Met |  |
| There should be a list of books that are related to each other displayed. | Met |  |
| Each related book should link to the corresponding about page. | Met |  |
| **Each author should have a profile page.** | | |
| It should display the author’s average book rating. | Met |  |
| It should display all the genres that the author writes for. | Partially Met[[2]](#footnote-1) |  |
| It should show all the books that the author has written. | Met |  |
| It should supply an about the author section. | Met |  |
| It should show the author’s name (or the name that they write under if applicable). | Met |  |
| Users should be able to follow each author. | Met |  |
| The page should display how many followers each author has. | Met |  |
| **Users should be able to leave reviews about books that they have read.** | | |
| Users should be able to leave reviews about each book | Met |  |
| Users should be able to delete any reviews that they have left | Met |  |
| Reviews should be listed on the corresponding book about page. | Met |  |
| Reviews must have an overall rating. | Met |  |
| Reviews can have a character rating, plot rating, review summary, and review body. | Met |  |
| Reviews on the book should show the user’s username. | Met |  |
| Users should only be able to review a book if they have read it, marked by it being in the users’ *have read* list. | Met |  |
| **Each genre should have an about page** | | |
| It should display the genre’s name | Met |  |
| It should show an about summary for the genre, if applicable | Met |  |
| It should show the books that are part of the genre | Met |  |
| **There should be a quick way for users to find books, authors and genres.** | | |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website | Met |  |
| The browse page should display books related to the last one added to a reading list | Met |  |
| The browse page should display books related to the last read book | Met |  |
| The browse page should display books from the authors the user is following | Met |  |
| There must be a search bar that searches for books, authors and genres within the website. | Met |  |
| The search bar should be able to search by name or ISBN number | Met |  |
| **There should be a simple way to configure the system** | | |
| Parameters that can affect performance should be configurable | Met |  |
| A default configuration should be used, which can be updated as needed based upon available hardware | Met |  |
| Recommendations should be configurable based upon the books and data stored in the database | Met |  |
| Issues with the configuration should be clear, easy to understand, and simple to debug | Met |  |
| Configurable parameters should be modified through an external configuration file, which is in a clear and readable format | Met |  |

## Backend

### Utilities

#### Database Connection Module (./backend/mysql\_handler.py)

This module repackages the mysql.connector module in python. This was done as using the module to make queries is complicated, and cannot be easily passed to other objects either. Repackaging this into a module allows for the query times to be accessed, queries to be made easily, and the connection closed automatically when the object is destroyed, which is not done automatically with the actual module. Its class definition is as follows, as it was not included within the documented design.

| **Connection** |
| --- |
| * query\_time * user * password * host |
| * query * query\_time\_getter * connect * constructor * destructor |

This is not required directly for any of the objectives, however, it is required to make any database query, so is important.

| **Constructor for the connection class** | |
| --- | --- |
| def \_\_init\_\_(self, user, password, schema, host):  self.\_user = user  self.\_password = password  self.\_schema = schema  self.\_host = host  self.\_connect() # Establish database connection  self.\_query\_time = None | |
| This method is executed whenever a connection class instance is created. It saves the parameters given as private variables, so that they cannot be accessed, and establishes a database connection, using the private connect method. The query\_time variable is also set to None, as there has not been a query yet, so the query time cannot be given. | |
| **Encapsulation and data hiding:**  This demonstrates encapsulation and data hiding, as it contains the information required to connect to the database, as well as the mysql.connector connection instance. | |
| **Composition:**  It creates a mysql.connector connection instance within the connect method. This object is created within this class, so is a form of composition. It cannot be accessed from outside the object, as it is private, and when the connection object is deleted/destroyed, the mysql.connector connection object is also destroyed. | |

# 

| **Method to create a database connection** | |
| --- | --- |
| def \_connect(self):  self.\_connection = mysql.connector.connect(  user=self.\_user,  password=self.\_password,  host=self.\_host,  database=self.\_schema  )  self.\_cursor = self.\_connection.cursor()  # The cursor is used to execute queries | |
| This method establishes a connection to the database using the mysql.connector module. It also creates a cursor instance, which is used to query the database. The cursor is also private, so cannot be used outside of the class. The connection uses the stored private variables for the username, password, the host IP address, and the database name to be used at that location. | |
| **Abstraction:**  Connecting and creating a database connection is fairly complex, especially as the cursor needs to be created to be used. Including this within this method makes it simpler and easier to use. Furthermore, having it inside a method, instead of within the constructor, means it can be reused if the connection is lost or times out. | |
| **Composition:**  It creates a mysql.connector connection instance within the connect method. This object is created within this class, so is a form of composition. It cannot be accessed from outside the object, as it is private, and when the connection object is deleted/destroyed, the mysql.connector connection object is also destroyed. | |

| **Object destructor** | |
| --- | --- |
| def \_\_del\_\_(self):  self.\_connection.close() | |
| This is a custom destructor for the connection class, and is called whenever the object is deleted. Whenever this happens, the connection to the database is closed explicitly. In python, this destructor is always called whenever the object is deleted, including explicitly deleted, when an error occurs, or due to garbage collection. This means that it does not need to be explicitly closed, which makes it simpler and easier to use, and reduces the chances of the connection being left open. | |
| **Abstraction**  It is simpler to use, as it removes the need for the connection to be explicitly closed. This makes the database simpler and easier to query, as well as removing the need to consider the connection not being closed after an error has occurred. | |

# 

| **Database querying** | |
| --- | --- |
| def query(self, query):  start\_time = time.time() # Changes this first incase multi-threading  # is used in the future - a query may be made in a different thread.  self.\_query\_time = None  try:  self.\_cursor.execute(query)  except mysql.connector.Error:  self.\_connect() # Some databases specifies connections close  # after certain amount of time inactive. This repoens the  # connection if a timeout occurs  self.\_cursor.execute(query)  try:  result = self.\_cursor.fetchall() # Needs to come before the if  # statement as otherwise can result in 'unread result found'  # error  except mysql.connector.errors.InterfaceError:  result = [] # Incase the method does not provide any result, like  # INSERT  self.\_connection.commit() # Applies changes from the query to the db  self.\_query\_time = time.time() - start\_time  return result # Use tuples as they are faster | |
| This method executes a single MySQL query. It takes the query as a string, as a parameter. After this it sets the start time of the query, so that the time taken for it to be performed to be calculated. It then sets the query\_time variable to None, as the query time cannot be specified whilst a query is taking place.  It then tries to execute the code, catching the mysql.connector.Error, that would be thrown if the database connection has been closed unexpectedly. Some databases will close connections to them if they have not been used in a certain length of time. If this occurs, the connection is re-established, using the connect procedure again. The query is then attempted to be executed again as well. If an error occurs during this, it won’t be caught. However, this is not an issue, as the connection was just made, it will not be down to the connection expiring.  Following this, it then stores the data from the query, using the fetchall method on the cursor object. If the query did not return any data, the fetchall method will raise the mysql.connector.errors.InterfaceError. In this case, the result is just set to an empty list. After this is then commits any changes that were made using the cursor object to the database, so that the results of any INSERT statements, or other statements that modify and update the data stored within it, actually take effect.  Finally, it takes the start time away from the current time, to give the time taken in seconds (including decimal points). This value can be accessed through the getter method for the query time. It then returns the result of the query. This is a list of tuples, with each tuple within the list storing the results for a single row in the query result. Each tuple then has an item for each of the columns in the query. Tuples have been used over lists for three reasons. Firstly, they are normally faster than lists, and secondly, they make the code safe from any accidental modification (BYJU'S, n.d.). Tuples aren’t necessarily as easy to use as lists, but the fetchall method also returns it in this format, so changing it where it may not be necessary is an unnecessary increase in the complexity of the function. | |
| **Abstraction**  It is much simpler to use. To query the database, only a single method needs to be used, as it packages all of the timing, database reconnection, interface error handling if there was not a result given, and committing the changes to the database. This approach will be slower than using the mysql.connector module directly, as the type of query would be known. This would allow for the second try-except statement to be removed where applicable, and the commit method to be excluded for queries such as SELECT statements. The performance overhead from this approach is insignificant enough to be ignored, and this much simpler and faster approach is to be used. | |
| **Exception Handling**   | try:  self.\_cursor.execute(query)  except mysql.connector.Error:  self.\_connect()  self.\_cursor.execute(query) | | --- | | try:  result = self.\_cursor.fetchall()  except mysql.connector.errors.InterfaceError:  result = [] |   This method uses exception handling to ensure that querying the database is reliable, and any possible errors can be handled correctly, if it is logical to do so. The first try-catch ensures that the database connection has not been closed, and if it has, it will reopen the connection, and re-execute the query. Any errors that then occur from this second execution must therefore be unrelated, such as errors with the query that was performed, and so should not be caught.  The second try-catch statment is for when the results of the query are fetched. If there is not any data returned by the query, it would raise the InterfaceError exception. This is caught so that no errors are raised when using the method with INSERT statments, or any statments that will not give a result. This ensures that this function is robust. | |

| **Query time getter** | |
| --- | --- |
| @property  def query\_time(self):  return self.\_query\_time | |
| This is a getter for the query time of the previous query. It works as a function, so can be accessed as object.query\_time, in the same way variables are accessed. However, the value of the query\_time cannot be changed from outside of the class, as it uses the private variable instead. | |
| **Encapsulation and data hiding**  This uses a getter method to get the query time, which is a private variable. It also ensures that it cannot be changed from outside of the class. This use of private variables, and the getter method is a form of encapsulation. The query time is kept as a private variable, as it should not be able to be changed from outside of the class, as it should only be used to time the query. There is not any need for this to be changed externally, so there is not a setter for the value. | |

#### Configuration Module (./backend/configuration.py)

This module addresses the ‘*There should be a simple way to configure the system*’ requirements. This is also extended past the scope of these requirements to include configuring the MySQL database connection, recommendation generation and configuration, search results, debugging, and aesthetic factors for the project. The MySQL configuration is essential for the database connection, so would be required, even if it was stored as a constant in the WSGI file. The hyperparameter must also be configurable, as it is data specific. However, the other recommendation values are unnecessary.

Extending beyond the requirements does not introduce any additional complexity, as the records in the configuration are accessed in the same way variables are. This brings the advantage of being able to configure the project completely, based upon taste, performance of the server, and the desired responsiveness of the webpage, without any additional effort during coding. These configurable parameters that are extensions can likely be given default values. All of the configurable values are given in the documented design section. The MySQL parameters that are not given default values must be specified in the configuration file. They cannot be given default values for security, but must be given to connect to the database.

To help make it simple to query and change the records, a clear, rigid syntax for the configuration file was developed and explained in the documented design section. This should make the file very readable, and easy to edit. An example configuration file is shown below, with the minimum amount of variables set for the project to work. The headers allow for clear grouping of the variables, with indentation making it clear which header they belong to. However, to make these files easier to understand, there needs to be clear error messages to specify the type of error, and the line the error has occurred on. This should make the configuration easier and simpler to do, as any errors can be easily corrected.

| **Configuration Errors** | | |
| --- | --- | --- |
| Issues with the configuration should be clear, easy to understand, and simple to debug | Met |  |
| class ConfigInvalidDataForType(Exception):  def \_\_init\_\_(self, datatype, value, line):  message = f"Error at line {line}: '{value}' is not a valid {datatype}"  super().\_\_init\_\_(message)  class ConfigInvalidDataTypeError(Exception):  def \_\_init\_\_(self, datatype, line):  message = f"Error at line {line}: '{datatype}' datatype is not known"  super().\_\_init\_\_(message)  class ConfigIndentationError(Exception):  def \_\_init\_\_(self, line):  message = f"Error at line {line}: Unexpected Indentation"  super().\_\_init\_\_(message)  class ConfigVariableNotFound(Exception):  def \_\_init\_\_(self, variable, file):  message = f"Configuration file '{file}' does not contain the variable '{variable}'"  super().\_\_init\_\_(message) | | |
| These are four custom exception classes. They are exceptions, because they are inherited from the Exception base class. Whenever they are raised, they will output the message specified within their constructors. This can then be used to explain what the error found was, and where it occurred, to help with debugging and simple configuration.  The ConfigInvalidDataForType exception is for when a value is given for a record, but the value cannot be converted to the datatype specified for that record. It takes the datatype, the value given, and the line number, and outputs where the error occurred. This should make it very easy to find the error within the configuration file.  The ConfigInvalidDataTypeError exception is for when a datatype is specified for a record that is not known. It takes the datatype that was given, and the line number that the mistake has occurred at. This makes it very easy to find and fix the issue, helping make the project configuration simple  The ConfigIndentationError error is for when there is an unexpected indent in the file. For example, a record indented as if it was under a header, but there not being a header present. This makes it easier to identify the issues, as it takes the line number, which will be especially useful in large configuration files.  The ConfigVariableNotFound error is similar to KeyError or NameError exceptions in Python. It is for when a configuration variable is attempted to be accessed from the configuration file, but is not in it. It outputs that the variable is not in the configuration file. It does not matter if it is not in the default configuration file either, as it is would prioritise the user defined one first, and the default one should be complete.  Specifying the line number of the error is a significant aid to finding issues within the file. It helps significantly when creating and maintaining the configuration file, which helps to achieve the simple system configuration objective. However, the project does not have many configurable options, so it is unlikely for the line count to be that beneficial. Its inclusion still may help, and especially if the project was expanded significantly, and used a very large configuration file, it would be extremely beneficial. For this reason, the line number is included within the error messages. | | |
| **Exception handling**   | raise ConfigInvalidDataForType("integer", "b", 10)  ConfigInvalidDataForType: Error at line 10: 'b' is not a valid integer | | --- | | raise ConfigInvalidDataTypeError("unknown-type", 10)  ConfigInvalidDataTypeError: Error at line 10: 'unknown-type' datatype is not known | | raise ConfigIndentationError(10)  ConfigIndentationError: Error at line 10: Unexpected Indentation | | raise ConfigVariableNotFound("default\_config.conf", "undefined\_var")  ConfigVariableNotFound: Configuration file 'undefined\_var' does not contain the variable 'default\_config.conf' |   When loading and interpreting the configuration file, any errors in it that are found should be taken, and the relevant exception thrown. Throwing exceptions should prevent the code executing further, and further issues happening because of missing or invalid configuration variables. It also gives a good way to explain the issue with the configuration to the user, and where the issue lies within the configuration file. This helps make it very easy to change and update the configuration files if there is an error with it. | | |
| **Issues with the configuration should be clear, easy to understand, and simple to debug**  This code helps meet this objective, as it creates four separate exceptions that can be thrown, which clearly outline the error that has occurred. When they are thrown, they clearly specify what has occurred, which should then allow for easy debugging of the code. They also specify the line that the error occurred, which, in combination with the error type, and any associated data at that line, it is very clear what the error was, and how it should be rectified. Therefore, this helps meet this objective. | | |

# 

| **Configuration object constructor** | |
| --- | --- |
| def \_\_init\_\_(self, filename, default\_conf\_filename=None):  self.\_filepath = os.path.join(  os.path.split(os.path.dirname(\_\_file\_\_))[0],  filename  )  # Needs to go back a directory, as this is in a folder that is  # further into the filesystem that the configuration files  if default\_conf\_filename is not None:  self.\_default\_filepath = os.path.join(  os.path.split(os.path.dirname(\_\_file\_\_))[0],  default\_conf\_filename  )  else:  self.\_default\_filepath = None  self.\_load() | |
| This is the constructor for the configuration class. It takes the name of the configuration file, and the name of the default configuration to use. The default configuration can be None, for when there is not a default provided. For this project, it is not necessary, but it allows for the module to be applied to other programs in the future, where a default is not needed.  It changes the file paths given to the parent directory. This is because the configuration file is within the backend folder, and the configuration files are stored in the project root folder. This is done using the os.path module in python. It gets the absolute directory of the python file, then uses the os.path.split to separate this. This gives the current folder name, and the absolute path to the parent directory of the current one as a tuple, therefore taking the first item in this, gives the absolute path to the parent directory. Finally, using the os.path.join method gives the absolute path of the file to be opened for the configuration. Examples of the results of each of these functions is given below.   | os.path.dirname(\_\_file\_\_) | /home/reuben/NEA/backend | | --- | --- | | os.path.split(  os.path.dirname(\_\_file\_\_)  ) | ('/home/reuben/NEA, 'backend') | | os.path.split(  os.path.dirname(\_\_file\_\_)[0]  ) | /home/reuben/NEA | | os.path.join(  os.path.split(  os.path.dirname(\_\_file\_\_)  )[0],  "project\_config.conf"  ) | /home/reuben/NEA/project\_config.conf |   This is done for the user-defined configuration file, and stored to the private filepath variable. It is a private variable, as once it has been set, it should not be changed. It also does not need to be accessed externally either, so does not require any getters and setters. It is also applied to the default configuration file, as it is also in the parent directory too, and is saved to the default\_filepath variable. If it was not provided, it is assigned to None instead, so it can be easily checked.  Finally, it calls the private load method to retrieve the data from the file’s, and create the dictionary for the values. This configuration dictionary is stored in the private variable file\_config. Calling this within the constructor means that as soon as the instance is created, the values in it can be used. The time it takes for the WSGI script to start does not need to be taken into consideration as it does not affect the user experience, it can be done when the object is created, which makes the configuration module simpler to use. | |
| **Encapsulation**  It uses private variables to store both the file paths for the default configuration and the configuration file. These should not be changed from outside of the class. They also are unlikely to be needed to be accessed either, so can be private, without any getter or setter methods. | |

# 

| **Converting value datatype** | |
| --- | --- |
| def \_cast\_to\_type(self, datatype, value, line\_num):  try:  if datatype == "int":  return int(value)  elif datatype == "str":  return value  elif datatype == "float":  return float(value)  elif datatype == "bin-str":  return value.encode("utf-8")  elif datatype == "bool":  if value == "true":  return True  elif value == "false":  return False  else:  raise ValueError  else:  raise ConfigInvalidDataTypeError(datatype, line\_num)  except ValueError:  raise ConfigInvalidDataForType(datatype, value, line\_num)  # Case statement to switch the datatype. Uses if-else as the switch  # statement cannot be used in the required python version. | |
| This method converts a data item taken from the configuration file, the specified data type, and the line number the record is within the configuration file. It uses an if-else selection statement to specify the data conversion required for the data. For example, ‘int’ requires the value to be cast to an integer, and ‘float’ requires a cast to a float. The data needs to be converted to the specific data type, as when read from the file they will be strings. This also means that the value parameter needs to take a string only.  Some special cases for the conversions exist however. Converting to an integer, or a floating point number is simple to do, as they are built-in functions in python. The other conversions are not this simple. When converting the value to binary (‘bin-str’), the string needs to be converted to binary. This uses the encode method, to convert each character to the binary value, as specified within the UTF-8 character set. This gives a binary value to be used.  The boolean data type needs an embedded selection statement. This is because the method in python to cast to a boolean, when given strings of ‘true’ and ‘false’, will always give True. If the data is ‘true’, it should return True, and if it is ‘false’, it needs to return False. If it is neither of these values, it is not a valid boolean, so raises a Value Error. This makes this behave like the built-in functions for data casting, so is simpler to handle later.  The string data type (‘str’) is a special case. As previously mentioned, when reading from a file, strings are returned, and therefore this method takes a string as the value parameter. Therefore, when the specified data type is a string, the inputted value does not need to be changed, so can be returned. This therefore means that the str() method does not need to be applied to the value before returning it.  If the specified data type does not match any of the clauses within the if-statement, it cannot be a valid data type for the configuration. This means that if the else runs, the ConfigInvalidDataTypeError is thrown, taking the line number passed in to the function originally, and the data type that was given, to show the error message to the user, to correct it easily.  The if-else statement does not handle any errors that occur, other than unknown data types. Both of the conversions for integers and floating point data types will throw a ValueError if it cannot be performed, such as trying to convert ‘a’ to an integer or a floating point. The boolean conversion throws this same error so they are consistent, and can all be handled the same. This approach is likely to be slower, but as the initial loading times of the project are not a major consideration, this method is more appropriate than handling booleans differently. The entire statement is enclosed in a try-catch, so if any of the options raise a ValueError, so cannot be converted, they can be all handled by the catch. This then raises the ConfigInvalidDataForType, which takes the originally passed line number, the value, and the data type given. This will help make it easier for the user to find and fix any errors with the configuration file.  A switch statement is would be more appropriate than this if-else. These are only available in Python since 3.10, and the Raspberry Pi used was limited to 3.9.2, it needed to use the if-statment approach. For completion, an approach using the structural pattern matching syntax introduced in 3.10. The \_ case is the equivalent of the else within the if-else.   | def \_cast\_to\_type(self, datatype, value, line\_num):  try:  match datatype:  case "int":  return int(value)  case "str":  return value  case "float":  return float(value)  case "bin-str":  return value.encode("utf-8")  case "bool":  if value == "true":  return True  elif value == "false":  return False  else:  raise ValueError  case \_:  raise ConfigInvalidDataTypeError(datatype, line\_num)  except ValueError:  raise ConfigInvalidDataForType(datatype, value, line\_num) | | --- | | |
| **Exception Handling**  This method uses exception handling. It catches any ValueErrors that were raised from the data conversion. These are then caught, and the appropriate ConfigInvalidDataForType error is raised instead to help the user debug and fix any issues that are within the configuration file. This helps make it simpler to use and debug, as the error messages specify the line number, and the error type. | |

| **Loading a configuration from the files (including default)** | | |
| --- | --- | --- |
| A default configuration should be used, which can be updated as needed based upon available hardware | Met |  |
| Configurable parameters should be modified through an external configuration file, which is in a clear an readable format | Met |  |
| Issues with the configuration should be clear, easy to understand, and simple to debug | Met |  |
| Parameters that can affect performance should be configurable | Met |  |
| def \_load(self):  with open(self.\_filepath, "r") as f: # Implicitly closes the file  # after opening. Opens as read only  contents = f.readlines()  if self.\_default\_filepath is not None:  with open(self.\_default\_filepath, "r") as f:  hierarchy = json.loads(f.readline()) # Load the json from  # the file and create dictionary with it.  # This is the default values. They will be overwritten as  # needed.  else:  hierarchy = {}  heading = "" # No header is represented as an empty string  for line\_num, line in enumerate(contents):  heading\_re = re.match("(\w+):\s\*", line) # Match headers in  # the line  if heading\_re: # Runs if there is a match to a heading  heading = heading\_re.group(1).lower() + " "  # .lower ensures case insensitivity. Adding space allows  # for just appending the value to the heading  else:  entry\_re = re.match(  "\t|\s{4}(\w+)\s+([\w-]+)\s\*:\s\*(.+)",  line  ) # Match indented records in line  if heading != "": # If it is, it should be indented  if entry\_re:  string = heading + entry\_re.group(1)  hierarchy[string.lower()] = self.\_cast\_to\_type(  entry\_re.group(2).lower(),  entry\_re.group(3),  line\_num + 1  ) # Change the datatype to the specified one  else: # There must be an unindented record, so it  # removes the header  heading = ""  if heading == "":  external\_re = re.match(  "(\w+)\s+([\w-]+)\s\*:\s\*(.+)",  line  ) # Match unindented records in line  if external\_re:  hierarchy[external\_re.group(1).lower()] = self.\_cast\_to\_type(  external\_re.group(2).lower(),  external\_re.group(3),  line\_num + 1 # Increments the line number by 1  # to give the line number of the exception  )  elif not (re.match("\s\*", line)): # If there is no  # header, and there is non-space characters on the  # line, it must be invalid indentation  raise ConfigIndentationError(line\_num + 1)  self.\_file\_config = hierarchy | | |
| This method loads a configuration from the specified configuration file, and the default configuration file, if specified. From this it then produces a dictionary, with the key being the query string, and the value being the same as the one specified in the configuration file, having been converted to the correct type.  It is important to note that there are three main ways that this configuration module could work. It could prioritise access speeds, load times, or have a hybrid between the two. The approach taken prioritises access speeds, completely preloading all of the values when the object is created. This will have much faster access times, at the cost of a long time to load the data. The second approach is to prioritise the initial load time. This would load the minimum amount of data, and would convert the query string when the data is accessed, and also cast the data at the same time. This would significantly increase access times, but reduce the initial time required to load it. The third approach that could be taken is a cross between the two. It does the minimum initially, like the second approach, and then whenever a request is made, it processes it, and over time builds up the dictionary found in the first using memoisation. This would offer a good tradeoff between load times and access times. The first approach is the one taken here, as the initial loading times do not need to be a consideration, and access times need to be as high as possible. Furthermore, the third approach, which would arguably be better, is significantly more complicated to develop, and would not offer any noticeable increase in performance, so is not been used;  It first opens the user-defined and editable configuration file, in read-only mode, reading all of the lines and storing them as a list. It then creates an empty dictionary for the hierarchy. If there was a default configuration JSON file specified, it then opens that, again in read-only mode, and reads the single line in the file. The json module is then used to convert the contents of the file into a usable dictionary. This dictionary is then saved as the hierarchy variable. This works, as if subsequently any values need to be changed, the syntax is the same as adding a new item into the dictionary. This means that this is a very quick and simple way to incorporate the default values, whilst prioritising the user defined ones.  Having created the hierarchy dictionary, with the default values, if applicable, it then loads the user defined configuration. Doing this second means that the user-defined values are prioritised above the default values. The header variable stores the most recent heading that was used within the file, as a string. If it is not applicable, it is represented by an empty string. This makes it simple and easy to check whether the next record should be placed under a header.  It iterates through every line in the configuration file, using Python’s enumerate iterator to give both the value at each line, and the iteration number. The iteration number is needed for the line number in any error messages that need to be given, for debugging the project. The line number is one larger than the iteration number, stored in line\_num, as enumerate indexes from 0, and the line number should be from 1.  For each line it carries out the same process. First it checks whether the string matches a header, using the regular expression given in the documented design. This is done using the ‘re’ module in Python. If the line matches the regular expression, the line must be a header, and therefore the header should be updated. A space is added after the header, so the record names can be concatenated directly to the end, and will not require the space to be added later. However, if it does not match a header, there are three possibilities that need to be checked. It is therefore an empty line, an indented record, or an empty line.  If the line is not a header, it checks whether the line is an indented record. Syntactically, it cannot be indented if there was not a header previously in the file, so first it checks whether there is a header set. If there is not a header set, it cannot be an indented record. However, if there is, it again uses the re.match method to check whether the line matches the intended record regular expression. If it does match, the first group is the variable name, the second group is the datatype, and the third is the value. These are then used to add a new item into the dictionary, generating the key by concatenating the variable name to the heading, which is the query string. It converts this to lowercase too, to ensure that it remains case insensitive. The private cast\_to\_type method to convert the given value into the given data type before storing it in the hierarchy dictionary. If the regular expression does not have a match, it should mark the heading as blank. This is because a blank line implies that the section has ended, and any subsequent values are for a different section.  Syntactically, it cannot be an unindented record if there is a header specified. Therefore it checks whether there is a left header. This cannot use an else if, and must be a separate statement, as if the header was removed in the last if statement, this one must be run. If this wasn’t performed, and the header was removed in the last selection statement, a line in the configuration would be missed.  First it checks whether the line matches the regular expression for an unindented item. If it does, it can be added to the dictionary. If it is unindented, it does not have a header, so the first group, converted to lowercase, can be used directly. The conversion to lowercase ensures that it can be case insensitive. It then uses the private case\_to\_type method to convert the value to the correct value, and stores it in the hierarchy dictionary. The second group is the datatype, which is not case sensitive, and the third is the value. As previously mentioned the line number is also given, which is the iteration number incremented by 1. This is given to help with any of the errors that could occur when converting the data type to the desired value. However, if it does not match this, and does not match the regular expression for an empty line, there must be unexpected characters on the line, which must be invalid. Consequently it throws the ConfigIndentationError, with the line number to specify where the error has occurred. It will only get to this if it does not match header, unindented or indented record, and a blank line. If it is a blank line it will move on to the next line. | | |
| **File Handling**  This method uses file handling to load both the default configuration and user-defined configuration files. This allows for it to be loaded from files, and not be hardcoded within the code. This helps make it faster to change, and edit the configuration. | | |
| **Exception Handling**  It uses exception handling to throw error messages, to help debug the configuration file. It makes it simpler and easier to change the configuration files to work correctly. It also ensures that the program will stop, and cannot run until the correct values have been entered, which ensures that further issues cannot occur because of incorrect values. | | |
| **Regular Expressions**  This method uses regular expressions to check the pattern that a string matches and find the values at specific locations within the string. Regular expressions are fairly expensive, but it is significantly simpler to develop and use than a custom implementation that can identify the sections. Furthermore, it allows for all the needed features for string matching for the project, through Python’s regular expression library, ‘re’, and given the initial loading times and performance are not a consideration, regular expressions are the most appropriate approach to use. | | |
| **A default configuration should be used, which can be updated as needed based upon available hardware**  This method meets this objective, as it allows a default configuration to be loaded from a JSON file, and then loading a custom configuration file over the top of it, prioritising the custom configuration file. This allows for a default configuration with reasonable values, which then can be updated based upon the performance of the available hardware. | | |
| **Configurable parameters should be modified through an external configuration file, which is in a clear an readable format**  This method meets this objective, as it loads a configuration file. It checks that the file follows a strict syntax, which was given in the documented design, to ensure readability and easy debugging. | | |
| **Issues with the configuration should be clear, easy to understand, and simple to debug**  This method meets this objective, as it throws exceptions when the configuration is invalid. It specifies the error with the configuration file, and the line number within the file that the error is at, so it can be easily debugged. | | |
| **Parameters that can affect performance should be configurable**  This method helps meet this objective, because it allows for any constant to be kept within the file. This therefore makes it easy to configure any aspect of the project through it, so all parts can be configurable. | | |

| **Get a configuration value** | | |
| --- | --- | --- |
| Parameters that can affect performance should be configurable | Met |  |
| def get(self, query\_string):  query\_string = query\_string.lower() # Prevent case sensitivity  if query\_string in self.\_file\_config.keys(): # Avoids needing the  # try-catch statement - is faster  return self.\_file\_config[query\_string]  else:  raise ConfigVariableNotFound(query\_string, self.\_filepath)  # Raises exception if it does not exist | | |
| This method gets a variable from the configuration file, including the default configuration. It first changes the query\_string to lowercase, to ensure that it is not case sensitive. It then checks that the query string is in the file\_config dictionary, and if it is, it returns the corresponding value. This works, as the file\_config keys are the query string, and therefore don't require any additional processing to generate. However, if it is not in the dictionary, the ConfigVariableNotFound error is thrown, as the variable does not exist.  This could use exception handling instead of an if statement, to check if the configuration exists. However, it is faster to check if it is in the dictionary, than using a try-except. This was tested using the timeit.timeit library, with 1,000,000 iterations, and using the try-except approach was roughly 0.5 seconds slower. While insignificant, using the selection was faster, which is why it was used.  This can be done almost as a dictionary lookup, as the values store the key to the value, which is the correct data type. The additional processing is required, so cannot be O(1), as just a dictionary lookup, however this is as efficient as it can likely be. This efficiency means it can also be used within the WSGI script, rather than assigning the values to constants, as it will not have an impact on the performance of the system. | | |
| **String Manipulation**  This method uses string manipulation. It converts the query string to being lowercase, before looking up with it. This, in combination with the variable names being converted to lowercase when the config dictionary is created, ensures that the variable names are case insensitive. | | |
| **Parameters that can affect performance should be configurable**  This method meets this objective, as it allows parameters to be queried from the file, and accessed like constants, without a significant performance overhead to do so. | | |

The default configuration file needs to contain a JSON object. The one used for this project is given below. For the Python module, json, to work, the JSON must only use a single line of this file, likely because it would include line breaks when read. This is not very readable, or easy to use. This could likely be fixed by reading one line at a time, and stripping the line break from the end of each one, however, this is not designed to be accessed or changed by the system user, readability is not a concern, so is not worth the additional complexity of adding this in. There are two copies of the JSON object below. The first is what would be stored in the file, and the second has been separated out into separate lines to make it easier to read. It contains the same default fields, and the same corresponding values as found in the documented design section.

| {"mysql username": "wsgi","mysql schema": "OpenBook","mysql host": "localhost","passwords hashing\_algorithm": "sha256","passwords number\_hash\_passes": 100000,"home number\_home\_summaries": 8,"home number\_about\_similarities": 10,"recommendations number\_converge\_iterations": 100,"recommendations hyperparameter": 0.1,"recommendations inital\_recommendation\_matrix\_value": 0.5,"recommendations reading\_list\_percentage\_increase": 0.5,"recommendations author\_following\_percentage\_increase": 0.5,"recommendations bad\_recommendations\_matrix\_value": 0.5,"recommendations minimum\_required\_reviews": 10,"recommendations number\_recommendations": 10,"search number\_results": 50,"session\_id\_length": 4,"debugging": false,"number\_display\_genres": 8} |
| --- |
| {  "mysql username": "wsgi",  "mysql schema": "OpenBook",  "mysql host": "localhost",  "passwords hashing\_algorithm": "sha256",  "passwords number\_hash\_passes": 100000,  "home number\_home\_summaries": 8,  "home number\_about\_similarities": 10,  "recommendations number\_converge\_iterations": 100,  "recommendations hyperparameter": 0.1,  "recommendations inital\_recommendation\_matrix\_value": 0.5,  "recommendations reading\_list\_percentage\_increase": 0.5,  "recommendations author\_following\_percentage\_increase": 0.5,  "recommendations bad\_recommendations\_matrix\_value": 0.5,  "recommendations minimum\_required\_reviews": 10,  "recommendations number\_recommendations": 10,  "search number\_results": 50,  "session\_id\_length": 4,  "debugging": false,  "number\_display\_genres": 8  } |

This meets the “*A default configuration should be used, which can be updated as needed based upon available hardware*” objective, as it gives default values for all of the configurable parameters that can be reasonably be given default values. Values like passwords need to be excluded to ensure that the project remains secure.

#### Searching algorithms module (./backend/searching\_algorithms.py)

The searching algorithms module was added to contain all the searching algorithms that were used in the project. It was created as a module over being hard coded where it was used, as it could then be used elsewhere, more easily. By the end of the project, it was only contained a binary search, but as this was still a logical decision to include it in the module, as it enables it to be packaged separately and be reused across multiple different files.

Like the mysql\_handler module, this does not meet any of the requirements. However, it is required for other parts of the project, which do meet the requirements, so are vital.

| **Binary search algorithm, which works with dictionaries and other data types** | |
| --- | --- |
| def binary\_search(arr, target, comparison\_func=None):  if not len(arr):  return None  if comparison\_func is None:  comparison\_func = lambda x: x  # geeksforgeeks.org/python-program-for-binary-search/  top = len(arr) - 1  bottom = 0  while bottom <= top:  mid = (top + bottom) // 2  if comparison\_func(arr[mid]) > target:  bottom = mid + 1  elif comparison\_func(arr[mid]) < target:  top = mid - 1  else:  return mid  return None | |
| This method performs a binary search on a given array. It is based upon the Python example given by (GeeksforGeeks, 2018), however it has an additional parameter for a comparison function. This function is executed every time to access the value that is being sorted by. This could be accessing a value from a dictionary, or averaging the values in a sub-array. The inclusion of this, over the (GeeksforGeeks, 2018) version, allows for it to search through lists containing any data type, including dictionaries. Other additions to the function include adding a check whether it is an empty list. The final difference is it returns None instead of -1 if it is not present in the given list. It is easier, and more logical to check that it exists using None over -1, as there is no valid index, so no value can be returned. Hence, None is a more appropriate return value than -1.  It first checks that the are items in the list that was passed. This prevents any further processing if it was empty, so should help it to be faster if the list is empty. If it was empty, the target item cannot be in the list. It then sets the top pointer to be the last item in the list, which is the length of the list, decremented by 1, as lists in Python are indexed from 0. It also sets the bottom pointer to 0. It also checks whether a comparison function has been given. If there was not one specified, it sets it to a lambda (anonymous) function, that returns the value that was passed to it. This allows for the comparison\_func to be applied, and it work correctly regardless of whether one was specified.  Following this it follows the same process as given by (GeeksforGeeks, 2018). It repeats until the bottom pointer is greater than the top pointer, each time calculating the midpoint. It then checks that the value at that location is greater than or less than the target, adjusting the pointers accordingly. It also checks whether it the midpoint value is the target value. The midpoint is the current index being checked, so if it is found, it is what os returned. When checking whether the target comes before or after the midpoint, the comparison\_func is applied to the data to give the value to compare. This allows for dictionaries or other data structures to be used within the search. | |

#### Data structures module (./backend/data\_structures.py)

The data structures module was created for varying abstract data structures to be used. A class for matrices and a vector subclass was initially created for this, to be used for generating recommendations. However, this had significant performance issues, running too slowly to be useable. Each iteration of the weighted alternating least squares method would take several minutes to process, and therefore numpy was used instead of the custom classes. Consequently, these classes have been excluded from the solution.

This module does not meet any of the requirements. However, it is used in some of the subsequent processes, which do then meet requirements. Consequently, this module is vital.

##### Queue

| **Queue errors** | |
| --- | --- |
| class QueueOverflowError(Exception):  def \_\_init\_\_(self):  super().\_\_init\_\_("Tried to push too many items into the queue")  class QueueUnderflowError(Exception):  def \_\_init\_\_(self):  super().\_\_init\_\_("Tried to get a value from an empty queue") | |
| These are two custom exception classes for queues. The queue underflow error should be used when a queue is popped from, or peeked, when there are not any items within it. The queue overflow error should be used when an item is pushed into a queue, which is full. These are standard behaviours and errors from queues, similar to those for stacks. | |
| **Exception Handling**  These exceptions are used as a clear way to indicate that the queue is either empty and full. They are then easy to catch using a try-except statement, or stop the execution of the program if they are not caught, preventing further errors to occur | |

The queue class follows the following class diagram. It also has a child class of a priority queue. The code for that class is found in following section. It uses the same methods, as the queue class, and overrides their behaviours to be a priority queue.

| **Queue** |
| --- |
| * items * max\_length |
| * push * pop * peek * get\_size |

#### 

| **Queue constructor** | |
| --- | --- |
| def \_\_init\_\_(self, max\_length=None):  self.\_items = []  self.\_max\_length = max\_length | |
| This is the queue class constructor. It creates a list, and stores it in the private items variable. This is used to store the data in the queue, and as it is a list, can hold any number of items. It also assigns the max\_length variable, which stores the maximum number of items that can be pushed into the queue before the QueueOverflowError should be thrown. It is the same as the optional parameter passed into the constructor, which defaults to None. A value of None for the maximum length indicates that the queue can contain any number of items, and the QueueOverflowError should never occur. | |
| **Encapsulation**  This uses private variables to store the maximum number of items that can be in the queue, and the actual data within it. These do not need to be accessed, especially the items list. Being able to access this would defeat the purpose of having the queue object as well, as data could be accessed outside of the FIFO (first in first out) structure. | |

| **Get number of items in the queue** | |
| --- | --- |
| @property  def size(self):  return len(self.\_items) | |
| This method gets the number of items in the queue. It gets this by calculating the length of the items list, which stores all of the entries. The use of the property decorator within the python means that it can be accessed like a variable (queue.size), and each time it is called, it calculates the length of the list. This prevents having to change a variable every time the queue is pushed or pulled. The use of the decorator, over a variable, is that it cannot be changed. This same effect could be done with a private variable, and a getter method, but this would be more complicated to use. | |

| **Add item to queue (push)** | |
| --- | --- |
| def push(self, item):  if (self.\_max\_length is not None  and self.size + 1 > self.\_max\_length):  # Checks if it is not None, and if so does not check second  # clause  raise QueueOverflowError  self.\_items.append(item) # Appends to end | |
| This method inserts a new item into the queue. It first checks whether a maximum length was specified, and if adding a new record would exceed the maximum length. The size of the queue must be increased by 1 before comparing it, as it needs to be checked after the record was added, not before. If it does exceed the maximum length, the QueueOverflowError needs to be thrown.  If there was not an error thrown, the value is added to the items list, at the end. The means that the first item in the list is the first item in the queue. | |
| **Exception Handling**  This method throws the QueueOverflowError exception to ensure that the program execution ends if the queue is filled. It also allows a clear way to tell if the queue is full, and can be caught using a try-except clause. | |

| **Remove item from queue (pop)** | |
| --- | --- |
| def pop(self):  if not self.size:  raise QueueUnderflowError  return self.\_items.pop(0) # List is reverse order - FILO | |
| This method removes the first item from the queue. It first checks if there are items in the queue, as if there are not, items cannot be removed from the queue. Consequently, the QueueUnderflowError is thrown. If there are items in the list, the first should be removed, as this is the first item pushed into the list. When removing the item from the list using the pop method, the value is returned, so this can be removed and returned on the same line. | |
| **Exception handling**  This method throws the QueueUnderflowError when there is an attempt to pop from the queue when it is empty. This prevents any errors from occurring when removing items from an empty list, and is also clear as to the error that occurred. This makes it easy to catch and handle the errors, as the error can clearly be specified, without being ambiguous. | |

| **Access first item from queue (peek)** | |
| --- | --- |
| def peek(self):  if not self.size:  raise QueueUnderflowError  return self.\_items[0] | |
| This method gets the first item in the queue, without removing it from the queue. This works in the same way as the pop method does, but instead of using the pop() method on the list, it accesses the first item in it, and returns that. This gives the value, without removing it from the list. | |

###### Priority queue

The priority queue is derived from the queue subclass, overriding the constructor, push, pop, and peek functions. This is because this implementation uses a list of tuples, with the priority and item in it, and also requires a different process to add and remove items from it because of this list. This class is defined using the following class diagram, inherited from the Queue class. The size method (property) is inherited from the Queue superclass, so does not need to be changed, or included in the class.

| **PriorityQueue (Queue)** |
| --- |
| * priority\_func |
| * push (overridden) * pop (overridden) * peek (overridden) |

#### 

| **Object constructor** | |
| --- | --- |
| def \_\_init\_\_(self, priority\_func=None, max\_length=None):  super().\_\_init\_\_(max\_length=max\_length)  if priority\_func is None:  self.\_priority\_func = lambda x: x  else:  self.\_priority\_func = priority\_func | |
| This is the constructor for a priority queue. It is inherited from the Queue class, so calls the same constructor for that class. However, it also sets a priority function, which allows for different structures to be used within the queue, such as dictionaries or lists. This parameter defaults to None, and if this is the case, the private priority\_func variable is set to a lambda variable that just returns the value that it is given, otherwise it is set to the function that is passed into the class. | |
| **Inheritance**  The priority queue class is inherited from the Queue class. This enables the old constructor to be reused, with additional values passed into it. This will also allow for methods and properties from the Queue class to be reused, or modified to work with the different structure of this class. | |
| **Overriding**  This uses overriding to assign new variables within the class, and take different parameters to the Queue superclass. However, it still uses the original constructor, and performs more subsequent processes to assign the priority function variable. | |

| **Add item to queue (push)** | |
| --- | --- |
| def push(self, item, priority=None):  if (self.\_max\_length is not None  and self.size + 1 > self.\_max\_length):  # Checks if it is not None, and if so does not check second  # clause  raise QueueOverflowError  if priority is None:  priority = self.\_priority\_func(item)    self.\_items.append((item, priority))  self.\_items.sort(key=lambda x: x[1], reverse=True) | |
| This method adds a new item into the queue. It first checks whether adding a new item would exceed the maximum size of the queue, if one was specified. If it would exceed it, it throws the QueueOverflowError. Checking this before adding it, while not as simple, prevents additional steps being made if they are not necessary.  This method takes the priority, as an optional parameter. This allows it to be specified when an item is pushed into the queue. If it is not specified, it defaults to None, which is easy to check for. If it was not given, it performs the applies the priority function to the item given to calculate the priority to be used. This allows for the priority to be a single item within in a dictionary, list, or other datastructure. If a function was not specified, it will set the priority to being the same as the one that is being pushed into the queue.  It then appends a tuple to the items list, containing the value, and the priority it has. A tuple is used as they are immutable, and also tend to be faster than lists. Having sorted it, it sorts the list of tuples, by the priority descending. It needs to be descending so that the highest priority items are at the front of the list. This sort does still retain the order of insertions, so within each priority level it remains FIFO. The sort method on the list sorts it inplace, so it does not need to be reassigned to the variable after this has been performed. | |
| **Overriding**  This method overrides the one from the Queue superclass. This is because it needs to insert both the value, and the priority, as well as making sure it is in the correct location. The superclass method just needs to add the value at the end of the items list. | |

| **Remove item from queue (pop)** | |
| --- | --- |
| def pop(self):  return super().pop()[0] # The result from the super would be a  # list, where the first item is the inserted value and the second  # is the priority | |
| This method removes the first item from the queue. It uses the pop method from the superclass, therefore uses inheritance. This therefore will include the check, and raising of the QueueUnderflowError exception if it is needed. The behaviour of the pop method needs to be overridden, however. This is because the items list in the PriorityQueue class is a list opf tuples, with the tuples having the value in the first index, and the priority in the second, but the items list in the Queue class is just one-dimensional. This means that when the super().pop() method is called, calling the pop method from the Queue superclass, a tuple will be returned, containing the value, and its priority in that order. Therefore, the pop method for the PriorityQueue takes the first item in the returned tuple. | |
| **Exception handling**  As in the Queue class, this method throws the QueueUnderflowError when there is an attempt to pop from the queue when it is empty. | |
| **Inheritance**  This method uses the inherited pop method from the superclass, as it requires the same process, just taking the first item in the result instead. | |
| **Overriding**  This method can be written in a similar way to the superclass, but if it worked the same way, it would give the wrong result. This uses the original method, and modifies it so it works correctly. | |

| **Access first item from queue (peek)** | |
| --- | --- |
| def peek(self):  return super().peek()[0] | |
| This method gets the first item in the queue, without removing it from the queue. It uses the method from the Queue superclass. However, because the items list in the superclass is a simple one-dimensional array, when applied in this class, with the items list being a list of tuples, will return a tuple, with the first item being the value and the second being the priority. This is why the first value of the result is taken. Calling the super method also checks whether the QueueOverflowError should be thrown.. | |
| **Exception Handling**  As in the method in the Queue class, it throws the QueueUnderflowError when there is an attempt to view the first item in the queue when it is empty. | |
| **Inheritance**  This method uses inheritance. It does this because it uses the peek method from the superclass, which was inherited, as the process required is almost identical. | |
| **Overriding**  Even though this method is almost identical to that in the superclass, it needs to take the first item in the resulting tuple. This means that it works differently, and needs to be changed from the superclass, hence why it is overridden. | |

##### Stack

| **Stack errors** | |
| --- | --- |
| class StackOverflowError(Exception):  def \_\_init\_\_(self):  super().\_\_init\_\_("Tried to push too many items into the stack")  class StackUnderflowError(Exception):  def \_\_init\_\_(self):  super().\_\_init\_\_("Tried to get a value from an empty stack") | |
| There are two custom exception classes for queues. The stack underflow error should be when there is an attempt to pop or peek a queue that has not got any values in it. The stack overflow error should be used when an item is pushed into a queue, which has reached its limit. These are standard behaviours and errors for stacks.  The super().\_\_init\_\_(string) within the class constructor uses the Exception superclass’s constructor. When the error is thrown, the string passed into it is outputted to the console, or wherever the file’s error messages are sent. This makes it clear what the error was, and why it was raised. | |
| **Exception Handling**  These exceptions are a very clear way of indicating that the stack is either empty or full. They can be easily caught using a try-catach statement, and if they are not caught, they will prevent the program from executing further, which will prevent any errors to occur later because of it. | |

The stack class follows the following class diagram. It has the same methods as the Queue class, however, is not inherited from it. This is because the way a stack works is fundamentally the opposite as a queue. A queue has a First In First Out (FILO) structure, whilst a stack has a First In Last Out (FILO) structure, and therefore they cannot be related.

| **Stack** |
| --- |
| * items * max\_length |
| * push * pop * peek * get\_size |

| **Stack constructor** | |
| --- | --- |
| def \_\_init\_\_(self, max\_length=None):  self.\_items = []  self.\_max\_length = max\_length | |
| This is the constructor for the stack class, and it works in the same way as the Queue class. It creates an empty list, stored in the private items variable, which stores all the data inside the stack. It also assigns the max\_length variable to the passed parameter, which specifies the maximum number of items that can be stored within the stack. It defaults to None, which indicates that there is not a limit on the number of items that can be pushed into the stack. If this value is exceeded, the StackOverflowError should be thrown. | |
| **Encapsulation**  This uses private variables to store the maximum number of items that can be in the queue, and the actual data within it. This data does not need to be accessed from outside the class, especially the items list, as that voids the purpose of the stack, if the values can be accessed without using the FILO (first in last out) structure of a stack. | |

#### 

| **Get number of items in the stack** | |
| --- | --- |
| @property  def size(self):  return len(self.\_items) | |
| This method gets the number of items in the stack. It works in the same was as the getter for the queue class. It uses the property decorator to make the function appear like a variable, and each time it is accessed, it calculates the size. It also prevents it from being changed, but with a simpler syntax than a getter method. Calculating the length over having a private variable with a getter method, is more likely to be faster, and more memory efficient, as it does not need to be changed every time a value is pushed or popped from the stack. It also reduces the number of variables, and therefore the amount of data being stored, so will be more memory efficient | |
| **Encapsulation**  This uses encapsulation, as it uses a method to get a value from a private variable. It is not truly a getter method, as it is not directly getting the value of a private variable, rather calculating the value from a private variable. It does not use the syntax of a standard getter method, but rather a public variable, which cannot be changed because of the property decorator not having a corresponding setter method. | |

#### 

| **Add item to the stack (push)** | |
| --- | --- |
| def push(self, item):  if (self.\_max\_length is not None  and self.size + 1 > self.\_max\_length):  raise StackOverflowError  self.\_items.append(item) | |
| This method adds a new item to the stack. It first checks whether adding the new item to the stack would exceed the maximum size of the stack, assuming that the maximum length is specified. This is done by checking whether the max\_length variable is None, indicating that the maximum length was not specified. If the maximum length is specified, and the addition of the new item would mean it exceeds this limit, it raises the StackOverflowError.  If it does not exceed the limit, it adds the new item to the end of the items list. This means that the first item within the list is the last to leave it, so when removing items, or peeking items, it should be the last item within the list that should be accessed. | |
| **Exception Handling**  This method throws the StackOverflowError exception if the stack is full. This ensures that the program will stop executing if the stack is filled, or offers a very clear and logical way to catch the error using a try-except statement. | |

#### 

| **Remove item from the stack (pop)** | |
| --- | --- |
| def pop(self):  if not self.size:  raise StackUnderflowError  return self.\_items.pop(-1) | |
| This method removes the first item from the stack. It first checks whether there are items in the stack. If there are not any items in the stack, the StackUnderflowError is thrown, as it is not possible to get the item from it. If there are items in the stack, it uses the list method pop() to remove the last item in the list. This method also returns the value, so it can be directly removed, and returned on the sameline. | |
| **Exception Handling**  This method throws the StackUnderflowError exception if the stack is empty. This prevents issues with removing values from the empty list, and is clear and simple to catch if the error occurs, using a try-except statement | |

#### 

| **Access first item from the stack (peek)** | |
| --- | --- |
| def peek(self):  return self.\_items[-1] | |
| This method gets the first item in the stack, which will be the last item in the list. It works by returning the last item from the items list, using -1 to access the last value. | |

##### Binary Tree

This binary tree class is from (Tutorials Point, 2023). Parts have been excluded if they were not necessary for the project, including some different traversals. The in order traversal is the only traversal needed for this project, so the others have not been included. As this is not original code, it is not explained here. However, it is important to note that the in order traversal method was altered slightly to be easier and simpler to use. The version given by (Tutorials Point, 2023) requires the root to be passed into the function when it was called. It was altered so it has a default value, which is then replaced with itself, so that it does not need to be passed to work recursively.

An access\_function was also added, so that it works with lists, dictionaries, tuples, and other data structures. If one isn’t specified in the constructor, it is set to a function that returns the value that was passed into it. This makes it easy to use, regardless of whether a function was given or not. It then is applied to the value every time a comparison is made to find where the value should be inserted into the tree. This works in a similar way to the function for stacks and queues. The entire code for the binary tree is shown below.

| class BinaryTree:  def \_\_init\_\_(self, value=None, access\_function=None):  self.left = self.right = None  self.value = value  if access\_function is None:  self.access\_function = lambda x: x  else:  self.access\_function = access\_function  def insert(self, value):  if self.value is None:  self.value = value  else:  if self.access\_function(value) < self.access\_function(self.value):  if self.left:  self.left.insert(value)  else:  self.left = BinaryTree(value, self.access\_function)  else:  if self.right:  self.right.insert(value)  else:  self.right = BinaryTree(value, self.access\_function)  def in\_order\_traversal(self, root=""):  if root == "": # Cannot be None, and an empty string cannot be  # used.  root = self  res = []  if root is not None:  res = self.in\_order\_traversal(root.left)  res.append(root.value)  res = res + self.in\_order\_traversal(root.right)  return res |
| --- |

#### Environ variable manipulation module (./backend/envrion\_manipulation.py)

The environ manipulation module is a module for manipulating the environ variable that is given to the WSGI script with any request. It contains all of the data regarding the request, including post data, and the request URI. Within the python, it is given as a dictionary. This module focusses on manipulating this, and extracting data from it.

It was to be broken down into different classes, or collections, each containing a group of related functions and methods for manipulating the environ variable in a specific way. Each collection would hold semantically linked functions. However, it became clear that only functions for extracting data for specifying the applications/handler, and the function within them was necessary, which is why there is only the application collection. It was left as a collection, rather than just having functions within the module, as it allows for future expansion easily, whilst not requiring a rewrite of both the WSGI file that uses it, and the module, to ensure it remains logical.

| **Get target application/handler** | |
| --- | --- |
| def get\_target(environ):  path = environ["REQUEST\_URI"]  temp = re.match("/[\w-]+/([\w-]+)", path) # Should not include  # dashes in result, but included, so it does not break if it does.  if temp:  return temp.group(1)  return None | |
| This method gets the handler that a request is targeting, The handler name forms the second part of the request URI. For example, /cgi-bin/recommendations/get\_recommendations would be directed to the recommend\ations handler. The regular expression matches any number of non-whitespace characters, followed by a forward slash, and then any number of whitespace characters again. This second one is in brackets, and is therefore extracted as a group.  If there are any matches to the regular expression, it takes the string that is in the group, and returns it. This is then the specified handler that should be used to handle the request, and this is returned as a string. However, if there are no matches, it returns None, as there is not a specified target. However, if this happens, it is likely to be an invalid request, and so will give a 404 Error. | |
| **Regular Expressions**  This method uses regular expressions to check a string matches a specific pattern, and also separate, and get the value of a part of that string. While regular expressions are fairly expensive, using Python’s regular expression module, ‘re’, they are fast enough, and also very easy to use. | |

| **Get target function/method at the specified handler** | |
| --- | --- |
| def get\_sub\_target(environ):  path = environ["REQUEST\_URI"]  temp = re.match("/[\w-]+/[\w-]+/([\w-]+)", path) # Should not  # include dashes in result, but included, so it does not break if  # it does.  if temp:  return temp.group(1)  return None | |
| This method gets the process that needs to be carried out from the URI. It does this using a similar regular expression to the previous method, matching any number of non-whitespace characters, followed by a forward slash, followed by any number of non-whitespace characters, followed by a forward slash, then any number of non-whitespace characters. The final set of characters after the slash form the group, which contains the process name.  If there is a match for the regular expression, the value of the group is returned, If there are not any matches, None is returned. In this case, it is likely to be an incorrect URI, and therefore will send a 404 Page Not Found error. | |
| **Regular expressions**  Like the previous method, this uses regular expressions, as it is a simple way to check that the URI follows the specific format, and also extract the target function name from the string too. | |

#### Logging module (./backend/logger.py)

The logging module is a way of outputting debug messages and other data to a file. This is required, as when the web server is running, there is not any way of outputting to a console, which makes tracking requests through the server, and checking how they are being processed very difficult. The logging module therefore writes to a file, which can then be viewed, and used as a output log. Furthermore, the web server does not have any debugging software, which can trace through functions and stop and view the contents of variables. An easy way around this is to output variables to a file, which makes this logging module more important. It should be able to be disabled, however, as using it will have a performance overhead. Consequently, there should be a debugging value in the configuration file that defaults to false, to prevent it being used, unless it is required.

The logging module follows the following class diagram. Note that this was not included in the documented design, as it is not essential the project, and is a utility to help with development. It is created as a class, as it needs to store data about the files, including the file path, and the line length, as opposed to being a file of separate functions.

This model does not meet any of the requirements. It is solely to help with development, and debugging of the website.

| **Logging** |
| --- |
| * debugging * line\_length * filepath * clear |
| * open * write * output\_message |

| **Logging object constructor** | |
| --- | --- |
| def \_\_init\_\_(self, debugging=False, filepath="/tmp/", clear=True, line\_length=150):  self.\_debugging = debugging  self.\_line\_length = line\_length  self.\_filepath = filepath  self.\_clear = clear  self.\_open() | |
| This is the constructor for the Logging class. It takes a boolean value to state whether the program is being debugged, the filepath for the log file to be saved in, a boolean value to state whether the file should be cleared each time, and an integer for the number of characters to be added to a line. These are then stored as private variables. They are kept private, as once the object has been created there is not any need for the values to be altered, or viewed. Each of the parameters has a default value, with the debugging defaulting to False, as being true will have a noticeable performance impact. The filepath defaults to /tmp/ as it will be cleared every time the computer restarts. The clear parameter defaults to true, as unless old logs are needed, it will be more accessible to have a new log every time the server restarts. Finally, the line length defaults to 150, as it is long enough to be comfortably displayed on most monitors, and also gives enough characters for the file to be read easily.  Having assigned these variables, it then opens the file initially, wiping the contents of the file if needed, and adding the new session created lines. This must be performed after assigning the variables, as they are needed for the private open method to be called. | |
| **Encapsulation**  This method uses private variables to store whether the logging file is in the debug mode, the filepath of the log file, whether it should be cleared, and the line length. These do not need to be accessed, as they are contained within the logging object, the purpose of which is to handle and abstract the details of maintaining the log. They do not need to be accessed, as it is not designed to be used with any other processing, only as a way to track the execution of the program. The values also do not need to be changed, as any changes would require an entire new log to be created. | |

| **Open the output file** | |
| --- | --- |
| def \_open(self):  if self.\_debugging:  if self.\_clear:  self.\_method = "w+"  start = ""  else:  self.\_method = "a+"  start = "\n\n"  now = datetime.datetime.now()  start += ("-" \* self.\_line\_length) + "\nNew session created: "  start += now.strftime("%d-%m-%Y %H:%M:%S") + "\n" + ("-" \* self.\_line\_length) + "\n"  self.\_write(start)  self.\_method = "a+" | |
| This handles the initial opening and creation of the log file. It first checks whether the debugging flag is set to true. If it is not true, then the log should not be created, which should help improve performance. If it is true, however, the log should be created.  The first check is to see whether the log file should be cleared for the new session. If it is needs to be cleared, it sets the write method to “w+”, which overwrites the contents of the file. It also then sets the start of the file to an empty string, so it does not have any additional lines at the start of the file. However, if it should not be cleared, it sets the write method to “a+”, which appends to the file, and also sets the start of the addition to the log file as two new lines, which ensures that the different sections are then separated.    It then adds a row of dashes, then a line stating that a new sesion was created, and the time in “dd-mm-yyyy hh:mm:ss” form. This is the closed by another row of dashes. This is added to the start string that was created, then written to the file. This means that if the file was cleaned, this is the top of the file, and if was not, it is written after two blank lines from the end of the previous log. This is then written to the file, and then the method of writing to the file is changed to append to the file, so the subsequent logs are added to the end of the file. The write method is used, as it handles handles opening and closing the file, so it can be reused. | |
| This does have an issue, which cannot be navigated using the approach that was taken here. The webserver uses multiple WSGI scripts at the same time, each running the same code. This means that the logging module is loaded the same number of times. This also means that whenever the clear option is set to false, the header will be written the same number of times as worker threads created by the webserver. This is not a big issue, as once all of them have been created, they all output to the same file, so no debug messages are missed. When it is cleared, this is not such a big issue. The same effect is there, but each one will clear the file, so there will only be one header created, by which ever worker script is the last to start. This is not a large enough issue to rewrite how this works to avoid this. It is designed to be a debugging tool, so does not really need to be user serviceable, and as it defaults to wiping the file anyway, this is not an issue.  This could be improved if the number of workers that the webserver was known, and could be assumed to be static, by having a file with the creation number stored in it, and the file wouldnt be written to until the count in the file has reached the same number as the number of worker threads. This is not a good solution as the number of worker threads can and should be configured based upon the hardware, and the usage of the website. Furthermore, it would be significantly more computationally expensive, and for a minor feature that is not intended to be used by end users, it is not worth the impact. | |

| **Write data to the file a string to the file** | |
| --- | --- |
| def \_write(self, message):  if type(message) == list:  message = f"[{','.join(message)}]"  elif type(message) == dict:  message = json.dumps(message)  elif type(message) != str:  message = str(message) # Make it easier to write too – and  # faster if it is not in debugging mode as conversions do not  # need to be made  with open(self.\_filepath + "output.log", self.\_method) as f:  f.write(message) | |
| This method writes a message to the log file. It uses an if-statment to convert all datatypes given to it to a string, so that they can be written to the file. It converts a list to a string, so that it can be written, and is clearly a list, and converts dictionaries to a string using the JSON dumps method from Python’s JSON library. Everything else can be converted directly into a string. This allows variables to be directly written to the file, without needing to be converted to a string before being used. This helps make the module very simple and easy to use, through abstraction.  Following the conversion of the datatype, it then writes it to the log file. This also shows why the method was made a variable belonging to the object, rather than within the open method’s scope, as it allows for it to be used to overwrite the file, as well as appending to it.  The with statement also closes the file once it has been used. It has to be opened and closed each time, rather than opening when the object was created and closing it when it is deleted, as this would cause issues with the multiple worker scripts. It also would prevent the changes appearing when using the ‘tail -f’ command in the CLI, which then defeats the purpose of this as it could not be used to track variables, and requests while the server is being used. | |
| **File Handling**  This method uses file handling. It opens a file, and writes strings to it, so that variables and other messages can be accessed and viewed during the execution of the program. | |

| **Output a message to the file** | |
| --- | --- |
| def output\_message(self, message):  if self.\_debugging:  message = str(message)  now = datetime.datetime.now()  string = "[" + now.strftime("%d-%m-%Y %H:%M:%S") + "] "  length = len(string)  new\_message = [message[i:i + (self.\_line\_length - length)] for i in range(0, len(message), self.\_line\_length - length)]  string += ("\n" + " " \* length).join(new\_message) + "\n"  self.\_write(string) | |
| This method is the one used when a message needs to be written to the log file. It first checks whether the log file is in the debugging mode, as if it is not, then there should not be any data written to the file. This is needed as having unnecessarily writing to the file will have a performance impact.  If it is not debugging, it converts the given value to a string. It then calculates the current time, putting it in square brackets at the beginning of the line, to show the time that the log was made. This would help link it to any requests made from a client. It then splits the message into separate lines, using a list comprehension, splitting them up at a specific interval. This interval is the line length, specified when the object was created, with the length of the time subtracted from it. The list produced by this list comprehension is then taken and using the .join method is reassembled into a string. Joining each list element with a new line, and the same number of spaces as taken up by the date-time part, and adding this result to the string, creates an output that has the time, then the text indented, and also keeping the date-time part separate, and clear to read. This can then be written to the file, using the private write method, avoiding the need to handle writing to the file here. It is important to note that an additional “\n” needs to be added to the end of the to ensure that the messages are well spaced, and easy to read. An example output is shown below | |
| This does not work perfectly. When there are carriage returns (“\n”) or other special characters, it breaks and the line spacing is lost. Furthermore, if the window that the log is being viewed on is less than the number of characters being viewed, the spacing can become incorrect. This cannot be easily corrected, as there is not an easy way to have an awareness of a terminal size, especially as the file is being accessed as a file, rather than through a script. The easiest way around this would be to have a script that can be used to view it, and ensure that it is always the correct spacing. However, it is not worth the additional complexity, as it does not need to be viewed by the users, and is still good enough to be used for debugging. The first image below gives what happens with new lines added into the image, and the second is for when the window is smaller than the line length.      These errors are not worth the complexity of fixing, and some are not possible to fix without creating an additional program to view them in. This is not worth it for a debug log, which can be read easily enough even if these errors do occur. | |
| **List Comprehension and string manipulation**  This method uses list comprehension to split the given string up into individual lines, also using slicing to specify the points of the string on each line. This is then joined to generate the line, using further string manipulation to generate a string of the required number of spaces. | |

#### Machine learning utilities module (./backend/ml\_utilities.py)

This module includes useful functions that will be needed for the machine learning. It contains different types of measures for similarity and errors. These are used for the recommendations, and other operations, like finding similar books. Cosine similarity is another metric used within this project. However, it was not included in the module as it is often done using parts of data, or using data from abnormal formats, so it was simpler and easier to develop bespoke methods for those situations.

This module does not needed to meet any of the requirements directly. It is used by other methods and functions within other parts of the project to help meet the requirements, so is still an essential part of the project.

| **Jaccard Similarity** | |
| --- | --- |
| def jaccard\_similarity(set\_1, set\_2):  # set\_1 and set\_2 must be sets not lists.  # Sets are faster for union+intersection etc, as they are unordered and  # cannot have duplicate values.  union = set\_1.union(set\_2)  intersection = set\_1.intersection(set\_2)  return len(intersection) / len(union) | |
| This method calculates the similarity between two sets using jaccard similarity. It takes sets as they can be used for intersection and union very easily, and also quickly. It finds the number of items in set 1 or set 2 (intersection), by the number of items in set 1 and set 2 (union). This is the jaccard similarity between the two sets. It is very simple to calculate, especially as in Python sets have methods for intersection and union built into them. | |

| **Mean Squared Error** | |
| --- | --- |
| def mean\_squared\_error(true, predicted):  return sum((i - k)\*\*2 for i, k in zip(true, predicted)) / len(true) | |
| This method calculates the mean squared error between two one dimensional lists. It is designed to work with lists, but will work with any iterable object, like dictionaries or tuples. It iterates through both the of the lists passed into it, using the zip function so they can be iterated through at the same time. For each of these pairs it calculates the difference between the two, and squares it, and also sums them together. This sum is done using a generator, which tends to be faster than a for loop, as well as being more memory efficient. Finally it divides through by the number of items in the list, which gives it as a mean, rather the sum of the squares.  It is important to note that this was coded for use with the recommendations WALS. However, having experienced significant performance issues with this, it was replaced with the mean squared error for scikit learn instead. However, this is still included, as, unlike the custom Matrix class, should be able to be replaced directly, and still work correctly. This is because the numpy array object can be iterated through in the same way a list can. The numpy arrays, however, use a completely different syntax to the custom matrix class that was developed, so cannot be swapped out, without a complete recode of the recommendations model. | |

### Components Module

The components module is for all of the individual components of the website backend. This includes the accounts, authors, books, diaries, genres, information retrieval, reading lists and recommendations. Each of these can be seen as a component of the website, and as they are not utilities, they can be grouped into a single module. This modularisation is logical, but also has another advantage over having them in files with the other utility modules. This had a problem, as they each contain a single object, which when instantiated would logical ybe called the same thing as when the file was imported. This led to name clashes, especially when there were functions also within the same file. Each file had to be imported under a different name, such as authors\_mod, to avoid this, which was clunky. Moving them to a single module avoided this, as each can be accessed as components.authors. This means that the object created from the file can be called authors, for example, and not clash with the module name either.

The \_\_init\_\_.py file is required to make the folder a module. It allows for import components to be used, and all of the relevant files to be made accessible. It is not a complex file, but is essential, and is given below.

| from .accounts import \*  from .authors import \*  from .books import \*  from .diaries import \*  from .genres import \*  from .information\_retrieval import \*  from .recommendations import \*  from .reading\_lists import \* |
| --- |

#### Accounts (./backend/components/accounts.py)

The accounts part of the components module has functions and processes to handle all of the related processes to accounts. This includes creating accounts, and checking login credentials, as well as generating and checking session IDs. The module includes two different classes, an accounts class, which handles creating accounts, and managing them, and a sessions class, which handles creating and managing session IDs.

They are made as classes as they hold related data, as well as a database connection object. They hold the data instead of passing it in each time, as it is more convenient to use. It may also be faster, as queries from the configuration file, whilst as fast as possible, will be slower than accessing a variable. They follow the following class diagrams, which was excluded in the documented design, as it is part of the construction of the module.

| **Accounts** |  | **Sessions** |
| --- | --- | --- |
| * hashing\_algorithm * hashing\_salt * number\_hash\_passes * connection (object) * reading\_lists (object) |  | * connection (object) * token\_size |
| * hash\_password * check\_credentials * create\_user * get\_user\_id * get\_user\_id\_list |  | * create\_session * update\_time * get\_user\_id * close * get\_sesson\_id\_list |

##### Accounts class

| **Exceptions** | |
| --- | --- |
| class UserExistsError(Exception):  def \_\_init\_\_(self, username):  message = f"User already exists with the username {username}."  super().\_\_init\_\_(message)  class InvalidUserCredentialsError(Exception):  def \_\_init\_\_(self, username):  message = f"Incorrect username or password entered for {username}"  super().\_\_init\_\_(message) | |
| Custom exceptions allow for more detailed exceptions to be thrown. This makes the errors easier to identify, and to catch, over repurposing standard Python exceptions. It also is then more readable when reading code, as the exception that is being handled is descriptive and therefore clear.  The UserExistsError is for when there is an account attempted to be created, with a username that is already in use. If this were to be allowed, it would mean that usernames would not be unique, and therefore could not be used to identify user IDs directly. This error should be thrown in this case. Being a custom exception, it is also very clear the error that has been thrown, and therefore very easy to catch, and relay the exception to the client.  The InvalidUserCredentialsError is used when checking provided login credentials. It should be thrown if the credentials given are not valid. This includes both situations where the username or password is not valid. A lack of specificity here is not an issue, as the client should only know that either their username or password is wrong, not which one is wrong, as this could help work out user’s passwords.  Both of these Exceptions are inherited from the Exception base class. This allows them to be thrown as exceptions. Furthermore, they each have a message string, which is passed into the Exception super class. This is then displayed in the console, or wherever any error messages are piped to. This makes it clear what has occurred, as beyond the error name, it gives a brief overview of what has happened. This includes giving the username that was used, either for the existing user, or the username entered for credentials checking. | |
| **Error handling**  These can be thrown by the different functions to prevent further execution. It also gives a clear output from any methods that need them, which can be then caught and handled separately. These can be checked more clearly using a try-catch statement than checking a return value, or using a built-in Python exception. This helps improve readability, and should also be more robust. | |

| **Accounts class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, hashing\_algorithm, hashing\_salt, number\_hash\_passes, reading\_lists):  self.\_hashing\_algorithm = hashing\_algorithm  self.\_hashing\_salt = hashing\_salt  self.\_number\_hash\_passes = number\_hash\_passes  self.\_connection = connection  self.\_reading\_lists = reading\_lists | |
| This is the constructor for the accounts class. It takes a database connection object, from the mysql\_handler module, the hashing algorithm type, and the hashing salt, in binary, the number of hash passes to make, and a reading lists object, from the reading\_lists part of the components module. These are stored within private variables, as they do not need to be accessed or used externally. They are also taken as parameters here, instead of when needed, as it is much simpler and easier to use any of the subsequent functions if these parameters do not need to be passed in.  It uses a database connection object as a parameter, so that it can query the database. It takes it as a parameter, rather than creating its own instance, as some databases have a limited number of users, especially if they use the same credentials. This allows each WSGI instance (workers) to use a single database connection, and therefore help avoid issues with there being too many connections to the database under the same user. | |
| **Encapsulation**  This uses encapsulation, as it uses private variables to store the values of parameters. These should not be changed at any point, as they would change how certain processes are performed, such as the hashing algorithm or salt. If these were changed during execution, all the passwords would be incorrect, and would therefore would prevent users from signing in correctly. | |
| **Aggregation**  This uses aggregation. It takes both a mysql\_handler Connection object as a parameter, to be able to query the database, and a reading\_lists ReadingLists object so that reading lists can be created when a new user account is created. Given that these are taken as parameters, they can exist after the Account instance is destroyed, so therefore it is aggregation rather than composition. | |

| **Hashing a password** | | |
| --- | --- | --- |
| All passwords should be hashed before being stored in the database. | Met |  |
| def hash\_password(self, password):  result = hashlib.pbkdf2\_hmac(  self.\_hashing\_algorithm,  password.encode("utf-8"), # Needs to be in binary  self.\_hashing\_salt,  self.\_number\_hash\_passes  ) # Salt needs to be in binary - stored as binary in config  # Recommended number of passes is 100,000  return result.hex() # Hash is returned as a hex string, so  # converts back | | |
| This method takes a string, and hashes so that it is in a form that can safely be stored within the database. It uses Python’s hashlib module, instead of a custom implementation, as it will be significantly more efficient, and secure, as well as being able to add salts to the hashes correctly. It uses the hashing algorithm given when the object was created, and the salt, and the number of passes. It also converts the password into binary, using the encode method, which replaces each character with the binary value as specified in the unicode character set. The hashing salt is already in binary so does not need to be converted. The result from the function is encoded using hexadecimal, so is converted into a string using the built-in .hex() method. This is then returned, and can safely be stored in the database, or used to check a given password matches a value stored within the database.  This method can be used to hash a password, so it can be safely stored within the database, as passwords must not be stored in plaintext. This also enables it to be used to convert a given password, so that it can be checked whether it matches the stored version. | | |
| **All passwords should be hashed before being stored in the database.**  This method helps to meet the objective. This is because it can be used to take a plaintext password and hash it, which then allows it to be stored in the database, or compared against items within the database. | | |

| **Check a given username and password are valid** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
| def check\_credentials(self, username, password):  entered\_password = self.hash\_password(password)  query\_result = self.\_connection.query(  """  SELECT password\_hash, user\_id FROM users  WHERE username="{}";  """.format(username)  )  if (len(query\_result) == 0 or  query\_result[0][0] != entered\_password):  raise InvalidUserCredentialsError(username)  else:  return query\_result[0][1] | | |
| This method checks whether a username and password are stored in the database. If they are, it returns the corresponding user ID, but if they are not, it throws the InvalidUserCredentialsError.  It first hashes the password given to the function. This needs to be done, as the value in the database for the password has also been hashed, so therefore the input needs to be hashed so that it is in the same form as that stored in the database to allow for the two values to be compared. Assuming the hashing algorithm, the salt, and the number of passes are the same from when the record was added to the database, they will then match if the passwords entered are the same.  It queries the database, getting the hash stored in the database, as well as the corresponding user ID for the provided username. If the length of the list that is stored in query\_result is then 1, the username must be a valid one, and be present in the database. However if it is 0, then there is not a user in the database with that username. It cannot be greater than 1, however, as the usernames must be unique.  Finally it checks whether the length of the query\_result list is 0 (so the username is invalid), and whether the password hash stored in the database is the same as the hash of the password passed into the function. It would logically raise an IndexError if the length of the query\_result is 0 and it tries to access the first item within it, however, this is not run, as soon as the first part of the condition is run, it will give true if the first index does not exist, and the second part of the clause is skipped. This prevents the IndexError occurring, so this always works. If there is not a known username, or the password does not match the stored one, it throws the InvalidUserCredentialsError. It cannot use a return value, as it will return the user ID otherwise, so it is not easy to check for otherwise. Throwing an errors is therefore the most logical and simple approach.  If both the username and password match, then the function returns the user ID. This was retrieved in the database query, and must be present if the error was not thrown. The result of the query is given as a list of tuples, regardless of how many rows, or columns, were given by the query. There must only be one row, so the first tuple can be selected. This is because the usernames must be unique, and it would have been stopped before it could get to the return statement if there were not any results. The user ID is then the second item within this tuple, and this value is then returned.  It returns the user ID, as it would likely be subsequently needed. It has a negligible impact on performance anyway, so returning it is worthwhile to simplify some uses of the function. In the most obvious application, where signing into the website, the user ID would then need to be found. Performing queries is expensive, so minimising the number is essential. Returning the user ID here eliminates the need for a subsequent query, and has a minimal impact on the time complexity of this method. Therefore it should be returned. | | |
| **Exception Handling**  This uses exception handling, through throwing the InvalidUserCredentialsError when the credentials supplied are invalid, either by an unknown username, or incorrect password. Throwing an error is very clear and simple to catch using a try-catch statement, and is much clearer than checking for a value, such as a boolean result if it is not valid. | | |
| **String Manipulation**  This method uses string manipulation to add the username provided to the function to the SQL query that is being made. It enables the where clause to be used, and prevents the need to search through the result to check whether the target username is included in the users table. Using the where clause will be significantly faster than searching through it using python. It adds the username to the query using the .format() on the string. | | |
| **MySQL**  This function uses a MySQL query to get the password\_hash and the user ID from the database with a specific user ID. This is done using the following query. This allows for clear checking of the stored data, and is also very clear and simple to perform.   | SELECT password\_hash, user\_id FROM users  WHERE username="<username>"; | | --- | | | |
| **Users must be able to log in to the website to access their personalised information.**  This method helps to meet this objective. This is because it checks whether a given set of credentials are valid. This can then be used to check whether a user can log on to the website using their own personalised information. | | |

| **Create a new user account** | | |
| --- | --- | --- |
| All passwords should be hashed before being stored in the database. | Met |  |
| Users must be able to create their own accounts on the website. | Met |  |
| Users should be able to record the books that they have read in the past. | Met |  |
| Users should be able to record the books that they are currently reading. | Met |  |
| Users should be able to record the books that they would like to read in the future. | Met |  |
| *Have Read*, *Currently Reading*, and *Want to Read* reading lists should be automatically created when a user signs up. | Met |  |
| def create\_user(self, first\_name, surname, username, password):  query\_result = self.\_connection.query(  """  SELECT username FROM users  WHERE username="{}"  """.format(username)  )  if len(query\_result):  raise UserExistsError(username)  else:  self.\_connection.query(  """  INSERT INTO users (first\_name, surname, username, password\_hash)  VALUES ("{first\_name}", "{surname}", "{username}", "{password}");  """.format(  first\_name=first\_name,  surname=surname,  username=username,  password=self.hash\_password(password) # Password must  # be hashed before storing in the database.  )  )  user\_id = self.get\_user\_id(username)  self.\_reading\_lists.create\_list(user\_id, "Want to Read")  self.\_reading\_lists.create\_list(user\_id, "Currently Reading")  self.\_reading\_lists.create\_list(user\_id, "Have Read")  return user\_id | | |
| This method creates a new user account. It takes the user’s first name, surname, username and their entered password as parameters, each as strings. It is a function, as it does not return any values. If it is done successfully, it returns the user ID of the new user, and if it fails it raises the UserExistsError.  It first queries the database, getting the username for a specific user, which is limited to a single username through the where clause. The username entered for the new account is inserted into the where clause before the query. The column that is retrieved from the users table does not matter, as it is only there to check whether the username already exists. If this query gives any results, the username is taken, and cannot be used. In this case it throws the UserExistsError, which indicates that the desired username is taken and cannot be used. However, if the query does not give a result, the username has not been used, and can therefore be used. This ensures that the usernames stored in the database remain unique.  If the username can be used, it then inserts a new user record with the the username, first name and surname. It also hashes the password and includes that in the MySQL statement too. It is hashed so that the plaintext password is never stored in the database.  Following this it gets the user ID of the new user, using the get\_user\_id method, so that it can be returned. It also is needed to create the default reading lists. After getting the user ID, it then uses the reading\_lists object passed in when the object was constructed, to create the three default reading lists, the ‘Want to Read’, ‘Currently Reading’ and ‘Have Read’ lists. This then returns the user ID after this. It wouldn’t be any faster to not return it, as it is required for parts of the function anyway. It would also likely to be needed after this function is called anyway, to generate the session ID, so it is logical to return it. Furthermore, as it uses an error rather than a return value to indicate the failure, it is not an issue to return it either. | | |
| **Error Handling**  This method uses error handling. When a username has been taken, and therefore cannot be used for the new user account as it would prevent usernames being unique. Throwing the custom exception is clearer than returning a boolean value, as well as making it simpler to return the user IDs if it succeeds. Using a try-catch is then very clear as to what has happened, which helps ensure that the code is readable and maintainable. | | |
| **String Manipulation**  This method uses string manipulation to insert the username into the where clause in the first query. This allows the query to be made specific to the data. It also inserts the values to be added to the new user record into the insert query. To do this, Python’s .format() method on a string is used. It is much more readable than concatenating the values, and also avoids issues having to convert the types of the variables, and also can be faster to perform too. | | |
| **MySQL**  This method uses MySQL queries, executed using the custom mysql\_handler connector object, to access and modify data in the database. This is much simpler, faster, and is likely to be more reliable than using text files, or any other approach to store the data. | | |
| **All passwords should be hashed before being stored in the database.**  This method meets this objective, because before the new user record is stored in the database, the plaintext password is then hashed. This hash is then stored in the database, which helps keep them secure. When the method has then finished executing, the local variables, including the plaintext password are then deleted, which further helps keep the project secure. | | |
| **Users must be able to create their own accounts on the website.**  This method helps meet this object as it allows for a user’s credentials to be taken, and a new record made in the user’s table. This allows for the new account to be created, and treated as a new, separate account. It then allows for a user to sign in, and access the account’s data. | | |
| **Users should be able to record the books that they have read in the past.**  **Users should be able to record the books that they are currently reading.**  **Users should be able to record the books that they would like to read in the future.**  This method helps to meet these objectives, as when the new user account is created, three empty reading lists are created, called Have Read, Currently Reading, and Want to Read. These then allow the users to record the books that they have read in the past, are currently reading, and those that they want to read in the future, respectively. Therefore this method helps to meet these three objectives. | | |
| **Have Read, Currently Reading, and Want to Read reading lists should be automatically created when a user signs up.**  This method meets this objective. When a user signs up, this function is used to create their new account, so is required to create the new account. As part of this process, it creates the Have Read, Currently Reading, and Want to Read lists for the user, and therefore, completely meets this objective. | | |

| **Get a user id from a username** | |
| --- | --- |
| def get\_user\_id(self, username):  query\_result = self.\_connection.query(  """  SELECT user\_id FROM users  WHERE username="{}";  """.format(username)  )  return query\_result[0][0] | |
| This method finds the corresponding user ID from the database from a given username. It does this by querying the database, getting the user ID from the users table, where the username matches the one passed to the function. The passed username is added to the query using the .format() method on the string.  This query will give either one or zero results, as the usernames are unique. Furthermore, the result from the query will be a list of tuples, so assuming there is a result, it needs to take the first item of the list, which will be the only tuple, and then the first item in that tuple, which will also be the only value. This value is the user ID, which is then returned. This method does not check whether the query returned a value, so if it does not, when the username is not valid, it will raise an IndexError. However, this method will not be used without already knowing the username exists, so this is not a consideration | |

| **Simple overview** | |
| --- | --- |
| def get\_user\_id\_list(self):  res = self.\_connection.query("SELECT user\_id FROM users")  return [i[0] for i in res] | |
| This function gets a list of all the user IDs from the database. It first queries the users table, getting all the user IDs from it. This returns a list of tuples, with each tuple only having a single value in it. This is not a logical format to use, so should be converted into a list of user IDs (integers).  The list of tuples is flattened, and converted into a normal list using a list comprehension. It iterates through the list of tuples, and for each one takes the first element from it (the only one), and adds it to the list. This list is then returned. Using a list comprehension is faster and more memory efficient than using a traditional for loop, which is why it is used. | |

##### Sessions class

| **Custom Exceptions** | |
| --- | --- |
| class SessionExpiredError(Exception):  def \_\_init\_\_(self, session\_id):  message = f"Session id '{session\_id}' has expired"  super().\_\_init\_\_(message) | |
| This custom exception is for when a session ID has expired. Expiring sessions prevents them for being used indefinitely, and with periods of activity. This should then be thrown when a single session token has not been used in a long time, which should help improve security. This is much easier to handle than checking for a specific return value from functions. It should make the code much more readable.  It works by inheriting the class from the Exception base class. When it is called, it is then given the session ID that was used, so it is easier to tell which session has expired when debugging the code. This is then inserted into a message string, which is passed to the Exception class’s constructor, which will output the message when the error is thrown. This should help make it easier to debug the program. | |
| **Error Handling**  This exception can be thrown when a session has expired to prevent further execution using an invalid session, which should help improve security. It can be caught quickly and easily, and then send a logical error message to the client, to get them to log back into the website. The error is also very easy to catch, and will also be readable. It is also more logical to be an error, as it is an issue, as the user ID cannot be used, and processing cannot continue assuming that the session ID is correct. | |

| **Sessions class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, token\_size):  self.\_connection = connection  self.\_token\_size = token\_size | |
| This is the constructor for the sessions class. It takes a database connection object, like with the accounts class. It also takes the token size, which refers to the number of bytes used within the session ID. They are stored within private variables as they do not need to be accessed or used anywhere else. Furthermore, they are taken as parameters here, rather than when the function is executed as accessing a variable is likely to be faster, and also simplifies how the other methods are called. Like the accounts class, it takes a database connection object, instead of creating its own, so each WSGI script can use only a single database connection, as these can be limited. | |
| **Encapsulation**  This uses encapsulation, as it uses private variables to store the database connection and the token size parameter. This is because they do not need to be changed or accessed from outside the class. Privatising these therefore makes it safer. | |
| **Aggregation**  This uses aggregation, as it takes a mysql\_handler connection object as a parameter. This allows for the methods to query the database without it being passed into it. As it is taken as a parameter, it can exist after the Sessions object is destroyed, and is therefore aggregation, not composition. | |

| **Create session** | |
| --- | --- |
| def create\_session(self, user\_id):  token = secrets.token\_bytes(self.\_token\_size).hex() + str(time.time()).split(".")[0]  # the time to make it shorter  # Generates a random string, and adds time to reduce required size  # of the randomly generated string for speed.  # https://docs.python.org/3/library/secrets.html  # Probability of getting duplicates is very low, and gets lower as  # the size of the string increases. It would also need to be within  # 1 second, as time.time() is added to the end which is the number  # of seconds since the epoch.  self.\_connection.query(  """  INSERT INTO sessions (client\_id, user\_id) VALUES ("{token}", {user\_id});  """.format(token=token, user\_id=user\_id)  )  return token | |
| This function creates a new session for a user, taking their user ID as a parameter. First it generates the session ID. It uses the secrets method, token\_bytes, to generate a string that is the same number of bytes long as the token size, provided when the object was instantiated. This gives a hexadecimal string, so it is converted back to a string using the .hex() method. It also adds the current time since the epoch to the end of the random string, This reduces the length of the random string that needs to be generated, as the count would then reset every second. It then inserts the new session ID into the sessions table, linking the session ID to the user ID. Finally, it returns the session ID.  It does not check whether the session ID is already in use. This should not be an issue, as the time means that the number of available sessions restarts every second. Also with a large enough token size, this will become very unlikely, as explained in the analysis. Furthermore, as the token size is configurable, it will match the number of users, so is incredibly unlikely, and is not worth the additional complexity of checking. Furthermore, it is not expected to have a particularly large number of users, with thousands signing in every second, and is therefore not an issue.  It uses the secrets library over the random library, as they should be secure strings. This does mean it is more computationally expensive than using the random library, however it needs to be secure, so must use the secrets library. | |

| **Update the time used of a session ID** | |
| --- | --- |
| def update\_time(self, session\_id):  self.\_connection.query(  """  UPDATE sessions  SET  date\_added=NOW()  WHERE client\_id="{}";  """.format(session\_id)  ) | |
| This procedure updates the time a session ID was last used to the current time. It does this by setting the time created in the sessions table to the current time, which will reset the expiry counter. This query is explained in the documented design chapter. | |

| **Get the user ID from a session ID** | |
| --- | --- |
| def get\_user\_id(self, session\_id):  res = self.\_connection.query(  """  SELECT user\_id, date\_added FROM sessions  WHERE client\_id="{}";  """.format(session\_id)  )  if len(res) == 0:  raise SessionExpiredError(session\_id) # If there is no entries  # it must have been deleted by a maintenance script, as it had  # expired.  else:  res = res[0] # Gets first element result from list - should  # only be one result  expiry\_datetime = res[1] + datetime.timedelta(days=7)  # Set expiry date to one day after it has been last used  if datetime.datetime.now() > expiry\_datetime:  self.close(session\_id)  raise SessionExpiredError(session\_id)  else:  return res[0]  # Does not update the session time - Excluded from this as any  # request from the client indicates that is still active,  # regardless of whether the user id is needed to carry out the  # required process. | |
| This function gets the user ID from a session ID. It also closes the session if it has expired, and throws the SessionExpiredError.  It first gets the user ID and the date the session was created from the sessions table, for the session ID passed into the function. If this does not give a result, it is either a guessed session ID and is invalid, or has been removed by a maintenance script. In these cases, the length of the list output will be zero, and therefore the SessionExpiredError must be thrown, as it cannot proceed without a session ID. This error can encorrporate the session ID not existing, as they both have the same effect.  However, if the length of the list from the query is one (it should only ever be one or zero as session IDs must be unique), it then needs to check whether it has expired. If it has expired, it then reassigns the query result to being the tuple of the first item, as it is the only one that is needed. This reduces what needs to be done to access each of the elements of the result, so makes it simpler. To check whether it has expired, it increases the date added (datetime object) by a week, and then compares it with the current time. If the expiry time is before the current time, it has therefore expired in the past, and should be removed. In this case, it closes the session given to it to keep the table size to a minimum, and then raises the SessionExpiredError, as it has expired. If it is not expired, however, it returns the user ID that was retrieved from the MySQL query. | |
| **Exception handling**  This method uses exception handling. It throws the SessionExpiredError if the session ID given to the function was expired. This prevents further processing being performed, with an invalid session ID, which should help improve security. It also throws the same error if the session ID does not exist, as it cannot be used either. Both of these occurrences need to be handled the same way anyway, so they use the same exception. | |

| **Close a session** | |
| --- | --- |
| def close(self, session\_id):  self.\_connection.query(  """  DELETE FROM sessions  WHERE client\_id="{}";  """.format(session\_id)  ) | |
| This method closes a session. This can be done explicitly from the client when they press the sign out button, or implicitly when a session ID has not been used in a long time. It performs the query explained in the document design chapter, inserting the session ID passed into the where clause to specify the single session ID. | |

| **Get a list of session IDs** | |
| --- | --- |
| def get\_session\_id\_list(self):  return [i[0] for i in self.\_connection.query("SELECT client\_id FROM sessions")] | |
| This method gets a list of session IDs from the database. It does this using a database query, then a list comprehension to convert it into a list of strings. This is more efficient, both in time and memory, than a for loop, and also can be done on one line.  It performs the query, which gives a list of single-element tuples. It then iterates through each item in this, taking the first and only value in the tuple, and adding it to the list. This constructs a list of all the IDs, using list comprehension, which can then be returned directly. | |

#### Authors (./backend/components/authors.py)

The authors part of the components module has functions and procedures to handle all of the related tasks to authors. This includes following and unfollowing authors, getting about data, and getting books for a users favourite authors. The module includes a class, as well as associated functions and exceptions. The class handles all the processes that require a database connection, and the other tasks are handled by functions.

| **Convert individual name components to a display name** | | |
| --- | --- | --- |
| It should show the author’s name (or the name that they write under if applicable). | Met |  |
| def names\_to\_display(first\_name, surname, alias):  if (alias is not None and  (first\_name is not None and surname is not None)):  author = f"{alias} ({first\_name} {surname})"  elif (alias is not None and  (first\_name is None and surname is None)):  author = alias  else:  author = f"{first\_name} {surname}"  return author | | |
| This function takes the author’s first name, surname and writing alias and converts them to a string that can be displayed on the webpage. It is required as some of these fields are not essential. Either all three can be specified, the alias can be specified and the first name and surname being excluded, or the first name and surname can be specified and the alias being excluded. This function takes the three possible options, and constructs the corresponding strings that can be displayed from this. The following table outlines the different options, and the corresponding output   | **Alias:** alias  **First name:** first-name  **Surname:** surname | alias (first-name surname) | | --- | --- | | **Alias:** alias  **First name:** None  **Surname:** None | alias | | **Alias:** None  **First name:** first-name  **Surname:** surname | first-name surname |   It uses f-strigns to insert the varying values into varying strings. This is because they are fairly quick, and easy to read. The three different valid combinations are then checked using an if-statement, as it is a clear and simple way of matching them. A switch statement would not be appropriate here, even if it were supported, as there are too many different combinations that need to be checked.  This is just a function, and is not included within the authors class. This is because it does not require any of the data stored within the class, such as the database connection. Furthermore, it is also a useful function that is needed elsewhere, but having a authors object created has no other benefit, so having it as a function is most logical. | | |
| **It should show the author’s name (or the name that they write under if applicable).**  This function helps to meet this objective, as it takes the three available name components, and converts them from any valid permutation of given and not given values, to a string that can be displayed on any of the parts of the website. | | |

| **Custom Exception** | |
| --- | --- |
| class AuthorNotFoundError(Exception):  def \_\_init\_\_(self, author\_id):  message = f"Author with ID '{author\_id}' was not found."  super().\_\_init\_\_(message) | |
| This custom exception is for when data is queried about an author, but the author ID used is not in the database. It provides an easy way to tell if an error has occurred, in a simple and readable way. If this occurs, a 404 response should be sent, as the author data requested does not exist. | |

The authors module uses a class rather than individual functions, as it allows parameters to be given once, and reduce the number of parameters for function calls. This makes it much simpler and easier to use. It also avoids using data from the configuration file, which, while as fast as possible, will be slower than using a variable. This class follows the following class diagram, which was excluded in the documented design chapter, as it is part of the construction of the module, and is not essential to the basic project structure.

| **Authors** |
| --- |
| * connection (object) * number\_summaries\_home * number\_genres |
| * follow * unfollow * get\_number\_followers * get\_about\_data * id\_to\_name * get\_author\_id\_list * get\_author\_favourite\_data |

| **Accounts class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, number\_genres, number\_summaries\_home):  self.\_number\_summaries\_home = number\_summaries\_home  self.\_number\_genres = number\_genres  self.\_connection = connection | |
| This is the constructor for the authors class. It takes a database connection object, from the mysql\_handler module, the number of genres to display on the authors about page, and the number of summaries given on the home page. These are then stored within private variables as they do not need to be modified or accessed from outside of the class. Taking the values as parameters here, rather than when the subsequent functions are called reduces the number of parameters that need to be passed, and therefore makes using the methods much simpler.  It passes a database connection object as a parameters so that it can query the database, without having to pass an object into each function. It also is taken as a parameter, so that each WSGI script that is running can use a single database connection, which reduces the number of simultaneous connections to the database. This could be an issue, as some databases have a limited number of concurrent users, which could be an issue if many workers were used for the webserver. | |

| **Follow an author** | | |
| --- | --- | --- |
| Users should be able to follow each author. | Met |  |
| def follow(self, user\_id, author\_id):  self.\_connection.query("""  INSERT INTO author\_followers (user\_id, author\_id)  VALUES ({user\_id}, {author\_id});  """.format(user\_id=user\_id, author\_id=author\_id)) | | |
| This procedure allows a user to follow a specific author, using their IDs. IDs can be used as author IDs are known to the client directly, and session IDs can be reversed to find the corresponding user ID. It marks this relationship by adding a new record to the author\_followers table, as explained in the document design chapter. It adds the user ID and author ID into the MySQL query using the string’s built-in .format() method.  It does not need to return anything, as the query is correct, and functions, and therefore any errors that occur when this is performed are handled within the query method of the custom connector object. This ensures that the query will work correctly. | | |
| **Users should be able to follow each author.**  This method meets this objective, as it adds a new record to the author\_followers table which tracks all the following relationships between users and authors. This then saves the fact that the specified user follows the specified author within the database, so it can be used in the future, and won’t be lost across server restarts either. | | |

| **Unfollow an author** | | |
| --- | --- | --- |
| Users should be able to follow each author. | Met |  |
| def unfollow(self, user\_id, author\_id):  self.\_connection.query("""  DELETE FROM author\_followers  WHERE user\_id={user\_id}  AND author\_id={author\_id};  """.format(user\_id=user\_id, author\_id=author\_id)) | | |
| This procedure allows a user to unfollow a specific author, using their IDs, as both the user ID and author ID will be known. It adds the author ID and user ID to the query using the .format() method, before executing the query. The query was given in the documented design chapter, and explained there, so will not be explained here. Because of the checks made in the query method, the query will run correctly and delete the values, assuming that there is one present. If there is not, it will not raise any errors, which is not an issue.  This method is added, even though it is a repackaged database query, as it abstracts the query away, and makes it much simpler to perform, especially if it needs to be performed multiple times in the code. | | |
| **Users should be able to follow each author.**  This method helps to meet this objective. While not explicitly stated within the objective, it is essential that a user can unfollow an author after they have started following them. When called, it removes the record of the follow from the author\_followers table, which therefore indicates that the user no longer follows the author. This is then persistent across server restarts, and between the varying WSGI workers running. | | |

| **Get the number of followers an author has** | | |
| --- | --- | --- |
| The page should display how many followers each author has. | Met |  |
| def get\_number\_followers(self, author\_id):  return self.\_connection.query("""  SELECT COUNT(author\_id) FROM author\_followers  WHERE author\_id={};  """.format(author\_id))[0][0]  # If the author ID is known, can safely assume that an author is in  # the DB with that name. | | |
| This function gets the number of followers an author has. It performs a database query, which gives the number of followers an author has, which was explained in the documented design chapter. Furthermore, the count in the database query ensures that even if the author ID is not valid, it will give 0, and therefore won’t raise errors. The query gives a list, with a single tuple in it, containing the number of followers that the author has. The first item of the first tuple in the list from the query can then be selected safely, and this is the number of followers the author has. It can then be returned directly. | | |
| **The page should display how many followers each author has.**  This method helps to meet this objective, as it performs a database query, and returns the number of followers the specified author has. | | |

| **Get an author’s about data** | | |
| --- | --- | --- |
| It should display the author’s average book rating. | Met |  |
| It should display all the genres that the author writes for. | Partially Met |  |
| It should show all the books that the author has written. | Met |  |
| It should supply an about the author section. | Met |  |
| It should show the author’s name (or the name that they write under if applicable). | Met |  |
| The page should display how many followers each author has. | Met |  |
| Each author stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| def get\_about\_data(self, author\_id):  res = self.\_connection.query("""  SELECT authors.first\_name,  authors.surname,  authors.alias,  authors.about,  (SELECT count(author\_followers.user\_id) FROM author\_followers  WHERE author\_followers.author\_id=authors.author\_id) AS followers,  ROUND(AVG(reviews.overall\_rating), 2) AS average,  COUNT(reviews.overall\_rating) AS number  FROM authors  LEFT OUTER JOIN books  ON authors.author\_id=books.author\_id  LEFT OUTER JOIN reviews  ON reviews.book\_id=books.book\_id  WHERE authors.author\_id={};  """.format(author\_id))  if len(res) == 0:  raise AuthorNotFoundError(author\_id) # Cannot safely assume  # that it is from a reputable source - it may not be from a  # link, so it should be verified.  else:  res = res[0]  first\_name, surname, alias, about, followers, average\_rating, number\_ratings = res # res is a 4 element tuple, so this unpacks it  author = names\_to\_display(first\_name, surname, alias)  output\_dict = {  "name": author,  "about": "</p><p>".join(("<p>" + about + "</p>").split("\n")),  "author\_id": int(author\_id),  "num\_followers": followers  }  books = self.\_connection.query("""  SELECT book\_id, title, cover\_image FROM books  WHERE author\_id={};  """.format(author\_id))  book\_arr = []  for i in books:  book\_arr.append({  "id": i[0],  "title": i[1],  "cover": i[2]  }) # Author name can be done implicitly from other sent data -  # reduce amount of data sent for speed  output\_dict["books"] = book\_arr  output\_dict["average\_rating"] = float(average\_rating)  output\_dict["num\_ratings"] = number\_ratings  genres = self.\_connection.query("""  SELECT genres.name  FROM genres  INNER JOIN book\_genres ON book\_genres.genre\_id=genres.genre\_id  INNER JOIN books ON books.book\_id=book\_genres.book\_id  INNER JOIN authors ON authors.author\_id=books.author\_id  WHERE authors.author\_id={author\_id}  ORDER BY book\_genres.match\_strength DESC  LIMIT {number}  """.format(author\_id=author\_id, number=self.\_number\_genres))  output\_dict["genres"] = [i[0] for i in genres]  return output\_dict | | |
| This function gets the about data of an author, and packages it into a dictionary. It contains their name, their about, the corresponding ID, the number of followers they have, a list containing dictionaries about all the books they have written (book ID, title and cover image), their average rating, the number of ratings, and a list containing the top number of genres they have written for, specified by the number\_genres parameter in the object constructor.  It first performs a query to get the author’s first name, surname, writing alias, their number of followers, their average rating, to two decimal places, and the number of reviews they have had. Author IDs must be unique, which is enforced by the primary key constraint on the column. | | |
| **It should display the author’s average book rating.**  This method helps to meet this objective, as it queries the database to get the average overall rating of all the books that they have written. This is then included within the dictionary that is returned from the function. | | |
| **It should display all the genres that the author writes for.**  This methods helps to meet this objective, as it queries the database to get the top genres that the specified author has written for. These genres are then included within the dictionary which is then returned at the end of the function.  It is only partially met, as it gives a limited number of genres. This was a conscious decision, as the matrix factorisation used that generates the recommendations and the book genre match weightings gives a weighting for every single genre, unless the match is exactly 0. However this is rare, so giving all of the genres would give almost all of them on the author’s about page, which would be incorrect. Instead it is limited to the top ones, the number of which can then be configured through the configuration file. | | |
| **It should show all the books that the author has written.**  This method helps meet this objective, as it queries the database to get all of the books that the author has written. For each of these books, the data required for a summary is included in a dictionary, and included within the dictionary that is returned by the function. | | |
| **It should supply an about the author section.**  This method helps to meet this objective, as it queries the database for the author’s about paragraphs, and processes them so that they can be inserted straight into the HTML. It is then added into the dictionary that is returned. | | |
| **It should show the author’s name (or the name that they write under if applicable).**  This method helps meet this objective. It gets the name components from the database, and converts them into the string that can be displayed directly. This is then added to the dictionary that is returned from the function, which can then be used to construct the author’s about page. | | |
| **The page should display how many followers each author has.**  This method helps meet this requirement, as it performs a database query, which gets the number of followers (within a larger query for more data). This is then added to the dictionary that is returned, which can then be used to construct the web page. | | |
| **Each author stored in the database should have a dedicated about page, which displays all the available information within the system.**  This method helps to meet this objective, as it gets all of the data required for the author’s about page, and puts it into a dictionary, which can be converted into a JSON, and sent to the client to be inserted into the webpage. Consequently, this can then be used to display all of the information stored within the system on the about page. | | |

| **Get an author’s name from ID** | |
| --- | --- |
| def id\_to\_name(self, author\_id):  res = self.\_connection.query("""  SELECT first\_name,  surname,  alias  FROM authors  WHERE author\_id={}  """.format(author\_id))[0]  return names\_to\_display(res[0], res[1], res[2]) | |
| This function gets the name of an authors, specified by the author ID. It performs a database query, getting the authors first name, surname, and their writing alias from the authors table, limited using the author ID. The author ID passed into the function is added to the query, within the where clause, using the .format() method, before performing the query. This will only give a single result, so it takes the first tuple from the result. The tuple contains the first name, surname and the alias in that order.  It then converts the three values into the string that can be displayed to the client, using the function within the file, which is unassociated with the class. This value can then be returned directly. | |

| **Get a list of authors (as summaries or as IDs)** | |
| --- | --- |
| def get\_author\_id\_list(self, names=False):  if names:  res = self.\_connection.query("""  SELECT author\_id,  first\_name,  surname,  alias  FROM authors  """)  output = []  for count, item in enumerate(res):  output.append({  "name": names\_to\_display(item[1], item[2], item[3]),  "id": item[0]  })  return output  return [i[0] for i in self.\_connection.query("SELECT author\_id FROM authors")] | |
| This function gets a list of the authors in the database. It can either get a list of dictionaries, each containing the name, and the corresponding author ID, or a list of author IDs. This reduces the number of methods required for the module, and can be grouped together, as they are semantically similar. To differentiate between the two, it takes a names parameter, which signifies whether the names should be given. If it is set to true, it should give the dictionaries, otherwise it should give a list containing IDs only. This value defaults to false, so if the parameter is not given, it will give a list of IDs only.  If the names parameter is false, it should return a list of the author IDs within the database. It does this by querying the database and using a list comprehension to take the list of single-element tuples returned by the query to a list of the IDs. It gets the all the author IDs from the authors table in the database, then iterates through the resulting list, taking the first, and only, item from each tuple and adding them to the resulting list. This could be done using a for loop, and appending each time, but this approach is faster, being both more memory and time efficient, as well as being more succinct. This can then be returned, as it is the list of author IDs.  If the names parameter is true, it should return a list of dictionaries containing both the author ID and the corresponding name string. It first queries the database, getting the first name, surname and writing alias for all of the authors in the database, which is returned as a list of tuples, each consisting of three elements (first name, surname and alias in that order). An empty list is created to contain the output dictionaries. It iterates through the list of tuples, from the query, and for each item in it, it appends a dictionary to the output list. This dictionary contains the id, which is the first item in the tuple. It also contains the authors name, in the form that can be displayed to the user, which is found using the names\_to\_display function. This output list is then returned. | |

| **Get most popular books from the authors a user is following** | | |
| --- | --- | --- |
| The browse page should display books from the authors the user is following | **Met** |  |
| def get\_author\_favourite\_data(self, user\_id):  res = self.\_connection.query("""  SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias,  AVG(reviews.overall\_rating) AS average\_rating  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  LEFT OUTER JOIN reviews ON reviews.book\_id=books.book\_id  WHERE books.author\_id IN (SELECT author\_followers.author\_id  FROM author\_followers  WHERE author\_followers.user\_id={user\_id})  GROUP BY books.book\_id  ORDER BY average\_rating DESC  LIMIT {limit};  """.format(  user\_id=user\_id,  limit=self.\_number\_summaries\_home  ))  if len(res) == 0:  return None    output\_dict = dict()  for i, k in enumerate(res):  output\_dict[i] = {  "author": names\_to\_display(k[3], k[4], k[5]),  "title": k[0],  "book\_id": k[1],  "cover": k[2],  }    return output\_dict | | |
| This function gets the highest rated books that have been written by authors that the specified user follows. It takes the user ID as a parameter, to specify the user to get the data for.  It first performs the query to get the books title, cover image, and the authors first name, surname and writing alias from the database. This query is explained in the documented design chapter. The where clause specifies the user that the query gets data for, and the limit specifies the number of results to be given. The limit is faster than limiting the results in the Python, as the amount of data returned is minimised. The limit is inserted into the query string, with the value that was passed into the number\_summaries\_home when the object was instantiated, and the user ID is also inserted. This is done using the .format() method, so the variable type does not need to be considered. This query is then executed, and returns a list of tuples.  It first checks that there was any results from the query. This is done by checking whether the length of the resulting query was zero, as if it is, there are no results. In this case, None is returned, as the user specified does not follow any authors, so there are not any applicable results. Checking for None values is easy to check for.  If there are results, however, it then creates an dictionary for the output. It iterates through the result of the query, and uses the enumerate function to give both the value in the result, as well as the iteration number. For each of these, it adds a new item to the dictionary, with the count as the key. The corresponding value is a dictionary, which contains the book title, ID, and cover, as well as the author’s name, in the correct display format. The first name, surname and alias found in the query are put into the names\_to\_display function to get the string. This is performed for all of the items in the query result. This dictionary is then returned. | | |
| **The browse page should display books from the authors the user is following**  This method helps to meet this objective, as it gets the top books, and the summary data for them, from all the authors a specific user is following. The dictionary that is returned can then be used to help construct the browse page. | | |

#### Books (./backend/components/books.py)

The books part of the components module has functions and procedures to handle all the related tasks to authors. This includes getting similar items, getting summary data, leaving reviews and deleting reviews. It has been done as a class, which needs to be instantiated before the functions can be used, as it holds required data. It includes a database connection object, to reduce the number of connections that are made to the database, a reading lists object, the number of similarities to be given, the number of summaries for the home page, and the number of genres to be displayed, when relating to the book. This reduces the number of parameters that need to be passed in for each function, which should make it easier to use, which is why it has been made as a class, so this data is stored within the class, and can be used directly. This is therefore simpler to use. It follows the following class diagram

| **Books** |
| --- |
| * connection (object) * reading\_lists (object) * num\_display\_genres * number\_summaries\_home * number\_similarites\_about |
| * get\_similar\_items * get\_summary * get\_newest * get\_about\_data * delete\_review * leave\_review * get\_highly\_rated |

| **Book not found exception** | |
| --- | --- |
| class BookNotFoundError(Exception):  def \_\_init\_\_(self, book\_id):  message = f"Book with ID '{book\_id}' was not found."  super().\_\_init\_\_(message) | |
| This custom exception is for when data is accessed about a book, using the ID, but the ID used is not in the database. It provides an easy way to tell if an error has occurred, in a very simple and readable way. If this error is thrown, it is likely that a 404 status code should be sent to the client, as the about page they are requesting does not exist. | |

| **Books class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, reading\_lists, number\_similarities\_about, number\_summaries\_home, num\_display\_genres):  self.\_reading\_lists = reading\_lists  self.\_num\_display\_genres = num\_display\_genres  self.\_number\_summaries\_home = number\_summaries\_home  self.\_number\_similarities\_about = number\_similarities\_about  self.\_connection = connection | |
| This is the constructor for the books class. It takes a database connection object, from the mysql\_handler module, a reading\_lists object, from the components module, the number of similarities to be shown on the books about page, the number of summaries to be shown on the home page, and the number of genres to be displayed on the about page. These are all stored as private variables, as they do not need to be accessed from outside of the class, and modifying them would change how the results appear, which should not be done from outside of the class, and only from explicit changes to the configuration file. These parameters are taken here, as opposed to when the functions are called, so that fewer parameters need to be passed to the functions, and if this was done instead, the class would be pointless. This makes it simpler to use each of the functions and procedures.  The database connection object is given as a parameter, so that the database can be queried in any of the functions. This reduces the number of database connections, which may be an issue, as some database servers have a maximum number of connections, which may then be quickly used across all of the WSGI workers. The reading\_lists object is also taken, so that some of those methods can be used directly. | |

| **Get similar book summaries** | | |
| --- | --- | --- |
| There should be a list of books that are related to each other displayed. | Met |  |
| def get\_similar\_items(self, book\_id):  genre\_num = math.ceil(self.\_connection.query("SELECT COUNT(genre\_id) FROM genres")[0][0] \* 0.025)  res = self.\_connection.query("""  SELECT books.book\_id,  GROUP\_CONCAT(book\_genres.genre\_id  ORDER BY book\_genres.match\_strength DESC  LIMIT {}  ),  books.author\_id  FROM books  INNER JOIN book\_genres  ON books.book\_id=book\_genres.book\_id  GROUP BY books.book\_id;  """.format(genre\_num)) # Match strength is included, as then more  # are more likely to appear, and therefore will impact on the  # similarity.  genre\_dict = dict()  for i in res:  items = set(int(k) for k in i[1].split(","))  items.add(-i[2]) # Must be negative, as otherwise it would be  # treated as a genre, which may not work  genre\_dict[i[0]] = items  target\_genres = genre\_dict[book\_id]  genre\_dict.pop(book\_id)  tree = data\_structures.BinaryTree(access\_function=lambda x: x["jaccard\_distance"]) # Insert into tree using  # similarity not id  for i in genre\_dict:  tree.insert({  "book\_id": i,  "jaccard\_distance": 1 - ml\_utilities.jaccard\_similarity(target\_genres, genre\_dict[i])  # 1 - similarity  # gives the distance, so this in ascending order gives them  # in similairty in descending order.  })  result = tree.in\_order\_traversal()[:self.\_number\_similarities\_about]  # Get the books ordered  # by similarity. Note that the distance is  # descending - This is correct, as 0 is identical genres, and 1 is  # different  return [self.get\_summary(i["book\_id"]) for i in result] | | |
| This function gets the book summaries (titlle, author name, cover image, and book ID) for the most similar books to a specific one, specified by the book ID. This is done by getting the top 2.5% of the genres for every book in the database, as well as the author, and using jaccard similarity to find the similarity between each of these items and the specified book. They are then ordered by the jaccard similarity descending, and the top number of the similarities, specified by the number of similarities parameter from the constructors. A list of all the summaries is then returned.  First, it gets the number of genres that should be considered for each book, which should be 2.5% of all the genres stored in the database. This is done by performing a query to get the number of genres that are stored in the database, and mltiplying it by 0.025 to get 2.5% of these. This cannot be a decimal number, so is then changed to being an integer. This is done using the ceiling method, always rounding up, so that it can never be 0 genres, although this would not be an issue if there was more than 1 genre in the database. The value of 2.5% was based upon the training dataset, which used 727 distinct genres. This would then mean the check was based upon 19 genres, which is a reasonable number, which will check for very similar books. If these 19 do not match well, values for all the books in the database will still be given.  Following this it performs a query to get the book ID, the top genres for that book, as a comma separated string, and the author ID for the book. The top genres, are the same number as found in the previous line. This uses a different query to that given in the doucmented design. This was because of an oversight, where it gave all the genres for each book, and not in any order, so could not be reduced in the Python, which would mean that the similarities would be invalid. This query gets the book ID and the author ID from the books table. It also uses the GROUP\_CONCAT method to get a comma separated string of the top genre IDs, with the highest match strength. This is done by ordernig by the match strength descending, so the highest matches come first, then limiting the number of genres in the string. To be able to do this, the query must be joined to the book\_genres table, matching the book ID from the books table to the book ID from the book\_genres table. For the GROUP\_CONCAT to work, the group by is used to group results by the book ID, so that they are all linked.   | SELECT books.book\_id,  GROUP\_CONCAT(book\_genres.genre\_id  ORDER BY book\_genres.match\_strength DESC  LIMIT <genre percentage>  ),  books.author\_id  FROM books  INNER JOIN book\_genres  ON books.book\_id=book\_genres.book\_id  GROUP BY books.book\_id; | | --- |   The number of genres is inserted into the query, using the .format method on the string, which limits the number of genres considered. Doing this dynamically means that the project can have any number of genres, and it will still work similarly. It will never use all the genres in the project, unless there is only one genre in the database, so will always give good similarities.  It then creates a dictionary, as a lookup table, for each of the books, containing a set of all the genre IDs for each book. This is done by iterating through all of the results from the query, with each item being a three-element tuple. It then uses a generator to create a set of the all the IDs, by splitting the concatenated string at the commas, and casting each item in it to an integer. It also adds the author ID to the set, but as a negative number. As the Jaccard similarity only compares the occurrences of items, it will work, as the negative number cannot match the positive values used with the genre IDs, so will not affect the result. This also ensures that the author is considered, but weighted the same as a genre. This set is then added to the dictionary, with the key being the book ID. This dictionary is used as a lookup table. It would have also been appropriate to use a hash table instead of a dictionary. However, within Python, dictionaries are implemented in the same way as a hash table is, when they are accessed, the key is hashed, and the location found from this. Therefore, implementing a custom hash table instead would have been more complex than using a dictionary, and also would, likely be significantly slower  It then removes the set for the specified book, specified in the parameter, and assigns this as a separate variable. This is the target set, and the most similar books will be as close to this as possible.  Following this, it creates a binary tree, using the custom class from the custom datastructures module. It also uses a custom access function so that dictionaries can be pushed into the tree, and the data from them be used directly after it has been traversed. It iterates through all the items within the dictionary, and inserts a new record into the binary tree for each of them, containing the book ID, and the jaccard distance, which is the 1 minus the jaccard similarity. The Jaccard similarity is calculated using the custom implementation in the ml\_utilities module, which is also why sets were used over lists. Sorting the distance in ascending order gives the similarity in descending order, which is what is desired. Traversing the binary tree, using an in order traversal will then give the similarity descending. The list returned from this traversal is then reduced to the first items, specified by the number\_similarities\_about parameter, given when the object was instantiated.  Finally, it uses a list comprehension, to create a list of dictionaries of the summary data for each of the books. It does this by iterating through the list of dictionaries returned from the tree traversal, and for each of them using the get\_summary data method, within this class, which gives a dictionary containing the book title, authors name, and the book cover. This can then be returned. It is done here, rather than in the first query, as it should scale better with more books. If it were done with the first query, the time for the query would increase as the number of items in the database increases. It would also impact the space efficiency greatly, as many of the books data is unnecessary, and would be discarded anyway. This ensures that it is as space efficient as possible, and should scale well with the number of books in the database. | | |
| **There should be a list of books that are related to each other displayed.**  This method helps to meet this objective, as it takes a book ID and finds the most similar books to it. The list of summary dictionaries that are then returned can be used as part of the home page, or to find related books on the *browse* page. | | |

| **Get a book summary by either ID or ISBN number** | | |
| --- | --- | --- |
| Book summaries should include the book’s cover image, author, and title. | Met |  |
| Each book summary should link the specific books about the page. | Met |  |
| def get\_summary(self, book\_id=None, isbn=None):  if book\_id is not None:  res = self.\_connection.query("""  SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  WHERE books.book\_id={}  """.format(book\_id))  else:  res = self.\_connection.query("""  SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  WHERE books.isbn="{}"  """.format(isbn))  if len(res) == 0:  raise BookNotFoundError(book\_id)  res = res[0]  return {  "author": components.authors.names\_to\_display(res[3], res[4], res[5]),  "title": res[0],  "book\_id": res[1],  "cover": res[2],  } | | |
| This function gets a books summary by book ID, or by ISBN number. Both parameters for the function, the book ID and the ISBN default to None. This allows for it to be specified easily, and also checked against. The book ID option is more likely to be used than the ISBN number, so it comes first. If a parameter is given, and it is not specified as to whether it is an ISBN, or book ID, it is assumed to be the book ID.  If the book\_id parameter has been given, it will not be None. If this is the case, it should get the book title, ID, and the authors name components by the book ID. The query that is used was explained in the documented design. The book\_id taken as a parameter is inserted into the MySQL query using the .format method, as it is simple to use, and fairly performant. This is then executed. If the book ID parameter was not specified, the ISBN must be being used. It then performs the same query, except the where clause uses the ISBN number instead of the book ID. It also uses the .format method to insert the ISBN passed, before executing the query.  The queries are the same, except for the where clause. This means that they both give lists of six-element tuples, with the fields in the same order, which also allows for the subsequent processes to be the same for both. If the list from the query is empty, there are no results for the query, either by book ID, or ISBN, depending on which parameter was given. The function checks for this, and if it is empty, it raises the BookNotFoundError, as the requested book is not within the database. However, if it is valid, it can take the first tuple in the result list, which can be safely done as both ISBN numbers and book IDs are unique, so the list will have at most one item in it.  It then constructs a dictionary, containing the title, book ID, and cover image. It also adds the author’s name, which is found using the function from the authors part of the components module. It has the first name, surname and writing alias from the query passed into it, and gives the name string. This dictionary contains the summary data, and can then be returned. | | |
| **Book summaries should include the book’s cover image, author, and title.**  This method helps meet this objective, as it gets the data from the database, for a single book. It puts this data into a dictionary, which contains the book’s cover image, author and title. This can be used directly by the JavaScript on the client, to create a book summary. Therefore, this helps meet this objective. | | |
| **Each book summary should link the specific books about the page.**  This method helps to meet this objective, as it includes the book ID, which can then be used by the client to load the corresponding about page. This is because the about pages are loaded by using the book ID, so therefore, it can be used to load the corresponding about page. | | |

| **Get newest books added to the database** | | |
| --- | --- | --- |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website | Met |  |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. | Met |  |
| def get\_newest(self):  res = self.\_connection.query("""  SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  ORDER BY books.date\_added DESC  LIMIT {};  """.format(self.\_number\_summaries\_home)) # Get the first n books  output\_dict = dict()  for i, k in enumerate(res):  output\_dict[i] = {  "author": components.authors.names\_to\_display(k[3], k[4], k[5]),  "title": k[0],  "book\_id": k[1],  "cover": k[2],  }  return output\_dict | | |
| This function getsthe summaries for the newest books in the database. It first performs a query, which contains all of the summary data for all of the books in the database, and orders the results by the date they were added. The first of these are then taken, which is specified by the number\_summaries\_home parameter, given when the object was instantiated. It is inserted into the query, again using the .format method, before executing the query. The results from this query are then iterated through, and constructs a dictionary, with the key being the number of the book in the result, so key 0, is for the newest book, and so on. Each of the subdictionaries contain the author name, again found using the method from the authors file, the title, book ID and cover image. This dictionary can then be returned after the all the query results have been iterated through. | | |
| **There should be a browse page on the website to display popular books, highest rated, and the newest additions the website**  **The home page should give users a list of trending books, which have the highest number of users currently reading them.**  This method helps to meet this objective. This is because it gets all of the book summaries for the newest books in the database. It gives the top number, in order, which the book summary data is then retrieved, and added to a dictionary, which can then be returned. This can then be converted to JSON, and sent directly to the client, to be processed by the JavaScript to display the newest additions on the website, on the about page, and on the home page. Therefore, it helps to meet all of the second objective, and the newest additions part of the first objective. | | |

| **Get the about data for a specific book, including the current user’s review** | | |
| --- | --- | --- |
| The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book | Met |  |
| A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website. | Partially Met |  |
| Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system. | Met |  |
| Reviews should be listed on the corresponding book about page. | Met |  |
| Reviews on the book should show the user’s username. | Met |  |
| def get\_about\_data(self, book\_id, user\_id):  res = self.\_connection.query("""  SELECT books.title,  books.cover\_image,  books.synopsis,  books.purchase\_link,  books.release\_date,  books.isbn,  authors.first\_name,  authors.surname,  authors.alias,  authors.about,  (SELECT COUNT(author\_followers.author\_id) FROM author\_followers  WHERE author\_followers.author\_id=books.author\_id) AS author\_num\_followers,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Want to Read"  AND reading\_lists.book\_id=books.book\_id) AS num\_want\_read,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  AND reading\_lists.book\_id=books.book\_id) AS num\_reading,  (SELECT COUNT(reading\_lists.book\_id) FROM reading\_lists  INNER JOIN reading\_list\_names ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_list\_names.list\_name="Have Read"  AND reading\_lists.book\_id=books.book\_id) AS num\_read,  authors.author\_id  FROM books  INNER JOIN authors  ON authors.author\_id=books.author\_id  WHERE books.book\_id={book\_id};  """.format(book\_id=book\_id))  if len(res) == 0:  raise BookNotFoundError(book\_id) # If the query result has 0  # entries, no book was found with the target name  else:  res = res[0]  author = components.authors.names\_to\_display(res[6], res[7], res[8])  genres = [i[0] for i in self.\_connection.query("""  SELECT genres.name FROM genres  INNER JOIN book\_genres ON book\_genres.genre\_id=genres.genre\_id  WHERE book\_genres.book\_id={book\_id}  ORDER BY book\_genres.match\_strength DESC  LIMIT {number}  """.format(book\_id=book\_id, number=self.\_num\_display\_genres))]  output\_dict = {  "title": res[0],  "cover\_image": res[1],  "synopsis": "</p><p>".join(("<p>" + res[2] + "</p>").split("\n")),  # Split at line breaks into paragraph blocks  # Can just be inserted without any processing as it includes  # spacing because of p elements  "purchase\_link": res[3],  "release\_date": res[4].strftime("%d/%m/%Y"),  "isbn": res[5],  "author": author,  "author\_about": "</p><p>".join(("<p>" + res[9] + "</p>").split("\n")),  "author\_number\_followers": res[10],  "num\_want\_read": res[11],  "num\_reading": res[12],  "num\_read": res[13],  "genres": genres,  "author\_id": res[14]  }  res = self.\_connection.query("""  SELECT IFNULL(ROUND(AVG(overall\_rating), 2), 0) AS average\_rating,  COUNT(overall\_rating) AS num\_ratings,  (SELECT COUNT(overall\_rating) from reviews  WHERE overall\_rating=5  AND book\_id={book\_id}) AS num\_5\_stars,  (SELECT COUNT(overall\_rating) from reviews  WHERE overall\_rating=4  AND book\_id={book\_id}) AS num\_4\_stars,  (SELECT COUNT(overall\_rating) from reviews  WHERE overall\_rating=3  AND book\_id={book\_id}) AS num\_3\_stars,  (SELECT COUNT(overall\_rating) from reviews  WHERE overall\_rating=2  AND book\_id={book\_id}) AS num\_2\_stars,  (SELECT COUNT(overall\_rating) from reviews  WHERE overall\_rating=1  AND book\_id={book\_id}) AS num\_1\_star  FROM reviews  WHERE book\_id={book\_id};  """.format(book\_id=book\_id))[0] # This always gives one tuple,  # regardless of whether there are any reviews.  # Even if there is no reviews, the query has results of 0 for all  # these.  output\_dict["average\_rating"] = float(res[0]) # The query gives a  # Decimal type, so cast to float to be used.  output\_dict["num\_ratings"] = res[1]  output\_dict["num\_5\_stars"] = res[2]  output\_dict["num\_4\_stars"] = res[3]  output\_dict["num\_3\_stars"] = res[4]  output\_dict["num\_2\_stars"] = res[5]  output\_dict["num\_1\_star"] = res[6]  if user\_id is None:  output\_dict["current\_user\_review"] = None  user\_id = -1 # This will match all entries, as it is never  # equal to an ID, as they are natural numbers.  else:  res = self.\_connection.query("""  SELECT review\_id,  overall\_rating,  plot\_rating,  character\_rating,  summary,  rating\_body  FROM reviews  WHERE user\_id={user\_id}  AND book\_id={book\_id};  """.format(user\_id=user\_id, book\_id=book\_id))  if len(res) == 0:  output\_dict["current\_user\_review"] = None  else:  body = res[0][5]  if body is not None:  body = "</p><p>".join(("<p>" + body + "</p>").split("\n"))  output\_dict["current\_user\_review"] = {  "review\_id": res[0][0],  "overall\_rating": res[0][1],  "plot\_rating": res[0][2],  "character\_rating": res[0][3],  "summary": res[0][4],  "rating\_body": body  }  res = self.\_connection.query("""  SELECT reviews.review\_id,  reviews.overall\_rating,  reviews.plot\_rating,  reviews.character\_rating,  reviews.summary,  reviews.rating\_body,  reviews.date\_added,  users.username  FROM reviews  INNER JOIN users ON users.user\_id=reviews.user\_id  WHERE reviews.user\_id!={user\_id}  AND reviews.book\_id={book\_id};  """.format(user\_id=user\_id, book\_id=book\_id)) # Inserting None  # will insert a string “None” so will not match any IDs. Does not  # include the current user's review. If it is None it includes all  # users.  stack = data\_structures.Stack() # date added will be newest comes  # last - reverses the order  for i in res:  stack.push(i)  output\_dict["author\_following"] = bool(len(self.\_connection.query("""  SELECT author\_id FROM author\_followers  WHERE author\_id={author\_id}  AND user\_id={user\_id};  """.format(author\_id=output\_dict["author\_id"], user\_id=user\_id))))  # Finds all entries with the same user and author id as required,  # which will either be 1 or 0. If it is 0, the user is not  # following the author, so the author\_following value should be  # false. If it is 1, they are, so it should be true. Len gets the  # number of results (1 or 0), and bool converts this to the  # corresponding boolean value, which is whether the user is  # following the author.  output\_dict["reviews"] = [] # Remove all items in stack and create  # array with them  for i in range(stack.size):  k = stack.pop()  body = k[5]  if body is not None:  body = "</p><p>".join(("<p>" + body + "</p>").split("\n"))  output\_dict["reviews"].append({  "id": k[0],  "overall\_rating": k[1],  "plot\_rating": k[2],  "character\_rating": k[3],  "summary": k[4],  "rating\_body": body,  "date\_added": k[6].strftime("%d/%m/%Y"),  "username": k[7],  })  return output\_dict | | |
| This method gets the about data for a book, including the current user’s review if applicable, and all the other reviews left for the book. It is a very simple process, and mainly involves database queries, and creating a dictionary from this.  First it queries the database, getting the book title, cover image, synopsis, purchase link, release date, ISBN number, the author’s first name, surname and writing alias, the number of people following the author and the number of people reading, have read and want to read the book. This query was explained within the documented design section. This can only ever give a single result, as book IDs are unique. If the list that this returns contains one tuple, the book was found, and the data for it retrieved. If it doesn’t contain anything, the book was not found. Consequently, after performing the query, it checks whether the list returned by the query contains any items. If it does not, it throws the BookNotFoundError, as the desired book is not in the database. An error is thrown instead of a return value, as it is clearer and simpler to check for. When this is thrown, it should return a 404 Error, as the book about page does not exist. If there are items, it then takes the first tuple in the list from the query, as it is then quicker and easier to access. It can be done safely, as again the book ID is unique so there can only ever be one tuple in it.  It then finds the author’s name, using the function from the authors file, of the components module. It takes the author’s first name, surname, and alias and packages them into a single string. This is so that the values in the dictionary can be put directly into the webpage, without any additional client-side processing, a sentiment that is continued throughout this method. The resulting values in the dictionary can be used without any processing client-side.  Next, it gets a list of the top genres that match the book, the number was given when the object was instantiated, within the num\_display\_genres parameter. It does this by performing a database query, which was explained in the documented design, and then flattens this by using a list comprehension, taking the first item in each of the tuples in the list returned by the database query. This creates a list of genre names that can be directly added to the resulting dictionary.  All the values that have been found at this point are then made into a dictionary, including the title, cover image, synopsis, purchase link, release date, author name, the author’s about, the author’s number of followers, the number of followers the author has, the number of people who want to, have, and are currently reading the book, the list of genre names, and the author ID. the author’s name is that calculated previously, as the display-ready string. Both the synopsis and author\_about require additional processing. It adds the HTML <p> paragraph tag to the start and end of the string, and then splits at each new line, creating a list. This list is then joined back up, with the end of the last paragraph being finished with the </p> tag ending, and the start of the next with a new <p> tag opening. This converts a string, into paragraphs that can directly be inserted into the html, including the line spaces. If it does not include new lines, it does not have any effect, other than adding the tags to the start and end of the string. This needs to be done, as the webpage does not show the Python new line character (\n), and doing it here avoids further client-side processing. The release date also needs processing, as it is returned from the query as a datetime object, so the strftime method converts it into a string in the form dd/mm/yyyy, excluding the time it was created, which is also within the datetime object.  Following this, it then gets the review breakdown, performing the query, explained in the documented design, getting the average rating, rounded to two decimal places, number of reviews, and the number of reviews for each of the 5 available stars. This will always give a result, because of the count giving 0, if there are not any reviews, so the first, and only, tuple in the list can be taken. From this, the average rating, number of ratings, and number of ratings for five, four, three, two, and one stars can also be added to the dictionary.  Next it checks whether a user has left a review. If the user ID parameter is None, the user cannot have left a review, so the item is added to the dictionary as None. However, if it is not None, there could be a review. It then queries the reviews table, to get the review data for the user, using the query explained in the documented design. If there is a result, there is a review for the current user. Consequently, if the length of the result is 0, the current user’s review is also set to None. However, if there is a result, the current review is inserted into a dictionary, which is set as the current user’s review. It checks wtherh the body is None, as it is not required for a review, and then treating it the same way as a paragraph would raise errors, so it needs to be checked. If it is None, it is kept as None, but if it is a string, it has the <p> tags added to it, in the same way as before. The dictionary contains the review ID, the overall rating, plot rating, character rating, summary and the body, following the processing (if it was a string).  It then gets all of the reviews in the database, using the query explained in the documented design. These will be in the reverse order, as they are added  This dictionary that is created contains all of the data for the book’s about page. This can then be returned. | | |
| **The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book**  **Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system.**  This method helps to meet these objectives, as it gets all of the stored data about the book, including the title, author, genres, synopsis, cover image, average rating, the current user’s rating and a purchase link. This information is then used to create a dictionary, which contains all of the required information. It can then be converted to a JSON, sent to the client, and then used to construct the about page. Consequently, in conjunction with the JavaScript, meets these objectives. | | |
| **A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website.**  This objective is partially met. A link to buy the book is taken from the database, and added to the dictionary, which can then be added to the link button, which then takes the user to the external website to buy the book. This method achieves this part of the objective, in combination with the JavaScript.  The current price is not shown. This is not stored within the database, as it would need to be updated frequently as it would be likely to change frequently. It must be webscraped, however, due to time constraints with the project, this was not able to be completed. As a result, this objective is only partially met. | | |
| **Reviews should be listed on the corresponding book about page.**  **Reviews on the book should show the user’s username.**  This method helps to meet these objectives, as it gets all of the reviews left for that book. These are then packaged into a dictionary, which can be converted to a JSON, sent to the client, and inserted into the webpage, so the reviews are listed on the corresponding about page, and therefore helps meet the first objective. This review data also includes the username of the user who left the review, so, in combination with the JavaScript, the second objective is met. | | |

| **Delete a review** | | |
| --- | --- | --- |
| Users should be able to delete any reviews that they have left | Met |  |
| def delete\_review(self, review\_id, user\_id):  self.\_connection.query("""  DELETE FROM reviews  WHERE user\_id={user\_id}  AND review\_id={review\_id};  """.format(user\_id=user\_id, review\_id=review\_id)) | | |
| This procedure deletes a review. It takes both the user ID and the review ID. This would work correctly with just the review ID, but needing both the user ID and review ID to be correct makes it harder to arbitrarily delete reviews by guessing IDs, which should improve security. It executes the query, explained in the documented design. It is done as a method instead of using the query there, as it is simpler to call the method, and the query is abstracted away. | | |
| **Users should be able to delete any reviews that they have left**  This method helps meet this reuirement. It deletes a specific review, identified by review ID, and removes it from the database through a MySQL query. This removes the recommendation from the project. | | |

| **Leave a review** | | |
| --- | --- | --- |
| Users should be able to leave reviews about each book | Met |  |
| Reviews can have a character rating, plot rating, review summary, and review body. | Met |  |
| def leave\_review(self, user\_id, book\_id, overall\_rating, plot\_rating, character\_rating, summary, thoughts):  params = locals()  params = {i: "null" if k is None else k for i, k in zip(params.keys(), params.values())}  # Convert all None parameters to null for insertion into query.  if thoughts is not None:  params["thoughts"] = '"' + re.sub("\n+", "\n", params["thoughts"]) + '"'  params["summary"] = '"' + params["summary"] + '"' # There is a  # check to ensure that 'thoughts' cannot be given without  # 'summary'.  self.\_connection.query("""  DELETE FROM reviews  WHERE book\_id={book\_id}  AND user\_id={user\_id}  """.format(book\_id=book\_id, user\_id=user\_id)) # This will remove  # any existing reviews, so there will only ever be one review per  # book per user  self.\_connection.query("""  INSERT INTO reviews (user\_id, book\_id, overall\_rating, plot\_rating, character\_rating, summary, rating\_body) VALUES  ({user\_id}, {book\_id}, {overall\_rating}, {plot\_rating}, {character\_rating}, {summary}, {rating\_body});  """.format(  user\_id=params["user\_id"],  book\_id=params["book\_id"],  overall\_rating=params["overall\_rating"],  plot\_rating=params["plot\_rating"],  character\_rating=params["character\_rating"],  summary=params["summary"],  rating\_body=params["thoughts"]  )) | | |
| This procedure adds a new review to the database for a book. It takes the user ID, book ID, overall rating, plot rating, character rating, summary and thoughts (rating body) as the parameters. The user ID, book ID, overall rating and summary cannot be None. However, the other parameters can be. These need to be inserted directly into the query ideally, but when they are inserted using .format, they are not inserted as null, which is what is required for MySQL. To do this, it gets the values of all the local variables at the start of the function’s execution. It then iterates through this, using a generator to create a new dictionary. It replaces None value with “null”, and keeps any other values the same. This is simpler to do than an switch statement, and is also easier to adapt to taking more parameters in the future.  To ensure that it is as robust as possible, leaving a review for a book removes any existing review for the user. While this shouldn’t be possible, as the option to leave a review should be hidden if a review exists, it is not impossible that there is a scenario where this could occur, or could be changed to be shown through the debug console by the user, so it is possible for a review to be left when there is one existing. The old review is then deleted from the database, using the query explained in the documented design, using the user ID and book ID. Doing this ensures that there can only ever be one review per book, per user.  If the thoughts (rating body) was given, which is checked by seeing whether it is None, it carries out a process on the data. It removes any multiples of new lines, so there is only ever at most one empty line in the review at a time. It also adds the quotation marks to the rating body string so that it can be directly inserted into the query. This is also done to the summary field, as there is a check to ensure that the summary has been given when a body has been given in the javascript.  The query is then executed, with the values of the recalculated parameters dictionary used over the passed parameters. | | |
| **Users should be able to leave reviews about each book**  This method helps to meet this objective, as it takes the all of the information given by the user, and then stores it within the database. This records the review, and saves it, linking the book, user and review together. It can then be viewed by any other user, and be deleted by the user who wrote it. Therefore, it, in conjunction with the JavaScript, meets this objective. | | |
| **Reviews can have a character rating, plot rating, review summary, and review body.**  This method helps to meet this objective, as it allows for the character rating, plot rating, review summary, and review body to be left empty. However, the other fields are still required for the function, and the query, so consequently meets objective. The fields that cannot be left blank are checked in the JavaScript to ensure that they are given, and that the review is left correctly. | | |

| **Get the highest rated books in the database** | | |
| --- | --- | --- |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website | Met |  |
| def get\_highly\_rated(self):  res = self.\_connection.query("""  SELECT books.title,  books.book\_id,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias,  AVG(reviews.overall\_rating) AS average\_rating  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  INNER JOIN reviews ON reviews.book\_id=books.book\_id  GROUP BY books.book\_id  ORDER BY average\_rating DESC  LIMIT {}  """.format(self.\_number\_summaries\_home)) # The number of summaries  # on the genre page should be the same as the layout is the same  output\_dict = dict() # priority queue  for i, k in enumerate(res):  output\_dict[i] = {  "author": components.authors.names\_to\_display(k[3], k[4], k[5]),  "title": k[0],  "book\_id": k[1],  "cover": k[2],  }  return output\_dict | | |
| This function gets the highest rated books in the database. It performs the database query, as explained in the documented design, which gets the highest rated books. It gets number as specified by the number\_similartires\_home parameter given when the object was instantiated, by inserting this number into the limit clause within the query, using the string .format() method. It then takes the results of this, and converts it into a dictionary of dictionaries, in the same way as in the get\_newest() function. Each of the subdictionaries contains the data required for a book summary. | | |
| **There should be a browse page on the website to display popular books, highest rated, and the newest additions the website**  This method helps to meet this objective, as it gets the highest rated books from the database, using an MySQL query. The results of this query are then taken, and converted into a dictionary of dictionaries, containing the book summary information. This dictionary can then be taken, converted into JSON, sent to the client and inserted into the browse page, to add the highest rated books to the browse page. Therefore, this, in conjunction with the JavaScript and WSGI, meet this objective. | | |

#### Diaries (./backend/components/diaries.py)

The diaries part of the components module has functions and procedures that relate to all of the operations required for the user’s reading diaries. This includes leaving and deleting diary entries, and getting lists of diary entries for a user. The module includes a single class, which contains methods for the diaries. It has been created as a class, so it can have a database connection object stored within it, which can then be used within all of the associated functions, without having to have an object passed into the function each time. It takes the object as a parameter, rather than creating it within the class to reduce the number of database connections used by the project, as some database servers only allow a certain number of connections, which would be used up quickly if each object had its own connection, within each of the multiple workers running. Passing the object allows for each worker to function off of a single a connection, which should avoid this issue. The class follows the subsequent class definition, which was not given in the documented design as it is a collection of functions, and only made into a class due to the limitations of the server, and is also part of the construction of the module.

| **Diaries** |
| --- |
| * connection (object) |
| * add\_entry * delete\_entry * get\_entries |

| **Diaries class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection):  self.\_connection = connection | |
| This is the constructor for the diaries class. It takes a database connection object as a parameter, from the custom mysql\_handler module. This allows the subsequent method to query the database without needing to have a connection object passed each time, and also means a connection does not need to be created for the class, which reduces the number of database connections in use. | |

| **Add a diary entry** | | |
| --- | --- | --- |
| Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating. | Met |  |
| Users must specify the number of pages that they read. | Met |  |
| def add\_entry(self, user\_id, book\_id, overall\_rating, character\_rating, plot\_rating, summary, thoughts, pages\_read):  params = locals()  params = {i: "null" if k is None else k for i, k in zip(params.keys(), params.values())}  if thoughts is not None:  params["thoughts"] = '"' + re.sub("\n+", "\n", params["thoughts"]) + '"'  if summary is not None:  params["summary"] = '"' + params["summary"] + '"'  self.\_connection.query("""  INSERT INTO diary\_entries (user\_id, book\_id, overall\_rating, character\_rating, plot\_rating, summary, thoughts, pages\_read)  VALUES  ({user\_id}, {book\_id}, {overall\_rating}, {character\_rating}, {plot\_rating}, {summary}, {thoughts}, {pages\_read});  """.format(  user\_id=params["user\_id"],  book\_id=params["book\_id"],  overall\_rating=params["overall\_rating"],  character\_rating=params["character\_rating"],  plot\_rating=params["plot\_rating"],  summary=params["summary"],  thoughts=params["thoughts"],  pages\_read=params["pages\_read"]  )) | | |
| This procedure creates a new diary entry for a user. It takes the user ID, book ID and the overall rating as parameters, which cannot be None. It also takes the character rating, plot rating, summary, and entry body (thoughts) as parameters, which can be None.  As with the method to leave a review, from the books object, this uses a generator to replace the None values given in the parameters with a string, “null”, so it can be inserted directly into the query. It then adds quotation marks to the thoughts parameter, and summary parameter, if they were specified (if the passed values are not None), so that they can be inserted directly into the MySQL query.  These processed values are then inserted directly into the query, given in the documented design, using the .format() method, before executing the query. The query does not include the quotation marks for the string values, as they could be null, and therefore must only be added if it is known not to be null. | | |
| **Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating.**  This method takes all of the fields required for the diary entry, as well as the optional fields. These are then stored within the database, so that they can then be accessed in the future, so consequently helps meet this objective. Combining this with the JavaScript, WSGI and HTML, it meets this objective, as it gets the data from the user through the form, processes it, and executes this method to add the record to the database. | | |
| **Users must specify the number of pages that they read.**  This method helps to meet this objective, as it takes the pages read, and inserts it into the database, so it is recorded. The JavaScript ensures that the field is not left blank, so this method, in combination with the JavaScript meets this objective. | | |

| **Delete a diary entry** | | |
| --- | --- | --- |
| Users should be able to delete diary entries | Met |  |
| def delete\_entry(self, user\_id, entry\_id):  # The user id is just a way of helping preventing a random deletion  # of a list. The corresponding user\_id must be known.  self.\_connection.query("""  DELETE from diary\_entries  WHERE user\_id={user\_id}  AND entry\_id={entry\_id};  """.format(user\_id=user\_id, entry\_id=entry\_id)) | | |
| This procedure deletes a single one of a user’s diary entries. It takes the user ID of the user, and the entry ID of the diary entry. The user ID is not necessary, but using it within the query makes it harder to guess entry IDs, and arbitrarily delete records. It inserts these values into the MySQL query, as explained in the documented design, and then executes it. This deletes the user’s diary entry. | | |
| **Users should be able to delete diary entries**  This method helps to meet this objective, as it takes an entry ID, and the user ID, and uses this to remove the record from the database, which deletes the specified diary entry. Between this, WSGI, JavaScript, and the HTML, this objective is fully met. | | |

| **Get a user’s diary entries** | | |
| --- | --- | --- |
| The users’ diary entries must be available on a dedicated webpage on the website. | Met |  |
| def get\_entries(self, user\_id):  res = self.\_connection.query("""  SELECT diary\_entries.entry\_id,  diary\_entries.book\_id,  diary\_entries.overall\_rating,  diary\_entries.character\_rating,  diary\_entries.plot\_rating,  diary\_entries.summary,  diary\_entries.thoughts,  diary\_entries.date\_added,  diary\_entries.pages\_read,  books.cover\_image,  books.title,  authors.author\_id,  authors.first\_name,  authors.surname,  authors.alias,  (SELECT IFNULL(ROUND(AVG(reviews.overall\_rating), 2), 0)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_rating  FROM diary\_entries  INNER JOIN books ON books.book\_id=diary\_entries.book\_id  INNER JOIN authors ON books.author\_id=authors.author\_id  WHERE diary\_entries.user\_id={}  """.format(user\_id))  queue = data\_structures.PriorityQueue(priority\_func=lambda x: x[7])  # Order by date added  for i in res:  queue.push(i)  output\_dict = dict()  for i in range(queue.size):  k = queue.pop()  author = components.authors.names\_to\_display(  k[12],  k[13],  k[14]  )  thoughts = k[6]  if thoughts is not None:  thoughts = "</p><p>".join(("<p>" + k[6] + "</p>").split("\n"))  output\_dict[i] = {  "entry\_id": k[0],  "book\_id": k[1],  "overall\_rating": k[2],  "character\_rating": k[3],  "plot\_rating": k[4],  "summary": k[5],  "thoughts": thoughts,  "date\_added": k[7].strftime("%d-%m-%Y"),  "pages\_read": k[8],  "cover\_image": k[9],  "title": k[10],  "author\_id": k[11],  "author\_name": author,  "average\_rating": float(k[15]),  "number\_ratings": k[16]  }  return output\_dict | | |
| This function gets a dictionary of dictionaries, each containing a dictionary containing all of the information for each of the diary entries. It performs a database query, explained in the documented design, to get all of the data for the users diary entries. For each of the user’s diary entries it gets the entry ID, book ID, overall rating, character rating, plot rating, entry summary, entry body (thoughts), date added, number of pages read, the book cover, title, the author’s name, the average rating and the number of ratings. It inserts the user ID into the where clause of the query, using the .format() method on the string, as it performs well, and is very simple to do.  These results will be given in the time order, with the  The dictionary that this creates acts as a priority queue, with the items indexed from, with item 0 being the most recent diary entry. It can be returned, which contains all of the data for all of the specified user’s results. | | |
| **The users’ diary entries must be available on a dedicated webpage on the website.**  This method helps to meet this objective, as it gets all of a specific user’s diary entries. It takes all of the data stored about each of the entries, and uses this to construct a dictionary of dictionaries. The sub-dictionaries then contain all of the data to display for each diary entry. The output dictionary can be converted into a JSON object, which can be processed by the javascript to display all of the user’s entries on the dedicated diary page. | | |

#### Genres (./backend/components/genres.py)

The genres part of the components module has functions and processes to handle all of the required tasks associated with genres within the website. This includes getting the about data for the genre, and getting a genre ID from a name. This module includes custom exceptions, and a genres class, which handles all of the required processing, which requires a database connection. It has been done as a class, as it has an associated mysql\_handler connection object passed into it, which avoids it needing to be passed into each function, every time they are used, as would be the case with functions instead of a class. This also helps make the functions easier to call, as they have fewer parameters required to work correctly. The genres class follows the following class diagram, which was excluded from the documented design as it is forms part of the construction of the module.

| **Genres** |
| --- |
| * connection (object) |
| * get\_about\_data * id\_to\_name |

| **Genre not found exception** | |
| --- | --- |
| class GenreNotFoundError(Exception):  def \_\_init\_\_(self, genre\_name):  message = f"Genre '{genre\_name}' was not found"  super().\_\_init\_\_(message) | |
| This custom exception is for when the about data is retrieved for a genre, using the genre’s name, but it is not within the database. This provides a quick and easy way to tell if an error has occurred, which can be caught and handled in a simple and readable way. In these cases a 404 status code should be sent to the client, as the page they are requesting cannot be generated, as the genre does not exist. When using the module outside of the script, it also gives a clear error message, specifying which genre was not found, which should help with debugging, as the error is very clearly explained, complete with the line it occurred at, and the description of the error. | |

| **Genres class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection):  self.\_connection = connection | |
| This is the constructor for the genres class. It takes a mysql\_handler connection object as a parameter, which allows the genre methods to query the database without needing to have a connection object passed each time, and also means a connection does not need to be created for the class, which reduces the number of database connections in use, which could be an issue with some database servers, where there is a limited number of simultaneous connections allowed. | |

| **Get a genre’s about data** | | |
| --- | --- | --- |
| Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| It should display the genre’s name | Met |  |
| It should show an about summary for the genre, if applicable | Met |  |
| It should show the books that are part of the genre | Met |  |
| def get\_about\_data(self, genre\_name):  res = self.\_connection.query("""  SELECT genre\_id, name, about FROM genres  WHERE name="{genre\_name}";  """.format(genre\_name=genre\_name)) # There will only be one entry  # with that name, so take only tuple from result list  if len(res) == 0: # Protect against a list out of range errors  raise GenreNotFoundError(genre\_name)  else:  res = res[0]  count = self.\_connection.query("""  SELECT CEIL(COUNT(book\_id) \* 0.15)  FROM book\_genres  WHERE genre\_id={};  """.format(res[0]))[0][0]  db\_books = self.\_connection.query("""  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM books  INNER JOIN authors ON books.author\_id=authors.author\_id  INNER JOIN book\_genres ON books.book\_id=book\_genres.book\_id  INNER JOIN genres ON genres.genre\_id=book\_genres.genre\_id  WHERE genres.genre\_id={genre\_id}  ORDER BY book\_genres.match\_strength DESC  LIMIT {count};  """.format(genre\_id=res[0], count=count))  book\_dict = dict()  for i, k in enumerate(db\_books):  book\_id, title, cover, first\_name, surname, alias = k  author = components.authors.names\_to\_display(  first\_name,  surname,  alias  )  book\_dict[i] = {  "id": book\_id,  "title": title,  "author": author,  "cover": cover  }  output\_dict = {  "name": res[1],  "about": "</p><p>".join(("<p>" + res[2] + "</p>").split("\n")),  # Split each paragraph into <p></p> elements  "books": book\_dict  }  return output\_dict | | |
| This function gets the about data for a specific genre, which is specified by the genre name being passed as a string to the function. First, it takes the genre name, and performs a database query to get the corresponding genre ID, name, and the about data from the genres table, limiting it to a single genre using the where clause, matching the genre name. The genre name passed into the function is inserted into the query, before executing, using the .format() method on the string. This query only gives a maximum of one result, as genres are unique. For this reason, the contents of the res variable is either an empty list, or a list containing a single three-element tuple. Therefore, checking the length of this list gives whether the query had a result, and consequently if the target genre exists in the database. If the list is empty, it throws the GenreNotFoundError, as the genre is not in the database, and consequently the about data cannot be found. This also stops the execution of the function, so there is not any subsequent, unnecessary processing. However, if there is an item in the result, the query result is reassigned to the only tuple within it, which makes the data in it simpler and faster to access, as fewer indexes need to be accessed.  Following getting the explicit genre about data from the database, it needs to get the books that are for that genre. It first performs a database query that a number which finds the how many 15% of the books in the database is. It gets the number of books that match the target genre, using the genre ID, specified through the where clause. This value is then multiplied by 0.15 to get 15% of this value. However, this is could be a decimal number and could not, therefore be used in the limit clause, and would also allow for it to give a result of 0, which should not be possible if there are any books written for the genre. If there are not any books matching it, it should still give 0. The ceiling function is then used to round the decimal value up to the next whole integer, which ensures that the number is not a decimal so can be used within a limit clause, that if there is any books matching the genre at least one will be shown, and if there are not any books for the genre, 0 will be shown.   | SELECT CEIL(COUNT(book\_id) \* 0.15)  FROM book\_genres  WHERE genre\_id=<genre id>; | | --- |   This query will only ever give a single result, which will always be given, because if there are not any books for the used genre ID, it will give a result of 0. This means that the number can be extracted by taking the first, and only value from the only tuple from the list. It needs to be used because of the way the matrix factorisation finds the matches for the genres. Matrix factorisation will give every book a non-zero rating for almost every genre in the database, so only a percentage of the matches can be used. 15% was taken as a reasonable percentage given the 250 books used within the test dataset.  Next, it performs a query to find the book summaries for all of the books for the specific genre. It is given in the documented design, however, has been modified slightly so that they are ordered by match strength, so the limit clause can be used to get the best 15% of matches, as calculated from the previous query. The number of books found from the previous query is inserted into the limit clause, and the genre ID is inserted into the where clause before executing the query using the .format() method, as it is fairly efficient, and very readable.  The result of this query is then iterated through, and a dictionary of dictionaries is created, where the key is the point in the priority queue, and the value is a dictionary containing the summary data for the corresponding book. The book with key 0 in the dictionary is the book with the strongest match to the genre. This is done in the same way as was performed on …  Finally an output dictionary can be created, containing the genre name, the about paragraph, and the dictionary of the book summary dictionaries. The about paragraphs need to be formatted, in the same way as the book synopses in the books part of the module, by adding the paragraph (<p>) tags to the start and end of the string, and splitting at the line breaks to add further tags so that it can be directly inserted into the webpage, and be displayed with the correct paragraphs, as it the Python line breaks do not show as paragraphs on the webpage. This dictionary can then be returned. | | |
| **Each genre stored in the database should have a dedicated about page, which displays all the available information within the system.**  **It should display the genre’s name**  **It should show an about summary for the genre, if applicable**  **It should show the books that are part of the genre**  This method helps to meet this objective, as it gets all of the available data for each genre from the database. It does this by getting the top 15% of books for that genre, the name and the about blurb from the database, and packaging it into a dictionary. This dictionary can then be used to construct the webpage through the JavaScript, and therefore meeting the objective. The data included in this dictionary also then means that it meets the rest of objectives. | | |

| **Get a genre name from an ID** | |
| --- | --- |
| def id\_to\_name(self, genre\_id):  return self.\_connection.query("""  SELECT name FROM genres  WHERE genre\_id={}  """.format(genre\_id))[0][0] | |
| This function gets a genre name from an ID. It performs a database query which gets the name, by specifying the ID through the where clause. The ID passed to the function is inserted into the query before executing it using the .format() method. After the result is executed, the only item in the only tuple in the resulting list is taken and returned.  This assumes that there is a result to the query, and by extension, that the ID being used is correct. It does not contain any checking for the genre existing, and therefore does not throw the relevant error, and if it does not exist would throw an IndexError, as items would be being accessed from the list which do not exist within it. This is not an issue, however, as this function is only ever used after it has already been checked that the genre already exists, so checking whether the genre exists would be unnecessary, and would just increase the time complexity of any of the processes it is used in. | |

#### Information retrieval and database searching (./backend/components/information\_retrieval.py)

The information retrieval part of the components module has functions and procedures that handle all of the processes required to search the database. This includes generating unique words, term frequency (TF) values, inverse document frequency (IDF) values, and searching the database. It includes a single class, and associated functions.

| **Clean a string** | |
| --- | --- |
| def clean\_data(string):  return "".join([i.lower() for i in string if i.isalnum() or i == " "]) | |
| This function cleans a string, so that it is in a form that can be used for the TF-IDF search. It takes a string as a parameter, which is to be cleaned. It works by using a list comprehension to create a list of the cleaned values. It iterates through every character within the passed string, converting it to lowercase, and only adding it to the resulting list if it is either an alphanumeric character (letter or a number), or if it is a space. This list is then taken, and merged back into a string, joining each character within the list together. By doing this, the function converts a string to lowercase, and also removes any character that is not a letter, number or a space. This string can then be returned, and is the cleaned version of the string.  If, in the future, more features were to be added to the search, this function is the one that would need to be modified. Spell checking, stemming or lematisation, removal of stop words, removal of contractions and converting any numbers to strings would need to be performed within this function. Modifying the function would alter both how the clean strings are stored within the database, and how the search query from the client would be cleaned too. | |

The majority of the functions and processes that need to be performed for the database searching are within the DocumentCollection class. It has been done as a class, as many of the functions need to have access to the database connection object, books, authors and genres objects, constants, and data stored within memory. Doing this as a class allows for each of procedures and functions to access the data stored within memory, which makes using the functions faster and simpler. The class follows the following class diagram, which was excluded from the documented design, as it was originally intended to be a collection of functions, without having lots of data stored in memory.

| **DocumentCollection** |
| --- |
| * connection (object) * books (object) * authors (object) * genres (object) * result\_limit * idf\_values * documents\_dict * documents |
| * load\_documents\_dict * gen\_unique\_words * gen\_tf\_values * num\_documents\_containing * gen\_idf\_values * get\_idf\_values * gen\_tfidf\_values * tfidf\_search * database\_search |

| **DocumentCollection object constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, books, authors, genres, result\_limit):  self.\_authors = authors  self.\_genres = genres  self.\_books = books  self.\_result\_limit = result\_limit  self.\_connection = connection  self.load\_documents\_dict()  self.gen\_tf\_values()  self.\_idf\_values = None | |
| This is the constructor for the DocumentCollection class. It takes a mysql\_handler connection object, books, authors, and genres objects from the components module, and a result\_limit. The handler connection is taken, so that queries can be made within the class, and to ensure that each WSGI worker can use a single database connection, to keep the number of active database connections as low as possible. The books, authors, and genres objects are given to the database so that associated data can be retrieved from the database using their methods. Finally, the result\_limit parameter (integer) specifies the number of results that should be given from a single search. Each of these parameters are then assigned to private variables, so that they can be used throughout the object’s functions. They are stored as private variables, as they do not need to be accessed from outside the class, and any unintentional alterations could significantly effect the object, for example the result\_limit should only be changed through changing the configuration file.  Following the assignment of the parameters, it the loads the documents into memory, using the load\_documents\_dict procedure. It creates a list of clean titles, and a list of dictionaries, containing clean titles, IDs, and the type the item is. These can then be used within the TF-IDF search at any point. Following this, it generates the TF values for all of the documents. Doing this initially does increase the time it takes before the WSGI script can handle requests. However, this allows for the performance once the object has been created to be higher, which is more important than the initial start-up times. Furthermore, it is single-threaded, which therefore means that it must be loaded initially, for it not to have a impact on the websites performance. | |

| **Load all the documents into memory** | |
| --- | --- |
| def load\_documents\_dict(self):  self.\_documents\_dict = []  self.\_documents = []  res = self.\_connection.query("""  SELECT books.clean\_title,  books.book\_id,  authors.clean\_name  FROM books  INNER JOIN authors ON authors.author\_id=books.author\_id  """)  for title, book\_id, author\_name in res:  new\_title = title + " " + author\_name  self.\_documents\_dict.append({  "type": "b",  "words": new\_title,  "id": book\_id,  "similarity": 0  })  self.\_documents.append(new\_title)    for title, genre\_id in self.\_connection.query("SELECT clean\_name, genre\_id FROM genres"):  self.\_documents\_dict.append({  "type": "g",  "words": title,  "id": genre\_id,  "similarity": 0  })  self.\_documents.append(title)  for title, author\_id in self.\_connection.query("SELECT clean\_name, author\_id FROM authors"):  self.\_documents\_dict.append({  "type": "a",  "words": title,  "id": author\_id,  "similarity": 0  })  self.\_documents.append(title) | |
| This procedure loads the documents into memory. It creates a list of all the clean titles, as the private documents variable, and a list of dictionaries, containing the type of the item, the words in the title, the ID of the item, and the similarity to the search query. It creates both, as, while less memory efficient, it is significantly faster to use. This is because searching through the list of dictionaries is slower than the list of titles, but also allows data to be stored in both. Furthermore, the indexes of the each item in both lists will remain the same, as they are appended in the same iteration.  For both the authors and genres, it performs the same process. It queries the database, getting the ID, and the clean name. It then iterates through these results, appending a dictionary to the documents\_dict, containing the clean title, the ID, and the similarity. The similarity starts as 0, because there has not been a search to compare to. For the genres, the type is also added, as a “g”, and for the authors, it is added as a “a”. For the books, it gets the ID, clean title, and the author’s clean name from the database, and then adds the author’s clean name on the end of the books clean title, before setting it to the words. The ID is also added, the similarity is set to 0, and the type is set to “b”. Adding the author’s name to the title allows for books to be given when the author’s name is searched for. For each of these, the clean title is also appended to the documents list, regardless of whether it was a book, author, or genre. | |

| **Generate the unique words** | |
| --- | --- |
| def gen\_unique\_words(self):  words = list(itertools.chain(\*[i["words"].split(" ") for i in self.\_documents\_dict]))  unique\_words = set(words)  self.\_connection.query("DELETE FROM unique\_words")  values = ""  for i in unique\_words:  if i != "":  if values != "":  values += ","  values += f'("{i}")'    self.\_connection.query(f"INSERT INTO unique\_words (word) VALUES {values}") | |
| This method finds all the unique words used within all of the searchable documents. It does this by iterating through the documents, and splitting the strings into individual words. It then flattens the resulting two dimensional list, into a single list. This list is then converted into a set, which removes any duplicate values.  The existing unique words are deleted from the database, so the newly calculated ones replace them. It then iterates through the set to create a comma separated lists of the words, each surrounded in quoteation marks and brackets, creates a query from this, containing each of the words, and inserts them into the unique words table in the database. | |

| **Generate the TF values** | |
| --- | --- |
| def gen\_tf\_values(self, term=None):  if term is None:  for count, document in enumerate(self.\_documents\_dict):  tf = dict()  arr = document["words"].split(" ")  one\_over\_n = 1 / len(arr)  for i in arr:  if i != "":  if i in tf:  tf[i] += one\_over\_n  else:  tf[i] = one\_over\_n  self.\_documents\_dict[count]["tf"] = tf  else:  arr = term.split(" ")  tf = dict()  one\_over\_n = 1 / len(arr)  for word in arr:  if word != "":  if word in tf:  tf[word] += one\_over\_n  else:  tf[word] = one\_over\_n  return tf | |
| This method calculates the term frequency values, either for all of the documents, or for a specific set of terms. The terms parameter defaults to None, and in those situations it gets the term frequencies for all of the strings within the database. However, if a string is provided, it calculates the term frequencies for that string instead.  When the parameter has not been specified, the term variable is None. This is then checked for, and if it is None, it calculates the TF values for every document. It iterates through every document in the document dictionary. Then for each iteration, it splits the clean title up, and for each of the words, increases the term frequency by the reciprocal of the number of terms each time. It checks whether it is already within the dictionary, as if it is, it can be increased. However if it is not already within the dictionary, it needs to be set instead, so that errors are not thrown. This dictionary can be added to the dictionary for the current document. Given that the for loop iterates through every document, it calculates the TF values for every document. The dictionary that is produced for each of the documents has each word as a key, and the TF value as the corresponding value in it.  If the parameter has not been specified, the variable will not be None. In this situation, it performs the same process as before, just without iterating through all the documents. It splits the given terms up into a list. It then iterates through each of the words in the terms, and increases the corresponding word in the term frequency by the reciprocal of the number of words in the string. Having iterated through all the terms, the produced TF dictionary can then be returned. | |

| **Get the number of documents containing a specific string** | |
| --- | --- |
| def num\_documents\_containing(self, string):  return sum(string in i for i in self.\_documents) | |
| This function gets the number of occurrences of a string in the documents. It uses a generator to iterate through each of the documents, and checks if the string is in the document. These values are then summed together using the sum function. In Python, sum can be used on True and False values, with True being treated as a 1, and False being treated as a 0, which is why this generator can work. Finally, it returns the number of occurences. | |

| **Generate the IDF values** | |
| --- | --- |
| def gen\_idf\_values(self):  num\_documents = len(self.\_documents)  self.\_idf\_values = dict()  for word\_id, word in self.\_connection.query("SELECT word\_id, word FROM unique\_words"):  idf = math.log10(num\_documents / self.num\_documents\_containing(word))  self.\_connection.query("""  UPDATE unique\_words  SET idf\_values={idf}  WHERE word\_id={word\_id}  """.format(  idf=idf,  word\_id=word\_id  ))  self.\_idf\_values[word] = idf | |
| This procedure generates the IDF values, and stores them in the database. It first calculates the number of documents, as this will remain constant throught the iterations, and will be significantly faster to access from memory than recalculating it. It also creates an empty dictionary to act as the a lookup for the idf values. It then gets all the words and their corresponding IDs from the database, and iterates through each of them.  For each word and ID pair, it calculates the IDF value by dividing the number of documents by the number of occurrences the word has in all the documents, and performing a log on the result, as shown in the formula given in the analysis. This corresponding record in the database is then updated, setting the IDF value to the calculated one for the current word ID. It also adds the IDF value to the dictionary, with the word as the key, and the IDF value as the corresponding value. This allows the IDF values to be stored in the database, and removes the need for them to be calculated | |

| **IDF Values getter** | |
| --- | --- |
| @property  def idf\_values(self):  if self.\_idf\_values is not None: # Should be faster as it only  # needs to be fetched from the DB once  return self.\_idf\_values  else:  self.\_idf\_values = dict()  for word, idf in self.\_connection.query("""SELECT word, idf\_values FROM unique\_words"""):  self.\_idf\_values[word] = idf  return self.\_idf\_values | |
| This is the getter method for the IDF values dictionary. This is not a standard getter method, as it is a property. This means it can be accessed like a variable normally, and it runs this code instead. It cannot be reassigned however, as there is not a setter method given. It uses a private variable, as this should be used instead.  If the private variable has already been changed from its initial value of None, set in the constructor, the data has already been retrieved from the database, so it can be returned directly. However, if it has not been changed from None, the values need to be retrieved from the database, and loaded into memory. To do this, it creates an empty dictionary for the values. It then gets all the words, and there corresponding IDF values from the unique\_words table from the database, and iterates through all of the pairs, adding them to the dictionary, with the word as the key, and the IDF value as the corresponding value. This dictionary is then also returned.  This approach is faster than loading it from the database every time, and also avoids the penalty of increasing the start-up times for the webserver. | |

| **Generate TF-IDF values** | |
| --- | --- |
| def gen\_tfidf\_values(self, document=None, search\_terms=None):  if document is None:  for count, document in enumerate(self.\_documents\_dict):  document\_words = document["words"].split(" ")  if search\_terms is None:  new\_search\_terms = document\_words  else:  new\_search\_terms = search\_terms    res = {i: 0 for i in new\_search\_terms}  for i in res.keys():  if i in self.idf\_values and i in document\_words:  res[i] = document["tf"][i] \* self.idf\_values[i]    self.\_documents\_dict[count]["tfidf"] = res  else:  document\_words = document.split(" ")  tf = self.gen\_tf\_values(document)  res = {i: 0 for i in document\_words}  for i in res.keys():  if i in self.idf\_values and i in document\_words:  res[i] = tf[i] \* self.idf\_values[i]  return res | |
| This method generates the TF-IDF values | |

**Perform a TF-IDF search**

| **Perform a TF-IDF search** | |
| --- | --- |
| def tfidf\_search(self, terms):  terms = clean\_data(terms)  term\_arr = terms.split(" ")  search\_tfidf = self.gen\_tfidf\_values(document=terms)  result = []  self.gen\_tfidf\_values(search\_terms=term\_arr)  for document in self.\_documents\_dict:  similarity = a\_total = b\_total = 0 # These are used to work out  # the magnitude of the vectors  tfidf = document["tfidf"]  for k in term\_arr:  similarity += search\_tfidf[k] \* tfidf[k]  a\_total += search\_tfidf[k] \*\* 2  b\_total += tfidf[k] \*\* 2    if similarity > 0:  similarity /= (math.sqrt(a\_total) \* math.sqrt(b\_total))  document["similarity"] = similarity  result.append({  "type": document["type"],  "similarity": document["similarity"],  "id": document["id"]  })    return sorted(result, key=lambda x: (-x["similarity"], x["type"]))  # Sort by similarity descending and type ascending.  # This puts authors above books if the rating is the same. Order  # would be authors -> books -> genres, if the certainty for all of  # them is the same | |
|  | |

| **Search the database** | | |
| --- | --- | --- |
| There must be a search bar that searches for books, authors and genres within the website. | Met |  |
| The search bar should be able to search by name or ISBN number | Met |  |
| def database\_search(self, search):  output\_dict = dict()  addition = 0 # If an isbn result is added, the initial value will  # be one larger, so would need to be increased by 1  if search.isnumeric():  try:  output\_dict[0] = self.\_books.get\_summary(isbn=search)  output\_dict[0]["type"] = "b"  output\_dict[0]["certainty"] = 100.0 # Set certainty to  # 100% (1 d.p) as it is an exact match  addition = 1  except components.books.BookNotFoundError:  pass  search\_result = self.tfidf\_search(search)  for count, res in enumerate(search\_result[:self.\_result\_limit - addition]):  if res["type"] == "b":  temp = self.\_books.get\_summary(res["id"])  temp["type"] = "b"  output\_dict[count + addition] = temp  elif res["type"] == "a":  temp = {  "name": self.\_authors.id\_to\_name(res["id"]),  "type": "a",  "author\_id": res["id"]  }  output\_dict[count + addition] = temp  else:  temp = {  "name": self.\_genres.id\_to\_name(res["id"]),  "type": "g"  }  output\_dict[count + addition] = temp  output\_dict[count + addition]["certainty"] = round(res["similarity"] \* 100, 1) # Convert similarity to percentage  # (1 d.p)    return output\_dict | | |
| This function performs a complete database search. It uses the simple TF-IDF search method, and gets the correct information to display, including the unclean titles/names, and IDs. It also includes a search by the ISBN number, limiting the number of search results, and calculating the certainty of the result.  To do this, it creates an output dictionary, which is used as a queue of the results. First, it checks whether the search has been performed by ISBN number, and as they are unique, this will only ever add a single extra result to the list of books. The addition variable is used to ensure that the number of results is respected. It starts at 0, and if a result is given for the ISBN match, it is increased to 1, so then 1 fewer results can be given for the TF-IDF search. ISBN numbers are numeric, so therefore, if the search string is numeric, it is most likely to be done by ISBN. If it is numeric, it tries to get the summary, using the books module, using the ISBN number. If it does not exist, it throws the BookNotFoundError, which is then caught. However, if it is found, it sets the type to a book (b), and sets the certainty to 100. The certainty can be assumed to be 100%, as it is an exact match. Finally, the addition variable is set to one so the number of results given by the search respects the limit, and can tell where to start adding items into the output dictionary.  Having checked whether it is an ISBN, it then performs the basic TF-IDF search, using the tfidf\_search function, limiting the results to the specified limit, given in the constructor, minus the addition variable, to ensure that it is respected. It then iterates through the results of this operation. If the item is book, it gets the summary data, using the function from the books object given when the object was instatiated. It adds the type (b) to this dictionary, and then assigns it to the next empty location in the output dictionary, which is the addition variable, plus the iteration number. The addition needs to be considered, as if an ISBN was found, it would have already used the key, 0, so key 1 would be the next to use, However, if it the item is an author type, it gets the authors name, using the ID to name method from the authors object, given when the object was instantiated. It also sets the type (a), and gives the author ID, before assigning it to the next available location, as with the books. Finally, if it was a genre, it uses the method from the genre object, passed when the object was instantiated, to get the name from the ID, and also sets the type, before assigning to the dictionary.  Finally, it calculates the certainty, by multiplying the similarity decimal (between 1 and 0) by 100, and rounding it to 1 decimal place to give the similarity. It does not need to matter about two results rounding to the same value, as they are already ordered using a much greater precision in the simple tfidf\_search function. Following this the dictionary can be returned to give the search results, with all the information required to display it correctly on the webpage. | | |
| **There must be a search bar that searches for books, authors and genres within the website.**  This method helps to meet this objective, as it takes a search query, and gets the top results, that are the most similar to it. It includes books, authors and genres within this search. It also then uses the results to create dictionaries, which are then returned. This output can then subsequently be sent to the client to be displayed using the JavaScript. Therefore, between this method and the javascript, this objective is fully met | | |
| **The search bar should be able to search by name or ISBN number**  This method meets this objective, as it checks whether the search given is an ISBN number, and if it is, returns that specific result. However, if it is not, it returns the normal result from a TF-IDF search, so consequently this objective is fully met. | | |

#### Reading lists (./backend/components/reading\_lists.py)

The reading lists part of the components module has functions and procedures that handle all of the related tasks. This involves creating new reading lists, adding entries, deleting entries, deleting lists and getting the items within a reading list. The module includes a class, and associated exceptions. The class handles and performs all the processes that require a database connection. It uses a class, again as it allows parameters to be given once, and reduce the number of parameters for function calls, making it easier to use. It also reduces the number of required database connections by taking it as an object, rather than creating its own instance. This class follows the following class diagram, which again was not included within the documented design as it is part of the construction of the module.

| **ReadingLists** |
| --- |
| * connection (object) * recommendations (object) * number\_summaries\_home * num\_display\_genres |
| * get\_popular * get\_list\_id * get\_names * get\_currently\_reading * get\_want\_read * get\_names\_check\_book\_in * get\_values * remove\_entry * add\_entry * move\_entry * remove\_list * create\_list * get\_most\_recent\_read * get\_newest\_addition |

| **Custom Exception** | |
| --- | --- |
| class ListNotFoundError(Exception):  def \_\_init\_\_(self, list\_name, user\_id):  message = f"User with id '{user\_id} does not have a list called {list\_name}."  super().\_\_init\_\_(message) | |
| This custom exception is for when data is queried about a specific reading list, but the target user does not have a list with that name. This provides a very simple and readable way to tell if an error occurs, and send a relevant status code, most likely a 404. It also helps when debugging the functions as, when it is not caught, it outputs the target list name, and the user that was trying to find it for. | |

| **ReadingLists class constructor** | |
| --- | --- |
| def \_\_init\_\_(self, connection, number\_summaries\_home, num\_display\_genres, recommendations):  self.\_recommendations = recommendations  self.\_connection = connection  self.\_number\_summaries\_home = number\_summaries\_home  self.\_num\_display\_genres = num\_display\_genres | |
| This is the constructor for the ReadingLists class. It takes a mysql\_handler Connection object, the number of summaries to show on the home page, the number of genres to displayed for a book, and recommendations object from the recommendations part of the components module.  It passes a database connection as a parameter so that the methods within the class can query the database, without needing to pass an object into each method. Taking it as a parameter, rather than creating a new instance within the class reduces the number of database connections, helping to allow the entire WSGI file to only use a single database connection. This is important as some database servers, including the one for this project, have a maximum number of connections which can be quickly exceeded across the multiple instances of the WSGI script. Furthermore, only using a single database connection does not have a negative impact on performance, as each WSGI file is single threaded, having multiple connections would not improve performance at all.  The recommendations object is also passed in to the object. This allows the associated methods to be used within the functions. It is needed so that when an item is added to a reading list, any recommendations for that book are then removed. This prevents recommendations being found for any books that the user is already aware of. | |

| **Get the most popular books** | | |
| --- | --- | --- |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. | Met |  |
| def get\_popular(self):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias,  COUNT(books.book\_id) as num  FROM books  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_lists  ON reading\_lists.book\_id=books.book\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  GROUP BY books.book\_id  ORDER BY num DESC  LIMIT {};  """.format(self.\_number\_summaries\_home))  output\_dict = dict()  for i, k in enumerate(res):  output\_dict[i] = {  "author": components.authors.names\_to\_display(k[3], k[4], k[5]),  "title": k[1],  "book\_id": k[0],  "cover": k[2],  }    return output\_dict | | |
| This function gets the most popular books. It performs a database query, as given in the documented design chapter, which gets the data required for book summaries. The popularity of a book can be measured by the number of people who are currently reading a book, and the higher this number is, the more popular the books is. This query gets the books that are being read the most at the time the query was performed.  The limit is set before the query is executed, by inserting the number\_summaries\_home variable, given in the constructor into the limit clause. This is done using the .format() method as it is fast, and readable. It uses a variable, rather than a hardcoded value, as it allows for the number of summaries to be adjusted, depending on the performance of the hardware, and the desired page loading times. This is because increasing this value increases the amount of data that needs to be sent to the client, and may not be beneficial. However, this has a default value in the configuration, but it can be altered if the server owner wants to be able to display more summaries on larger displays/windows.  The results from the query are then iterated through, and a dictionary of dictionaries created. The outer dictionary acts as a queue, with key 0 being the first summary, and therefore the most popular book. This is done in the same way as was performed within the get\_author\_favourite\_data() from the authors part of the components class. The output dictionary can then be returned after this has been performed. | | |
| **The home page should give users a list of trending books, which have the highest number of users currently reading them.**  This method helps to meet this objective, as it gets the books from the database which the most people are currently reading (identified by being in the Currently Reading list). These can then be taken as the trending books. The book summary data for these is then added to a dictionary and then returned. Following this function’s execution, the dictionary can be converted to JSON, and sent to the client, to be inserted into the home page. | | |

| **Get the list ID from user ID and list name** | |
| --- | --- |
| def get\_list\_id(self, list\_name, user\_id):  res = self.\_connection.query("""  SELECT list\_id  FROM reading\_list\_names  WHERE user\_id={user\_id}  AND list\_name="{list\_name}";  """.format(user\_id=user\_id, list\_name=list\_name))  if len(res) == 0:  raise ListNotFoundError(list\_name, user\_id)    return res[0][0] | |
| This function gets the list ID from the list name, and a user ID. It performs a database query, explained in the documented design, which gets the list ID, by user ID and list name. It inserts the user ID and list name into the query before executing it, using the .format() method, as it was simple to use, and is readable. This needs to use both the user ID and the list name to find the ID, as users can have reading lists called the same thing, so the user ID must be known to distinguish between them. A clear example of this is the system-generated reading lists, which are created for every user when they sign up to the website. All the user’s lists are called the same thing, but the IDs must be kept separate, as otherwise the users would be able to access the data from each other’s lists.  The list names should be unique within a user’s lists. Therefore this query can only ever return one or zero results. If a result is given, the list ID has been found, and therefore the list exists. However, if there are not any results, the list cannot exist. The length of the result is checked, and if it is 0, it throws the ListNotFound error, as the target list does not exist for the specified user.  If the error was not thrown, the ID can be returned. The list of tuples returned from the query, will therefore only contain one, single-element tuple, which can be safely accessed, as it has already been checked as to whether it exists or not, and cannot cause an IndexError. This value can be returned. | |

| **Get the names and list IDs for a specific user** | |
| --- | --- |
| def get\_names(self, user\_id):  res = self.\_connection.query(  """  SELECT list\_id, list\_name FROM reading\_list\_names  WHERE user\_id={};  """.format(user\_id)  ) # List of single element tuples  output\_queue = data\_structures.Queue()  for i in res:  output\_queue.push({  "id": i[0],  "name": i[1]  })  return output\_queue | |
| This function generates a queue containing all of the | |

| **Get the books a user is currently reading** | | |
| --- | --- | --- |
| The home page should give users a summary of what is in their currently reading list. | Met |  |
| def get\_currently\_reading(self, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM reading\_lists  INNER JOIN books  ON reading\_lists.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Currently Reading"  AND reading\_list\_names.user\_id={};  """.format(user\_id))  return [{  "author": components.authors.names\_to\_display(i[3], i[4], i[5]),  "title": i[1],  "book\_id": i[0],  "cover": i[2],  } for i in res] | | |
| This function gets a list of dictionaries containing the data for a book summary, for all the books a user is currently reading. It performs a database query to get all the required data for a book summary, for all the books in the *Currently Reading* reading list, for a specific user. The user ID, given as a parameter, is inserted into the query before it is executed, using the .format() method, as it is simple and readable. It also handles converting the type of the user ID to a string so that it can be added into the query.  Once the query has been executed, it is iterated through and constructs a list of dictionaries, with a dictionary per book from the query. The query gives a list of tuples, which can then be constructed into the dictionary, using the names\_to\_display method from the authors part of the components module to take the component parts of the author’s name and stitch them together to form the displayable string. The list is constructed using a list comprehension, as it is considerable faster than using a for loop, and appending to a list. The list that is generated after the list comprehension can then be returned. | | |
| **The home page should give users a summary of what is in their currently reading list.**  This method helps to meet this objective, as it gets the summary data for the most recent additions to a specific user’s currently reading list. This data is retrieved from the database, and then is packaged into a list of dictionaries, which can then be subsequently taken, converted to JSON, and sent to the client for the JavaScript to add the data to the home screen. | | |

| **Get the books a user wants to read** | | |
| --- | --- | --- |
| The home page should give users a summary of what is in their want to read list. | Met |  |
| def get\_want\_read(self, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM reading\_lists  INNER JOIN books  ON reading\_lists.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_list\_names.list\_name="Want to Read"  AND reading\_list\_names.user\_id={};  """.format(user\_id))  return [{  "author": components.authors.names\_to\_display(i[3], i[4], i[5]),  "title": i[1],  "book\_id": i[0],  "cover": i[2],  } for i in res] | | |
| This function works in exactly the same way as the previous, get\_currently\_reading() function works. The database query is subtly different, with the target list name being *Want to Read*, instead of *Currently Reading*. | | |
| **The home page should give users a summary of what is in their want to read list.**  In the same way as for the previous, get\_currently\_reading() function, this gets the data for summaries, and packages them in a list of dictionaries, which can be used by the JavaScript to add the want to read summaries to the home page. | | |

| **Get a list of user’s list names, and whether a specific book is in them** | | |
| --- | --- | --- |
| It should have a method to add the book to the have read, currently reading, and want to read lists. | Met |  |
| def get\_names\_check\_book\_in(self, user\_id, book\_id):  res = self.\_connection.query(  """  SELECT list\_id, list\_name FROM reading\_list\_names  WHERE user\_id={};  """.format(user\_id)  )  lists = dict()  for i, k in enumerate(res):  list\_id, list\_name = k  in\_list = bool(len(self.\_connection.query("""  SELECT book\_id FROM reading\_lists  WHERE list\_id={list\_id}  AND book\_id={book\_id};  """.format(list\_id=list\_id, book\_id=book\_id))))  lists[i] = {  "id": list\_id,  "list\_name": list\_name,  "has\_book": in\_list  }  return lists | | |
| This function creates a dictionary of dictionaries for all of a user’s reading lists, with each sub-dictionary containing the list ID, list name, and whether the specified book (through the book\_id parameter) is in the list. To do this, it first queries the database to get all of the list IDs and the corresponding names for a user. This gets a list of tuples, with one tuple per record returned from the query, where the first item in the tuple is the list ID, and the second is the list name.  It then iterates through this list of tuples. For each of them, it unpacks the tuple, extracting the list ID and list name from it. It then performs another database query, to get the entries where the book ID is in the list that the current iteration is checking for. Taking the length of this (which can be at most 1), and casting it to a boolean gives true, if the book is in the list, and false if it is not. This result then creates a new dictionary, containing the list ID, list name, and whether the specified book is in the list. It is then stored in the outer dictionary, with the key being the iteration number, and the value being the created dictionary.  After all of the iterations, the dictionary can then be returned, as it contains all the data for all of the speicified user’s reading lists. This dictionary acts as a queue, with the key being the dictionaries position in the queue. Acting as a queue means that the lists will be displayed in the order that they were created, which also ensures that the system-defined lists will always be displayed first in the list. This function is required for the book’s about page, as it gets the information for the popup, which allows the user to add books to lists. It needs to know whether the book is already in the list, so the JavaScript can then show the book as in the list, and prevent the user from adding it again. | | |
| **It should have a method to add the book to the have read, currently reading, and want to read lists**  This function helps meet this objective, as it gets all of the lists, and whether the specified book is in them. The JavaScript then uses this data to create a form, which allows for the book to be added to a reading list, as the list ID is known, and it can be prevented from being added to a list if it is already in the list. Therefore, this function meets this objective, in conjuction with the WSGI and the JavaScript. | | |

| **Get the contents of a user’s reading list** | | |
| --- | --- | --- |
| Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
| def get\_values(self, list\_id, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.cover\_image,  books.title,  books.synopsis,  authors.first\_name,  authors.surname,  authors.alias,  authors.author\_id,  reading\_lists.date\_added,  (SELECT GROUP\_CONCAT(genres.name)  FROM book\_genres  inner join books on book\_genres.book\_id=books.book\_id  inner join genres on genres.genre\_id=book\_genres.genre\_id  WHERE book\_genres.book\_id=reading\_lists.book\_id  GROUP by books.title) AS genres,  (SELECT CAST(IFNULL(AVG(reviews.overall\_rating), 0) as FLOAT)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_ratings  FROM reading\_lists  INNER JOIN books  ON books.book\_id=reading\_lists.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  INNER JOIN reading\_list\_names  ON reading\_list\_names.list\_id=reading\_lists.list\_id  WHERE reading\_lists.list\_id={list\_id}  AND reading\_lists.user\_id={user\_id}  ORDER BY reading\_lists.date\_added DESC, books.title ASC;  """.format(  list\_id=list\_id,  user\_id=user\_id # This is not strictly necessary, but  # helps protect against people being able to view other  # people's list contents by guessing the list id. Prevent  # any null values - replace with 0s.  )  )  queue = data\_structures.Queue()  for i in res:  queue.push(i)  list\_name = self.\_connection.query("""  SELECT list\_name FROM reading\_list\_names  WHERE list\_id={};  """.format(list\_id))[0][0] # See which list the button would move  # to.  if list\_name == "Currently Reading":  button = "Mark as Read"  move\_target = self.\_connection.query("""  SELECT list\_id FROM reading\_list\_names  WHERE list\_name="Have Read"  AND user\_id={};  """.format(user\_id))[0][0]  elif list\_name == "Want to Read":  button = "Start Reading"  move\_target = self.\_connection.query("""  SELECT list\_id FROM reading\_list\_names  WHERE list\_name="Currently Reading"  AND user\_id={};  """.format(user\_id))[0][0]  else:  button = None  move\_target = None  output\_dict = dict()  for i in range(queue.size):  k = queue.pop()  author = components.authors.names\_to\_display(k[4], k[5], k[6])  synopsis = "</p><p>".join(("<p>" + k[3] + "</p>").split("\n"))  # Change new lines to new paragraphs  output\_dict[i] = {  "id": k[0],  "cover": k[1],  "title": k[2],  "synopsis": synopsis,  "author": author,  "author\_id": k[7],  "date\_added": k[8].strftime("%d-%m-%Y"),  "genres": k[9].split(",")[:self.\_num\_display\_genres],  "average\_rating": k[10],  "num\_reviews": k[11]  }  return output\_dict, button, move\_target | | |
| This function gets all of the items that are in a specific user’s reading list. To specify the list, it takes the list ID, and the user ID as a parameter. While it can get the entries from list ID alone, taking the user ID helps to introduce an additional level of security, as list IDs cannot just be guessed, the user ID must be known too. Furthermore, due to the use of session IDs, the correct corresponding session ID must be known, as well as the correct list ID to get the user’s data.  First, it performs a database query, which gets all of the data for each of the entries. It is explained in the documented design, and it gets the books ID (so that the it can link to the about page), the cover image, title, synopsis, the author’s name (as component parts), the author ID (so it can link to the about page), the date it was added to the list, the genres, average rating, and the number of ratings. The result of this query is given as a list of tuples, with a tuple per result of the query, and each tuple having one element per field in the query result. Each result of the query is for an entry within the reading list.  Each of these results is then pushed into a queue, so that the order of the results is preserved. This is because the output from the query is in date-order, so ensures that they are then sent to the client in the correct order later.  Following this, it gets the list name of the target list, by performing another database query, by getting the list name from the database, matching the list ID passed into the function. It then takes the first element of the first tuple, as this should be the list name. This will throw an exception if it is not a valid list ID given, but as it is specified from the client, with IDs that are sent directly from the server, and are known explicitly, this does not need to be checked for. The list name needs to be found to establish what the button to move entries to different lists should say, assuming that it is needed, and the list ID of the destination after the move. If the list name of the list that is being accessed is ‘Currently Reading’, the button should say ‘Mark as Read’, so it sets the button variable to that string. It also gets the list ID for the ‘Have Read’ list, by performing a database query. This can also take the first element from the first tuple, as the user ID must exist, and therefore the ‘Have Read’ list must also exist as it is created automatically. This is then assigned to be the target list ID of the move. However, if the list name is ‘Want to Read’, it sets the button text to ‘Start Reading’, as the user should be able to move a book to the ‘Currently Reading’ list from the ‘Want to Read’ list. It then gets the target list ID, in the same way as it did for the ‘Mark as Read’ button, except for the ‘Currently Reading’ list. The value given by this is saved as the move target. Finally, if it is any other list (not ‘Currently Reading’ or ‘Want to Read’), it sets both the button and move target variables as None, as a clear indication that they are not needed, and the button should not be displayed.  Finally, it creates an empty dictionary, to contain the data for all of the diary entries. It then iterates through the queue, running the number of times as there are items within the queue. In each iteration, it then gets the tuple from the original database query, and uses this to create another dictionary, containing the data for each entry. It creates the author’s display name from the author’s name components, and also converts the newlines in the book’s synopsis to HTML paragraph tags, so that it can be displayed as paragraphs on the webpage. It also converts the datetime object given for the date added in the query into a string date, in the form dd-mm-yyyy. Furthermore, it converts the string of genres into a list, and taking the first number, which is specified by the num\_display\_genres parameter in the object constructor, through using slicing on the list. These values are then made into a dictionary, which is then assigned to the output dictionary, taking the iteration number as the key. This makes the key be the index of the item in the output, which can then be treated as a queue.  Having iterated through all of the items in the queue, it can return the output dictionary, button text, and the target list ID for the destination, when the move button is pressed. A dictionary was chosen as the output over a list, as it allows additional values to be added to it. This is easier when the result is converted to a JSON. The entries in the reading list can be accessed as if it were a list, as the keys are indexes, derived from the iteration number, which works in the same way a list would. It can also add additional data to the dictionary later, using a non-numerical value as the key, which can also be accessed alongside the entries. This can make it simpler to use later. | | |
| **List slicing**  This function uses list slicing to limit the number of genres given to the constant. It uses list slicing to get the first n items from the list, which is then stored in the dictionary for each entry | | |
| **Aggregate MySQL query**  This funcion uses aggregate MySQL queries as part of the subqueries in the main query. This is because it uses the average function, and the count function, as part of the two subqueries. They are used to get the average rating for the book for each entry, and the number of ratings given for the book for each entry. | | |
| **Queues**  This function uses queues. It pushes each entry from the main query into a queue to ensure that the date order of the results is maintained. The dictionary that is returned is also a queue, as it uses the index to record where each sub-dictionary is within the queue. This can then be used in order to display the reading list entries in date order. | | |
| **Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list**  This function helps meet this objective, as it gets the all of these fields from the database, and creates a dictionary of results from it, which then is returned. This dictionary can be converted to JSON, and sent to the client, which is used by the JavaScript to take the entry data, and display them, including the cover image, title, genre, author name, average rating, and the date added, and therefore meets this objective. | | |
| **Reading lists, and the entries within them should be specific to the user.**  This function helps to meet this objective, as it is the only way to get the user’s reading list data. It requires the user ID to display them, which ensures that the results shown are specific to the user, and therefore this function helps to meet the objective | | |
| **Each entry into the Currently Reading reading list should supply a method to mark an entry as read and move it to the have read reading list.**  **Each entry into the Want to Read reading list should supply a way to the user to start reading the book and move the entry to the currently reading list.**  This method helps to meet these objectives, as it gives the list ID of the list that the button to move the entry to (Currently Reading from Want to Read, and Have Read from Currently Reading). This means that the entry can be moved to a separate list, from the ID. This ensures that this is simple to do. It also gives the text that should be shown within the button (Start reading from Want to Read, and Mark as Read from Currently Reading). This can then be used with the JavaScript to allow for the users to move the entries, as specified by the entries, and therefore, between this, the WSGI and the JavaScript, these objectives are met. | | |

| **Remove an entry from a reading list** | | |
| --- | --- | --- |
| Each entry into a reading list should allow the user to delete the entry. | Met |  |
| def remove\_entry(self, user\_id, list\_id, book\_id):  self.\_connection.query("""  DELETE FROM reading\_lists  WHERE user\_id={user\_id}  AND book\_id={book\_id}  AND list\_id={list\_id};  """.format(  book\_id=book\_id,  user\_id=user\_id,  list\_id=list\_id  )) | | |
| This procedure deletes a reading list entry. It takes the list ID and the book ID to specify the entry within the list. The combination of list ID and book ID specifies a single entry, as each list can only have a book appear in it once. The user ID is required to help with security, as it makes it harder to delete the entry without permission, as both the list ID, book ID, and session ID must be known. These values are inserted into a database query, which was explained in the documented design, to delete the specific entry, and executed. This removes the reading list entry. | | |
| **Each entry into a reading list should allow the user to delete the entry.**  This method helps to meet this objective, as it removes the record from the database, using the user ID, list ID and the book ID to specify the entry. Between these, the specific entry can be identified. This means that, in combination with the JavaScript and WSGI executing and receiving the request, this objective is met. | | |

| **Add an entry into a reading list** | | |
| --- | --- | --- |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| def add\_entry(self, user\_id, list\_id, book\_id):  self.\_recommendations.delete\_recommendation(  user\_id,  book\_id,  bad\_recommendation=False  )  # Delete recommendation when added to a list    lists = {i[0] for i in self.\_connection.query("""  SELECT list\_id FROM reading\_list\_names  WHERE list\_name IN ("Currently Reading", "Have Read", "Want to Read")  AND user\_id={}  """.format(user\_id))}  if list\_id in lists:  self.\_connection.query("""  DELETE FROM reading\_lists  WHERE user\_id={user\_id}  AND book\_id={book\_id}  """.format(book\_id=book\_id, user\_id=user\_id))  # Delete entry from other lists to prevent duplicates  self.\_connection.query("""  INSERT INTO reading\_lists (user\_id, book\_id, list\_id) VALUES  ({user\_id}, {book\_id}, {list\_id});  """.format(  user\_id=user\_id,  book\_id=book\_id,  list\_id=list\_id  )  ) | | |
| . | | |
| **Reading lists, and the entries within them should be specific to the user.**  This procedure helps to meet this objective, as it takes the user ID, and specifies it within the record that it adds to the database. Doing this associates the entry with the specific user, which ensures that the users’ reading list entries remain specific to them. This allows for the entries to be retrieved for a specific user. | | |

| **Move an entry from one reading list to another** | | |
| --- | --- | --- |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
| def move\_entry(self, user\_id, start\_list\_id, end\_list\_id, book\_id):  self.add\_entry(user\_id, end\_list\_id, book\_id) # This changes the  # date added, but this is not an issue as it would be a new  # addition to the list, so the date should change.  self.remove\_entry(user\_id, start\_list\_id, book\_id) | | |
| This procedure moves a reading list entry from one reading list to another. It takes the user ID, book ID and the starting list as parameters to specify the reading list that the entry is originally in. It also takes the end list ID, for the list that the entry should finish in. To do this, it first adds the new entry to the end list ID, with the samme book ID and user ID as passed in. Finally, it removes the old entry from the starting list. Adding the entry to the end list, and removing the entry from the start list, is the same as moving the entry between reading lists. | | |
| **Each entry into the Currently Reading reading list should supply a method to mark an entry as read and move it to the have read reading list.**  **Each entry into the Want to Read reading list should supply a way to the user to start reading the book and move the entry to the currently reading list.**  This procedure helps to meet these objectives, as it takes a book ID, start list, and a user ID to specify and entry, and then an end list ID to specify the final list. It then adds it to the new list, and removes the old entry, which then moves the entry to a different list. This allows entries to be moved from the Want to Read list to the Currently Reading list, and the Currently Reading list to the Have Read list, and therefore, in conjunction with the JavaScript, this method meets these objectives. | | |

| **Delete a reading list** | | |
| --- | --- | --- |
| Users should be able to delete their own custom lists | Met |  |
| def remove\_list(self, user\_id, list\_id):  # Do not need to check whether the list is protected, the delete  # button is hidden by the JS. To delete it would still require  # session id, so cannot be done accidentally.  self.\_connection.query("""  DELETE FROM reading\_lists  WHERE list\_id={list\_id}  AND user\_id={user\_id};  """.format(list\_id=list\_id, user\_id=user\_id))  # Only the specific users list will be deleted, as it targets the  # single list  self.\_connection.query("""  DELETE FROM reading\_list\_names  WHERE list\_id={list\_id}  AND user\_id={user\_id}  """.format(list\_id=list\_id, user\_id=user\_id))  # Delete the list name | | |
| This function deletes a reading list. It takes the user ID to help to improve security, and the list ID of the list to delete. The list ID is known, as it is given directly to the client, so can be used instead of the list name in combination with the user ID. First it deletes all of the entries within the specified list. This needs to be done, otherwise the foreign key constraints between the reading\_lists table and the reading\_list\_names table would be broken. This wouldn’t be able to be performed, so it must remove the entries first. This is done using a database query, which was explained in the documented design chapter. Having deleted the list entries, it can then delete the list. It then performs a second database query, which was also given in the documented design, to delete the reading\_list\_names record. Deleting this record is the part that actually deletes the reading list from the database. | | |
| **Users should be able to delete their own custom lists**  This procedure helps to meet this objective, as it takes the user ID and the list ID, to specify the list to delete, and removes all of the entries in it, and the list. Therefore, in conjunction with the JavaScript sending the request to delete the list, this method meets the objective. | | |

| **Create a new reading list** | | |
| --- | --- | --- |
| Users should be able to create their own custom reading lists | Met |  |
| def create\_list(self, user\_id, list\_name):  self.\_connection.query("""  INSERT INTO reading\_list\_names (user\_id, list\_name) VALUES  ({user\_id}, "{list\_name}")  """.format(user\_id=user\_id, list\_name=list\_name)) | | |
| This procedure creates a new list, and can be used for creating custom, user-defined lists, or the system defined ones. It takes the user ID and the list name, and performs a database query, inserting the new list name into the reading\_list\_names table, for that user. This creates a new list, which then will be shown on the navigation part of the ‘my books’ page on the website. It can then also be subsquently used to insert reading list entries into as well. | | |
| **Users should be able to create their own custom reading lists**  This procedure helps to meet this objective, as it takes the user ID, and the list name, and creates a new record in the database, which creates a new reading list. In combination with the JavaScript getting the user input, and making the request, the WSGI processing the request, and this method, this objective is met. | | |

| **Get the most recent read book for a specific user** | | |
| --- | --- | --- |
| The browse page should display books related to the last read book | Met |  |
| def get\_most\_recent\_read(self, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  INNER JOIN books ON books.book\_id=reading\_lists.book\_id  WHERE reading\_lists.user\_id={}  AND reading\_list\_names.list\_name="Have Read"  ORDER BY reading\_lists.date\_added DESC  LIMIT 1;  """.format(user\_id))  if len(res) > 0:  return res[0]  return None | | |
| This function gets the book ID and title of the most recent book the user has read. It takes the user ID as a parameter, and then performs a database query, as given in the documented design, to get the most recent addition in their ‘Have Read’ reading list. The limit clause within the query, in combination with the order by descending, ensures that it is the most recent addition. Furthermore, the limit ensures that there is only ever one or zero results given by the query. If there is a result given, the user has a most recent read book. The length of the list will then be 1, so it checks the length of the list, and if it is longer than 0, it returns the first tuple in the list, which contains the book ID and the title in that order. If it has 0 items in it, however, it then returns None, as the user has not read any books yet. | | |
| **The browse page should display books related to the last read book**  This function helps to meet this objective, as it performs a database query, which gets the book ID for the most recent book that the user read. It takes the user ID to specify the current user. In combination with the get\_similar\_books method to get the related books, and the JavaScript to add the data to the page, this method meets this objective. | | |

| **Get the newest addition to a reading list (which is not the *Have Read* list)** | | |
| --- | --- | --- |
| The browse page should display books related to the last one added to a reading list | Met |  |
| def get\_newest\_addition(self, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  INNER JOIN books ON books.book\_id=reading\_lists.book\_id  WHERE reading\_lists.user\_id={}  AND reading\_list\_names.list\_name!="Have Read"  ORDER BY reading\_lists.date\_added DESC  LIMIT 1;  """.format(user\_id))  if len(res) > 0:  return res[0]  return None | | |
| This function gets the book ID and the title of the last book that was added to a reading list, which is not the have read list. It works in the same way as the previous function, except the where clause in the query is for where the list name is not ‘Have Read’. | | |
| **The browse page should display books related to the last one added to a reading list**  In exactly the same way as the previous method, get\_most\_recent\_read, this helps to meet the objective. It can be used in conjunction with the get\_similar\_books function, and the JavaScript to add the related books to the page, which meets this objective. | | |

#### Recommendations (./backend/components/recommendations.py)

| **Exception for when a user has not set initial preferences** | |
| --- | --- |
| class NoUserPreferencesError(Exception):  def \_\_init\_\_(self, user\_id):  message = f"User with id {user\_id}, has no preferences"  super().\_\_init\_\_(message) | |
| . | |

| **Plot the curve during training (for testing and model tuning)** | |
| --- | --- |
| def plot\_learning\_curve(model):  import matplotlib.pyplot as plt # This is bad practice, but cannot be  # imported with lighttpd running  linewidth = 3  plt.plot(model.test\_mse\_record, label='Test', linewidth=linewidth)  plt.plot(model.train\_mse\_record, label='Train', linewidth=linewidth)  plt.xlabel('iterations')  plt.ylabel('MSE')  plt.legend(loc='best')  plt.show() | |
| . | |

| **Recommendations class constructor** | | |
| --- | --- | --- |
| Recommendations should be configurable based upon the books and data stored in the database | Met |  |
| def \_\_init\_\_(  self,  connection,  num\_converge\_iters,  hyperparam,  number\_display\_genres,  initial\_recommendation\_mat\_val,  reading\_list\_percentage\_increase,  following\_percentage\_increase,  bad\_recommendation\_value,  minimum\_required\_reviews,  number\_recommendations,  debug=False  ):  self.\_connection = connection  self.\_num\_converge\_iters = num\_converge\_iters  self.\_hyperparam = hyperparam  self.\_num\_factors = len(self.\_connection.query("SELECT \* FROM genres"))  self.debug = debug  self.\_num\_users = len(self.\_connection.query("SELECT user\_id FROM users"))  self.\_num\_books = len(self.\_connection.query("SELECT book\_id FROM books"))  self.\_number\_recommendations = number\_recommendations  self.\_min\_required\_reviews = minimum\_required\_reviews  self.\_initial\_recommendation\_mat\_val = initial\_recommendation\_mat\_val  self.\_reading\_list\_percentage\_increase = reading\_list\_percentage\_increase  self.\_following\_percentage\_increase = following\_percentage\_increase  self.\_bad\_recommendation\_val = bad\_recommendation\_value # This is  # not 0 as the genres may still be applicable, but should still be  # small  self.\_num\_display\_genres = number\_display\_genres  self.test\_mse\_record = []  self.train\_mse\_record = []  self.\_list\_users\_no\_preferences = {i[0] for i in self.\_connection.query(  "SELECT user\_id FROM users WHERE preferences\_set=FALSE")}  # Uses a set as it is faster for 'item in var' operations  # levels of configuration is required as the recommendations need  # to vary depending on the hardware, and user base, such as average  # number of reviews, sparsity of data, and preferences for  # recommendations, which would affect how easily recommendations  # can change.  self.gen\_lookup\_tables() | | |
|  | | |
| **Recommendations should be configurable based upon the books and data stored in the database**  This method meets this objective, as it takes parameters which can then be used to tailor the recommendations, including the number of recommendations, the number of reviews required before the initial preferences can be ignored, the increase to the ratings for a book being in a reading list or whether an author is being followed, the number of WALS iterations to perform, and the model’s hyperparameter. These can be changed through the configuration file, although are given defaults to the test data, so this meets the objective. | | |

| **Train the recommendations model** | | |
| --- | --- | --- |
| The system should generate recommendations that are unique to each individual user’s tastes. | Met |  |
| def fit(self):  train, test, = self.create\_train\_test()  self.\_num\_users, self.\_num\_books = train.shape  self.book\_factors = np.random.random((self.\_num\_books, self.\_num\_factors))  self.user\_factors = np.random.random((self.\_num\_users, self.\_num\_factors))  if self.debug: # Debug is about 10 times slower  self.test\_mse\_record = []  self.train\_mse\_record = []  for i in range(self.\_num\_converge\_iters):  print(f"Iteration {i + 1} of {self.\_num\_converge\_iters} Start")  self.user\_factors = self.wals\_step(train, self.book\_factors)  self.book\_factors = self.wals\_step(train.T, self.user\_factors)  predict = self.predict()  self.train\_mse\_record.append(self.mean\_squared\_error(train, predict))  self.test\_mse\_record.append(self.mean\_squared\_error(test, predict))  print(f"Iteration {i + 1} of {self.\_num\_converge\_iters} End")  return self.test\_mse\_record, self.train\_mse\_record  else:  for i in range(self.\_num\_converge\_iters):  self.user\_factors = self.wals\_step(train, self.book\_factors)  self.book\_factors = self.wals\_step(train.T, self.user\_factors)  self.save\_book\_genres() # Not included in the debug option, as  # it increases time cost, and would likely be rerun a lot to  # find optimum parameters, so is unnecessary. | | |
| . | | |
| **The system should generate recommendations that are unique to each individual user’s tastes.**  This method helps meet this objective, as it trains the recommendations model, which finds the factors for both the users and the books. When doing this, it factors in the user’s activity, which then tailors their genres to their taste. The factors can then be used to generate personalised recommendations, and therefore this, in conjunction with the generate user recommendations method, meets this objective. | | |

| **Write the new genres to the database** | |
| --- | --- |
| def save\_book\_genres(self):  query = "INSERT INTO book\_genres (book\_id, genre\_id, match\_strength) VALUES "  for count, facts in enumerate(self.book\_factors):  # i will be the rating for each the genres.  book\_id = self.book\_lookup\_table[count]  query += ",".join(f"({book\_id}, {self.genre\_lookup\_table[i]}, {strength})" for i, strength in enumerate(facts)) + ","  self.\_connection.query("DELETE FROM book\_genres") # Done here to  # minimise time without data in DB  self.\_connection.query(query[:-1]) | |
| . | |

| **Generate user recommendations** | | |
| --- | --- | --- |
| The system should generate recommendations that are unique to each individual user’s tastes. | Met |  |
| def predict(self):  return self.user\_factors.dot(self.book\_factors.T) | | |
| . | | |
| **The system should generate recommendations that are unique to each individual user’s tastes.**  This method helps meet this objective, as it takes the factors for the books and the users, and generates a matrix, of the expected ratings for each of the books in the database. This can then be used to generate the recommendations, and therefore helps meet this objective. | | |

| **Generate the target review matrix (explicit and implicit data used)** | | |
| --- | --- | --- |
| Recommendations should include whether the user follows an author | Met |  |
| Recommendations should include whether the book is in a reading list | Met |  |
| Recommendations should include the user’s diary entries | Met |  |
| def gen\_review\_matrix(self):  mat = np.zeros((self.\_num\_users, self.\_num\_books))  for user in self.user\_lookup\_table:  user\_id = self.user\_lookup\_table[user]  # Reviews #  reviews = self.\_connection.query("""  SELECT book\_id,  (overall\_rating + IFNULL(character\_rating, overall\_rating) + IFNULL(plot\_rating, overall\_rating)) / 3  FROM reviews  WHERE user\_id={}  GROUP BY review\_id;  """.format(user\_id))  for book\_id, rating in reviews:  used\_book\_id = list(self.book\_lookup\_table.values()).index(book\_id)  # This finds the key for the value stored in the lookup  # table.  # geeksforgeeks.org/python-get-key-from-value-in-dictionary  mat[user][used\_book\_id] = float(rating)  # Initial Preferences #  books = self.\_connection.query("""  SELECT books.book\_id  FROM initial\_preferences  INNER JOIN books  ON books.author\_id=initial\_preferences.author\_id  WHERE initial\_preferences.user\_id={}  GROUP BY books.book\_id  """.format(user\_id)) # Get a user's existing preferences  if len(reviews) <= self.\_min\_required\_reviews:  if len(books):  for i in books:  used\_book\_id = list(self.book\_lookup\_table.values()).index(i[0])  mat[user][used\_book\_id] += self.\_initial\_recommendation\_mat\_val # This is a non-zero value so  # recommendation is made. This is not affected by  # the average preference expressed by all the  # user's selected authors.  else:  continue # The user has not set up any initial  # preferences yet.  elif len(books):  self.\_connection.query("""  DELETE FROM initial\_preferences  WHERE user\_id={}  """.format(user\_id))  # Reading Lists #  lists = self.\_connection.query("""  SELECT reading\_lists.book\_id  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_lists.user\_id={}  GROUP BY reading\_lists.book\_id;  """.format(user\_id))  for i in lists:  used\_book\_id = list(self.book\_lookup\_table.values()).index(i[0])  if mat[user][used\_book\_id] == 0:  mat[user][used\_book\_id] = self.\_initial\_recommendation\_mat\_val  else:  mat[user][used\_book\_id] \*= (1 + self.\_reading\_list\_percentage\_increase)  # Authors following #  following = self.\_connection.query("""  SELECT books.book\_id  FROM author\_followers  INNER JOIN books  ON books.author\_id=author\_followers.author\_id  WHERE user\_id={}  """.format(user\_id))  for i in following:  used\_book\_id = list(self.book\_lookup\_table.values()).index(int(i[0]))  if mat[user][used\_book\_id] == 0:  mat[user][used\_book\_id] = self.\_initial\_recommendation\_mat\_val  else:  mat[user][used\_book\_id] \*= (1 + self.\_following\_percentage\_increase)  # Bad Recommendations #  for book in self.get\_bad\_recommendations(user\_id):  used\_book\_id = list(self.book\_lookup\_table.values()).index(book)  mat[user][used\_book\_id] = self.\_bad\_recommendation\_val  # = is used in case there is a good value here. It should  # be marked as bad.  # Diary entries #  entries = self.\_connection.query("""  SELECT book\_id,  (SUM(overall\_rating) + SUM(IFNULL(character\_rating, overall\_rating)) + SUM(IFNULL(plot\_rating, overall\_rating))) / (COUNT(entry\_id) \* 3)  FROM diary\_entries  WHERE user\_id={}  GROUP BY book\_id;  """.format(user\_id))  for book\_id, rating in entries:  used\_book\_id = list(self.book\_lookup\_table.values()).index(book\_id)  mat[user][used\_book\_id] += float(rating) # += is used  # incase there is already a value at that index  return mat | | |
| . | | |
| **Recommendations should include whether the user follows an author**  This method meets this objective, as it increases the weighting given to any books written by any authors that the user’s follow. This increases the impact of the books that the user follows on the recommendations, so this objective is met. | | |
| **Recommendations should include whether the book is in a reading list**  This method meets this objective, as it increases the weighting given to any of the books that are in any of the user’s reading lists. This increases the impact that these books have on the recommendations. This is done as if they are in a reading list, it is a clear expression of interest in the book. | | |
| **Recommendations should include the user’s diary entries**  This method meets this objective, as it increases the weighting for any books that diary entries have been left for, by increasing it by the average of all the ratings left in all the entries for that book. This is done, as it is a clear expression of interest in the books. | | |

| **Create the train/test data** | |
| --- | --- |
| def create\_train\_test(self, ratings=None):  if ratings is None:  self.ratings = self.gen\_review\_matrix()  else:  self.ratings = ratings  train = self.ratings.copy()  while self.ratings.tolist() == train.tolist():  train = self.ratings.copy()  for user in range(self.\_num\_users):  nonzero = self.ratings[user].nonzero()[0]  indexes = np.random.choice(  nonzero,  size=round(len(nonzero) \* 0.2),  replace=True  )  for i in indexes:  train[user, i] = 0.0  return train, self.ratings | |
| . | |

| **Weighted alternating least squares (WALS)** | |
| --- | --- |
| def wals\_step(self, ratings, fixed):  A = fixed.T.dot(fixed) + np.eye(self.\_num\_factors) \* self.\_hyperparam  B = ratings.dot(fixed)  A\_inv = np.linalg.inv(A)  return B.dot(A\_inv) | |
| . | |

| **Generate ID lookup tables** | |
| --- | --- |
| def gen\_lookup\_tables(self):  self.user\_lookup\_table = dict()  users = self.\_connection.query("SELECT user\_id FROM users")  for count, i in enumerate(users):  self.user\_lookup\_table[count] = i[0]  self.book\_lookup\_table = dict()  books = self.\_connection.query("SELECT book\_id FROM books")  for count, i in enumerate(books):  self.book\_lookup\_table[count] = i[0]  self.genre\_lookup\_table = dict()  genres = self.\_connection.query("SELECT genre\_id FROM genres")  for count, i in enumerate(genres):  self.genre\_lookup\_table[count] = i[0] | |
| . | |

| **Generate Recommendations** | | |
| --- | --- | --- |
| The system should generate recommendations that are unique to each individual user’s tastes. | Met |  |
| **Recommendation entries should show the certainty of the recommendation** | Met |  |
| def gen\_recommendations(self):  predictions = self.predict()  query = "INSERT INTO recommendations (user\_id, book\_id, certainty) VALUES "  for user, books in enumerate(predictions):  user\_books = []  user\_id = self.user\_lookup\_table[user]  if user\_id not in self.\_list\_users\_no\_preferences:  avoid\_recs = {  i[0] for i in self.\_connection.query("""  SELECT book\_id  FROM recommendations  WHERE user\_id={}  AND date\_added>=DATE\_SUB(NOW(), INTERVAL 2 DAY)  """.format(user\_id))  } # sets are faster for "is val in list" operations  res = self.\_connection.query("""  SELECT book\_id  FROM reading\_lists  INNER JOIN reading\_list\_names  ON reading\_lists.list\_id=reading\_list\_names.list\_id  WHERE reading\_lists.user\_id={}  AND reading\_list\_names.list\_name IN (  "Currently Reading",  "Have Read",  "Want To Read"  )  """.format(user\_id)) # Note that this covers the diary  # entries as well, as entries cannot be made unless it is  # in the have read/currently reading list  for i in res:  avoid\_recs.add(i[0])  for i in self.get\_bad\_recommendations(user\_id):  avoid\_recs.add(i)  for book, rating in enumerate(books):  book\_id = self.book\_lookup\_table[book]  if book\_id not in avoid\_recs:  user\_books.append({  "id": book\_id,  "dot\_product": rating  })  user\_books.sort(key=lambda x: x["dot\_product"], reverse=True)  user\_books = user\_books[:self.\_number\_recommendations]  for count, i in enumerate(user\_books): # Done after as  # this is faily expensive, to avoid unecessary  # calculations  user\_books[count]["certainty"] = self.calculate\_certainty(  i["id"],  user\_id,  i["dot\_product"]  )  query += ",".join(  f"({user\_id}, {i['id']}, {i['certainty']})" for i in user\_books[:self.\_number\_recommendations]) + ","  self.\_connection.query("""  DELETE FROM recommendations  WHERE date\_added<=DATE\_SUB(NOW(), INTERVAL 2 DAY)  """)  self.\_connection.query(query[:-1]) | | |
| . | | |
| **The system should generate recommendations that are unique to each individual user’s tastes.**  This method meets this objective, as it takes the predicted weightings for each of the users, and finds the highest expected weightings in it, which the user is not already aware of. They are then added to the database, so they are stored as new recommendations, whilst also removing the oldest ones. Consequently, it generates new recommendations for each user, which are personalised to their individual tastes, and hence meets this objective. | | |
| **Recommendation entries should show the certainty of the recommendation**  This method helps to meet this objective, as for each recommendation that is found, the certainty is calculated and stored in the database. | | |

| **Delete a recommendation (with an option to impact future recommendations)** | | |
| --- | --- | --- |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. | Met |  |
| Recommendation entries should have a method to move it to the *want to read* reading list. | Met |  |
| def delete\_recommendation(self, user\_id, book\_id, bad\_recommendation=True):  self.\_connection.query("""  DELETE FROM recommendations  WHERE user\_id={user\_id}  AND book\_id={book\_id}  """.format(  user\_id=user\_id,  book\_id=book\_id  ))  if bad\_recommendation:  self.\_connection.query(  "INSERT INTO bad\_recommendations (user\_id, book\_id) VALUES ({user\_id}, {book\_id})".format(  user\_id=user\_id,  book\_id=book\_id  )  ) | | |
| . | | |
| **Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation.**  This method helps to meet this objective, as it deletes the recommendation from the database, and takes an optional parameter, which also records it in the bad recommendations table, which marks it as a bad recommendation. This, in combination of the JavaScript which makes the request, and the WSGI that handles it, meets this objective. | | |
| **Recommendation entries should have a method to move it to the want to read reading list.**  This method helps to meet this objective. This is because when the recommendation is moved to the Want to Read reading list, it needs to be removed from the recommendations. It removes the recommendations, and the bad\_recommendations optional parameter should be set to False, so that it does not mark it as a bad recommendation, and won’t have a negative impact on the recommendations. | | |

| **Get recommendations that have explicitly been marked as bad** | |
| --- | --- |
| def get\_bad\_recommendations(self, user\_id):  bad\_recommendations = self.\_connection.query("""  SELECT recommendation\_id,  book\_id,  date\_added  FROM bad\_recommendations  WHERE user\_id={}  AND date\_added<=DATE\_SUB(NOW(), INTERVAL 2 DAY)  """.format(user\_id))  existing\_recommendations = self.\_connection.query("SELECT book\_id FROM recommendations WHERE user\_id={}".format(user\_id))  existing\_recommendations = {i[0] for i in existing\_recommendations}  # Sets are faster for 'in' operations  reading\_list\_items = self.\_connection.query("""  SELECT reading\_lists.book\_id  FROM reading\_lists  WHERE reading\_lists.user\_id={}  """.format(user\_id))  return\_vals = []  remove = []  for rec\_id, book, date in bad\_recommendations:  if date + datetime.timedelta(weeks=10) > datetime.datetime.now():  # 10 week expiry, so it can start recommending books if the  # user's preferences have changed. 10 weeks is a long  # enough time for it to be plausible to be a good  # recommendation  return\_vals.append(book)  else:  remove.append(rec\_id)  self.\_connection.query(  "DELETE FROM bad\_recommendations WHERE recommendation\_id IN ({})".format(",".join(str(i) for i in remove)))  # Delete expired recommendations.  return return\_vals | |
| . | |

| **Get the user’s recommendations** | | |
| --- | --- | --- |
| Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre. | Met |  |
| Recommendation entries should show the certainty of the recommendation | Met |  |
| def get\_user\_recommendations(self, user\_id):  items = self.\_connection.query("""  SELECT recommendations.book\_id,  ROUND(recommendations.certainty \* 100, 1) as certainty,  recommendations.date\_added,  books.cover\_image,  books.synopsis,  books.title,  authors.first\_name,  authors.surname,  authors.alias,  authors.author\_id,  (SELECT GROUP\_CONCAT(genres.name) FROM book\_genres  INNER JOIN genres ON book\_genres.genre\_id=genres.genre\_id  WHERE book\_genres.book\_id=recommendations.book\_id  GROUP BY books.book\_id) AS genres,  (SELECT ROUND(CAST(IFNULL(AVG(reviews.overall\_rating), 0) as FLOAT), 2)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS average\_rating,  (SELECT COUNT(reviews.overall\_rating)  FROM reviews  WHERE reviews.book\_id=books.book\_id) AS num\_ratings  FROM recommendations  INNER JOIN books ON recommendations.book\_id=books.book\_id  INNER JOIN authors ON books.author\_id=authors.author\_id  WHERE recommendations.user\_id={}  ORDER BY recommendations.certainty DESC;  """.format(  user\_id)) # ORDER BY does not use calculated certainty for  # higher accuracy, and avoiding collisions IFNULL prevents any null  # values - replace with 0s.  if len(items) == 0 or user\_id in self.\_list\_users\_no\_preferences:  raise NoUserPreferencesError(user\_id)  output\_dict = dict()  for i, k in enumerate(items):  author = components.authors.names\_to\_display(k[6], k[7], k[8])  output\_dict[i] = {  "book\_id": k[0],  "certainty": k[1],  "date\_added": k[2].strftime("%d/%m/%Y"),  "cover\_image": k[3],  "synopsis": "</p><p>".join(("<p>" + k[4] + "</p>").split("\n")),  "title": k[5],  "author\_name": author,  "author\_id": k[9],  "genres": k[10].split(",")[:self.\_num\_display\_genres],  "average\_rating": round(k[11], 2),  "number\_ratings": k[12]  }  return output\_dict | | |
| . | | |
| **Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre.**  This method helps to meet this objective, as it gets all of the data available within the database, which includes the cover image, author, book’s synopsis, average rating, the date recommended, and the books genres. This returns a dictionary, which can then be converted to JSON later, and sent to the client, where the JavaScript fills out the webpage with the recommendation data. Consequently, this method, in combination with the JavaScript, meets this objective. | | |
| **Recommendation entries should show the certainty of the recommendation**  This method helps to meet this objective, as it gets the certainty from the database, and includes it in the dictionary that is returned from the function. Then, like the other objective, it can be inserted and displayed on the webpage using the JavaScript. Consequently, it meets the objective, in combination with the JavaScript. | | |

| **Get a list of summary data for a user’s recommendations** | | |
| --- | --- | --- |
| The home page should give users a summary of their recommended books. | Met |  |
| def get\_user\_recommendation\_summaries(self, user\_id):  res = self.\_connection.query("""  SELECT books.book\_id,  books.title,  books.cover\_image,  authors.first\_name,  authors.surname,  authors.alias  FROM recommendations  INNER JOIN books ON recommendations.book\_id=books.book\_id  INNER JOIN authors ON books.author\_id=authors.author\_id  WHERE recommendations.user\_id={}  ORDER BY recommendations.certainty DESC;  """.format(user\_id))  return [{  "author": components.authors.names\_to\_display(i[3], i[4], i[5]),  "title": i[1],  "book\_id": i[0],  "cover": i[2],  } for i in res] | | |
| . | | |
| **The home page should give users a summary of their recommended books.**  This method helps to meet this objective, as it gets a list of summary dictionaries of the top recommendations that have been generated. The output of this function can then be taken, and converted to a JSON, and sent to the client when it is requested for the home page, and inserted into the webpage using the JavaScript. Consequently, this meets the objective, with the WSGI and the JavaScript. | | |

| **Calculate the certainty of a recommendation** | | |
| --- | --- | --- |
| Recommendation entries should show the certainty of the recommendation | Met |  |
| def calculate\_certainty(self, book\_id, user\_id, dot\_product):  book\_id = list(self.book\_lookup\_table.values()).index(book\_id)  book\_vec = [i for i in self.book\_factors[book\_id]]  user\_id = list(self.user\_lookup\_table.values()).index(user\_id)  user\_vec = [i for i in self.user\_factors[user\_id]]  abs\_book\_vec = math.sqrt(sum(i \*\* 2 for i in book\_vec))  abs\_user\_vec = math.sqrt(sum(i \*\* 2 for i in user\_vec))  similarity = dot\_product / (abs\_book\_vec \* abs\_user\_vec)  if similarity > 1: # Slim chance it ends up larger than 100%, so  # limits it artificially.  similarity = 1  return similarity | | |
| . | | |
| **Recommendation entries should show the certainty of the recommendation**  This method helps to meet this objective, as it takes the book ID, user ID, and the dot product between two unknown vectors. The predicted model only has the dot product, so this takes the IDs to find the vectors and calculate the cosine similarity between the two and returns it. This allows the certainty to be found, and is used when the recommendations are generated to store the certainty, which is then used to display it on the webpage. | | |

| **Add new user to the model using initial author preferences** | | |
| --- | --- | --- |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
| def add\_user(self, user\_id, author\_ids):  vals = [f"({user\_id}, {author\_id})" for author\_id in author\_ids]  self.\_connection.query(  "INSERT INTO initial\_preferences (user\_id, author\_id) VALUES {}".format(  ",".join(vals)  )  )  self.\_connection.query("""  UPDATE users  SET preferences\_set=TRUE  WHERE user\_id={}  """.format(user\_id))  res = self.\_connection.query("""  SELECT AVG(book\_genres.match\_strength),  book\_genres.genre\_id  FROM book\_genres  INNER JOIN books  ON book\_genres.book\_id=books.book\_id  INNER JOIN authors  ON books.author\_id=authors.author\_id  WHERE authors.author\_id IN ({})  GROUP BY book\_genres.genre\_id;  """.format(  ",".join(str(i) for i in author\_ids)  )  )  target\_vec = np.zeros(self.\_num\_factors)  for avg, genre\_id in res:  genre = list(self.genre\_lookup\_table.values()).index(genre\_id)  target\_vec[genre] = avg  rec = target\_vec \* self.book\_factors  output = []  for count, val in enumerate(rec[0]):  output.append({  "book\_id": self.book\_lookup\_table[count],  "strength": val  })  output.sort(key=lambda x: x["strength"], reverse=True)  output = output[:self.\_number\_recommendations]  for count, i in enumerate(output):  output[count]["certainty"] = self.calculate\_certainty(  i["book\_id"],  user\_id,  i["strength"]  )  self.\_connection.query("INSERT INTO recommendations (user\_id, book\_id, certainty) VALUES {}".format(",".join(f"({user\_id}, {i['book\_id']}, {i['certainty']})" for i in output)))  return output | | |
| . | | |

| **Calculate mean squared error** | |
| --- | --- |
| @staticmethod  def mean\_squared\_error(true, pred):  mask = np.nonzero(true)  mse = sklearn.metrics.mean\_squared\_error(true[mask], pred[mask])  return mse | |
| . | |

### Flup server file (./backend/flup.server.fcgi)

This file is the file that is run by the webserver. It uses flup module to run the backend WSGI script. It needs to import the app from the custom WSGI file. This means that the middleware object needs to be instantiated, and stored within a variable called app. Lighttpd will run multiple instances of this file, which means that the WSGI file is also being run multiple times. This is where the need to minimise the number of database connections within the WSGi file comes from, as the file will be run multiple times.

| #!/usr/bin/env python3  from flup.server.fcgi import WSGIServer  from wsgi import app  if \_\_name\_\_ == '\_\_main\_\_':  WSGIServer(app).run() | |
| --- | --- |

### Database maintenance (./backend/maintenance.py)

The maintenance script should be run periodically, such as every night. This script should close any old sessions, remove old recommendations, and generate new ones. It is a simple process, but is essential. It ensures that the database remains at a reasonable size, and also ensures that there will be new recommendations given daily.

| **Object instantiation and imports** | |
| --- | --- |
| # -----------------------------------------------------------------------  # Project imports  # -----------------------------------------------------------------------  import components.accounts  import components.recommendations  import configuration  import mysql\_handler  # -----------------------------------------------------------------------  # Project constants  # -----------------------------------------------------------------------  config = configuration.Configuration(  "./project\_config.conf",  default\_conf\_filename="./default\_config.json"  )  # -----------------------------------------------------------------------  # Database connection  # -----------------------------------------------------------------------  connection = mysql\_handler.Connection(  user=config.get("mysql username"),  password=config.get("mysql password"),  schema=config.get("mysql schema"),  host=config.get("mysql host")  )  # -----------------------------------------------------------------------  # Class instantiation  # -----------------------------------------------------------------------  sessions = components.accounts.Sessions(  connection,  config.get("session\_id\_length")  )  recommendations = components.recommendations.Recommendations(  connection,  config.get("recommendations number\_converge\_iterations"),  config.get("recommendations hyperparameter"),  config.get("number\_display\_genres"),  config.get("recommendations inital\_recommendation\_matrix\_value"),  config.get("recommendations reading\_list\_percentage\_increase"),  config.get("recommendations author\_following\_percentage\_increase"),  config.get("recommendations bad\_recommendations\_matrix\_value"),  config.get("recommendations minimum\_required\_reviews"),  config.get("recommendations number\_recommendations"),  ) | |
| This first imports the required modules, namely the accounts and recommendations from the components module, and the configuration and mysql\_handler. These are then instantiated so that they can be used, using parameters from the configuration files. The sessions class is required to remove expired sessions, and the recommendations is required to generate re-fit the model, and generate new recommendations.  *Note that the dashed lines separating the sections have been reduced in length so they can fit on the line without wrapping.* | |

| **Close expired sessions** | |
| --- | --- |
| for i in sessions.get\_session\_id\_list():  try:  sessions.get\_user\_id(i)  except components.accounts.SessionExpiredError:  pass | |
| This closes all the expired sessions. It gets a list of all the session IDs in the database, and iterates through them, trying to get the corresponding user ID. The get\_user\_id() method closes any sessions if they are expired, so by performing this on all the session IDs will close all expired ones. If the session has expired, it throws the SessionExpiredError, so this needs to be caught so that it does not stop it from executing. | |

| **Generate new recommendations** | |
| --- | --- |
| recommendations.fit()  recommendations.gen\_recommendations() | |
| This generates new recommendations and removes the old ones. It first re-fits the recommendations model to incorporate any new activity from the users since the last recommendations were made, including following new authors, adding books to reading lists, and leaving reviews. Following this it generates new recommendations. The gen\_recommendations() method removes any old recommendations before generating new ones, so this also prevents the recommendations table from increasing constantly. | |

| **Scheduling the maintenance (through command line using Cron)** | | |
| --- | --- | --- |
| Recommendations should be regenerated daily | Met |  |
| sudo apt-get install cron -y  sudo crontab -e  In the opened text file:  0 1 \* \* \* python3 /absolute/path/to/project/backend/maintenance.py | | |
| When these commands are executed in the command line, it sets up a cron job. The first command installs the cron software. It then executes a command to open the text file, which contains all of the cron jobs that are running. The cron software runs a single command at a specified day or interval, which therefore allows the maintenance script to run daily.  The second command will then open a text file for editing, and the string is added to the file. For example, with this project it could be added as 0 1 \* \* \* python3 /home/reuben/projects/NEA/backend/maintenance.py. This then sets the maintenance script to run every day at 1am. 1am was chosen as it would likely be a point when the website would see very low usage. It is important to choose a time when the website is experiencing low usage, as generating recommendations, and closing expired sessions is expensive, during testing, normally using 100% CPU (on a single thread). While this is unlikely to have a significant effect on performance for users, as the FastCGI scripts use multiple threads, it should be avoided as best as possible for the best user experience. 1am is unlikely to have high usage, which is why it was chosen, but the cron job could be altered to improve this, but would be deployment specific. This is not configurable through the configuration file, as it must be done through the command line. Doing this ensures that the recommendations are generated daily, and the sessions are kept to a minimum, without requiring it to be run manually. | | |
| **Recommendations should be regenerated daily**  This meets this objective, as it sets up the maintenance script to run every day. As part of this, it generates new recommendations, so this ensures that new recommendations are generated daily. Consequently this objective is met. | | |

### WSGI Script (./backend/wsgi.py)

| **Module imports, including custom ones** | | |
| --- | --- | --- |
| import json  import urllib.parse  import components.accounts  import components.authors  import components.books  import components.diaries  import components.genres  import components.information\_retrieval  import components.reading\_lists  import components.recommendations  import configuration  import environ\_manipulation  import logger  import mysql\_handler | | |
| Explanation  *Note that the comments dividing each of the sections have been removed here. This isn’t necessary here as it is a short code snippet.* | | |
| Technique 1 | | |
| Why the objective is met | | |

| **Components object instantiation** | | |
| --- | --- | --- |
| Parameters that can affect performance should be configurable | Met |  |
| A default configuration should be used, which can be updated as needed based upon available hardware | Met |  |
| Recommendations should be configurable based upon the books and data stored in the database | Met |  |
| Configurable parameters should be modified through an external configuration file, which is in a clear and readable format | Met |  |
| config = configuration.Configuration(  "./project\_config.conf",  default\_conf\_filename="./default\_config.json"  )  # The json does not need to be user editable, so is not very readable.  connection = mysql\_handler.Connection(  user=config.get("mysql username"),  password=config.get("mysql password"),  schema=config.get("mysql schema"),  host=config.get("mysql host")  )  number\_home\_summaries = config.get("home number\_home\_summaries") # This # is a constant, as is is used multiple times, and will always be faster  # to access as a variable, and otherwise, the get function would be have  # to run during the calling of methods as part of a response.  diaries = components.diaries.Diaries(connection)  genres = components.genres.Genres(connection)  sessions = components.accounts.Sessions(  connection,  config.get("session\_id\_length")  )  authors = components.authors.Authors(  connection,  config.get("number\_display\_genres"),  number\_home\_summaries  )  recommendations = components.recommendations.Recommendations(  connection,  config.get("recommendations number\_converge\_iterations"),  config.get("recommendations hyperparameter"),  config.get("number\_display\_genres"),  config.get("recommendations inital\_recommendation\_matrix\_value"),  config.get("recommendations reading\_list\_percentage\_increase"),  config.get("recommendations author\_following\_percentage\_increase"),  config.get("recommendations bad\_recommendations\_matrix\_value"),  config.get("recommendations minimum\_required\_reviews"),  config.get("recommendations number\_recommendations"),  )  reading\_lists = components.reading\_lists.ReadingLists(  connection,  number\_home\_summaries,  config.get("number\_display\_genres"),  recommendations  )  books = components.books.Books(  connection,  reading\_lists,  config.get("home number\_about\_similarities"),  number\_home\_summaries,  config.get("number\_display\_genres")  )  accounts = components.accounts.Accounts(  connection,  config.get("passwords hashing\_algorithm"),  config.get("passwords salt"), # Stored in the config as binary  config.get("passwords number\_hash\_passes"),  reading\_lists  )  information\_retrieval = components.information\_retrieval.DocumentCollection(  connection,  books,  authors,  genres,  config.get("search number\_results")  ) | | |
| Explanation | | |
| Technique 1 | | |
| **Parameters that can affect performance should be configurable**  This code meets this objective, as it every parameter that can be configured can, and is, included within the configuration file, and the default. The configuration module is used to handle all of these configuration options, and get them from the file. Consequently this is met. | | |
| **A default configuration should be used, which can be updated as needed based upon available hardware**  This meets this objective, as the configuration object from the custom configuration module takes both the custom configuration file, which can be updated based on the available hardware, and preferences. It also takes the default configuration options as a JSON file, which can then be overwritten. | | |
| **Recommendations should be configurable based upon the books and data stored in the database**  This code meets this objective, as the configuration contains parameters for the recommendations, which are given defaults based upon the training dataset. However, they can be overwritten and configured based upon the books and data used within the database, so therefore this objective is met. | | |
| **Configurable parameters should be modified through an external configuration file, which is in a clear and readable format**  This code meets this objective, as the custom configuration module is used to load and get data from the configuration. The module uses a syntax that is clear and readable, and loads the data from this external file. It therefore allows the parameters to modified quickly and easily, so consequently meets this objective. | | |

#### Handlers and Middleware

| **Middleware class** | | |
| --- | --- | --- |
| class Middleware(object):  def \_\_init\_\_(self, routes, log):  self.\_routes = routes  self.\_log = log  self.\_log.output\_message(f"Create Middleware object")  def \_\_call\_\_(self, environ, start\_response):  target\_name = environ\_manipulation.application.get\_target(environ)  self.\_log.output\_message(f"Attempting to redirect to {target\_name} application")  target\_application = self.\_routes.get(target\_name) or ErrorHandler("404 Not Found", log)  return target\_application(environ, start\_response) | | |
| Any request that is made to WSGI is passed through this middleware object. Its purpose is to redirect any requests made to the WSGI file to the appropriate handler, as well as logging its path through the script. This class is based upon the Application class given by (Bard, 2016). The concept for using the dictionary as a lookup, and the \_\_call\_\_ method to redirect the requests through different handlers to the correct functions.  **Constructor**  The constructor for this class takes a dictionary for the routes. Each of the handler names in the URLs that are used are given as keys in the dictionary, and the value is the corresponding handler.    These routes are then stored so that it can redirect the request to the correct handler, by looking up the handler from the dictionary, by the handler name in the URL, and calling it, passing the environ and start\_response to it. This can the passes the processing onto the relevant handler.  It also takes a Logging object from the custom logger class, so that the requests can be tracked. It allows for debug messages to be written to the log, and viewed from outside of the script.  **\_\_call\_\_ method**  The flup server module, when dealing with a request, calls the application like a function. Therefore, the middleware class needs to be able to be called like a function, which is what this enables. When it handles a request, it extracts the target handler name from the URL using the custom environ manipulation module. It then logs a message to the log file, stating that it has redirected to the corresponding handler. It then gets the target application object from the routes dictionary. It uses the .get() method to get the value from the dictionary, and if it does not exist, it sets the target as an ErrorHandler object instead, as the handler cannot exist.  Having found the target handler, it calls it, passing the environ variable and start\_response that were given on to it. This also means that the handler objects need to have a custom \_\_call\_\_ method so that this works as well. The output of calling the handler needs to be the response data, converted into binary. | | |
| **Magic Methods**  This code uses magic methods to allow the object to be called like a function. This enables it to work with flup to handle the requests correctly, and respond to them. | | |
| **Dictionary operations**  This uses the get method to get values from the dictionary, but if it does not exist, it returns a default value instead. It gets the correct handler object from the dictionary, and if the target handler does not exist, it then gives the ErrorHandler instead, as the request URL cannot exist. | | |

##### Handler Class

| **Handler class constructor** | | |
| --- | --- | --- |
| def \_\_init\_\_(self, log):  self.\_routes = {} # There are no routes for the base class - included so the \_\_call\_\_ should still work  self.\_log = log  self.\_log.output\_message("Created " + \_\_class\_\_.\_\_name\_\_ + " instance") # Cannot use  # commas as it the method only takes 2 parameters, and these would  # pass each element as a parameter | | |
| The handler class is designed to process individual requests to the WSGI file. It is the superclass, with each of its child classes then being used to handle a specific set of related requests. This is the constructor for this superclass. It takes a logging object from the custom logger module, so that it can record data from requests, and track the handling of the request, and assigns it to its own private variable within the class, so it can be used later. It also creates an empty dictionary, to be used in the same way as (Bard, 2016) used for the Application/Middleware class. Finally, it outputs the name of the handler that was created to the log file. | | |

| **Get POST parameters from the environ dictionary** | | |
| --- | --- | --- |
| def retrieve\_post\_parameters(self):  try:  body\_size = int(self.\_environ["CONTENT\_LENGTH"])  except ValueError:  body\_size = 0  return self.\_environ["wsgi.input"].read(body\_size).decode("utf-8") | | |
| This function gets the data that is sent in a POST request from the environ variable. This uses part of the example given by (Parsing the Request - Post — WSGI Tutorial, n.d.) to get the data from the environ variable. It only uses the part of the code that gets the post data.  This function gives the request payload as a string. This also includes JSON, so any JSON needs to be converted from a string into a dictionary for the values to be accessed. This cannot be done within this function, however, as there is not any way to tell if the data is JSON or a string. It requires knowledge of the request to know whether the post data needs to be converted from JSON or not.  It also requires the environ dictionary to be saved within a private variable. This is so that it does not need to be passed into this function when it is called. This would also require the environ to be passed into the corresponding function when it is called, which increases the complexity of calling and making the outer functions. Storing the environ dictionary in a private variable each request, instead of passing it into functions helps keep the code simpler and more readable. | | |

| **Get GET request parameters from the environ dictionary** | | |
| --- | --- | --- |
| def retrieve\_get\_parameters(self):  query = self.\_environ.get("QUERY\_STRING")  arr\_dict = urllib.parse.parse\_qs(query) # Returns dictionary of arrays {str: list}  res = {i: arr\_dict[i][0] for i in arr\_dict.keys()} # Convert to dictionary {str: str}  # Use urllib as it handles the non-printable characters – %xx  return res | | |
| This function gets the GET request parameters from the URL. It gets the query string from the environ dictionary, and then converts it from the percent-encoded form to a usable string. The dictionary of lists is then converted to a dictionary of values using a generator expression. This dictionary is then formed of the parameter name as the key, and the value as the corresponding value from the URL. This can then be returned  This, like the POST request method, requires the environ variable given with the most recent request to be stored within a private variable. This allows for these methods to be used without having to pass the environ variable into each of the functions within the handler. | | |

| **Call method for the handler class** | | |
| --- | --- | --- |
|  | | |
| This is the \_\_call\_\_ magic method for the handler class. When the middleware object is called to handle a request, it looks up the handler in the routes dictionary. This then is called like a function. Therefore, the handler class needs to have this call method to handle this, which also finds and executes the correct function for the request.  It has a general catchall exception for all possible exceptions when handling the request. This includes invalid data that throws unexpected, unhandled exceptions. This is not normally good practice, but experiencing any errors because of one request will then have an impact on the others, and after enough exceptions occurring, the webserver will fail. This is because when an exception occurs, each worker script will stop executing, and handling requests. If this then happens enough times, all the worker scripts will stop working. This needs to be avoided, so all exceptions are caught to prevent this, and returns the error handler instead for the 500 error. This allows any requests to treat it as a 500 error, as it returns a 500 status code, but the exception is caught, and does then not stop the script from running.  First it saves the environ dictionary to a private variable within the object. This allows it to be used within any other method within the class, without it needing to be passed. This makes calling it simpler. Furthermore, it assumes that the target function to be executed always take the same parameters. This would mean that all of the functions need to have the environ as a parameter, irrespective of whether it is required.  Following this, it then logs which object was called (with the class name), and the name of the request that is being processed. This helps track the request as it passes through the handler. It then gets the target function name, and assigns it to a variable. This is then logged, to help tracking execution This uses a routes dictionary, as (Bard, 2016) suggested for the Middleware, and assigns the function to a temporary name. If it doesn’t exist, it then creates an ErrorHandler object, and uses the error\_response function instead, to send the relevant error message to the client. This target function is then executed, assuming that they all return the response content, status code, and headers in this format.  If any exception occurs during this time, it then creates an ErrorHandler instance, for a 500 error. The error\_response is then called on this instance, to get the status, response and headers. The status code is already known, as the exception is returned, but as the return values are fixed for the other instances where the handler is used, it needs to be in this form. It uses the same variables for the status code, response and the headers so that the same code can be used to perform the start response, and log the response regardless of whether an error occurs or not.  The start\_response function is then executed, as required for the WSGI specification, as shown by (Bard, 2016), and returns the response in binary, converted using the encode function. It uses yield instead of return, so that it returns an iterable, as required for WSGI to work. Finally, it logs the response, with the status, headers, and response. | | |

##### App/Middleware creation

| **Creation of the application** | | |
| --- | --- | --- |
|  | | |
| This code creates the log, handlers and Middleware class. It first creates the logging object. This sets the debugging variable as the same as the one specified in the configuration file, to enable and disable logging. Following this, it creates a dictionary for each of the handlers. The key is the handler part of the request URL, and the value is a corresponding handler. Instantiating these classes here means that they are persistent, and faster than instantiating them when they are needed.  This routes dictionary is passed into the middleware, along with the log object, before instantiating it. The middleware is assigned to an app variable, as this is what needs to be imported by the flup.server.fcgi file that is run by the webserver. This app is then what is called whenever a request is made to the server, but as it has all of the other objects within it, through aggregation, it handles and redirects the requests to the correct handler, and functions within it. | | |

### 

#### Error Handler

| **ErrorHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the ErrorHandler class. It takes the status code for the error that has occurred, and the logging object. First it performs the super class (Handler class) constructor, to store the logging object, followed by storing the status code as a private variable. The status needs to be stored so that the error that is handling can be accessed later to generate the appropriate response. | | |
| **Aggregation**  This function uses aggregation. It takes a logging object so that the outputs can be logged to track the requests and the process of handling the exception. This helps to make debugging simpler, and can also be disabled through the configuration file. Aggregation is used over composition, as it reduces the number of objects used in each script, and as using a single logging object does not have a performance impact as the requests are single threaded, aggregation should be used. | | |

### 

| **Responding to an error (within ErrorHandler class)** | | |
| --- | --- | --- |
|  | | |
| This function produces the error response. It first logs the error code that is being handled. It then creates a heading for the response, the literal exception that has occurred. It inserts the status in heading tags, so it is very clear to the user. It then adds an error description after this header as a paragraph to explain the error. If the error begins with a 4, it is a page that does not exist, or cannot be accessed. In this case it adds a paragraph explaining that the page does not exist, within the HTML paragraph tags. However, if the status begins with a 5, it is a server error, so adds a paragraph explaining this. This creates a HTML page from the error message, which can then be displayed. It then sets the response\_headers to set the content type to HTML, before returning the response, the status for the handler, and the headers. These are required to be in this order so it can be called by the \_\_call\_\_ method, and use the error\_response function elsewhere, and be treated the same as other functions from different handlers. | | |

### 

| **Call method for ErrorHandler class** | | |
| --- | --- | --- |
|  | | |
| This overrides the \_\_call\_\_ method from the Handler superclass. This is so that some processes can be removed, for example finding the corresponding function from the routes dictionary. It logs the error handler name, and the request URL that is being handled. It then generates the response HTML, and carries out the start\_response function. Finally, it logs the response, status code, and the headers. Finally it returns the content, in binary, using the encode method. It uses yield instead of return so it can gives an iterable response, which is required for WSGI to work (Bard, 2016). | | |
| **Binary**  This function uses binary this is because it takes the HTML response data and converts it into binary, using the encode method. This allows it to be sent to the client, as WSGI requires the data to be in binary to work. | | |
| **String manipulation**  This function uses string manipulation. It uses Python’s f-strings to insert the status code, response, response headers and the request URI into strings so that they can be logged. It also uses contatenation to append the class name to specify which handler is being used to handle the request. This allows the code to be significantly easy to trace and debug, if the log is enabled through the configuration. | | |

#### 

#### Handler classes

Each of the handlers process a set of related requests. They each follow the same basic format, as they all inherit from the same Handler superclass, and each method within them needs to have the same return values and format, parameters, and exception handling within them.

The constructors follow the following pro-forma. It calls the superclass constructor first, and then overrides the routes dictionary. This redirects each request to the appropriate function to process it, specifying which URL matches to each function call. Consequently when the object is called by the Middleware class, it performs the corresponding function within the class, because of the inherited \_\_call\_\_ method. When the function is called, it extracts the sub-target from the URL. This is the function that is being requested to be run, so looks up the request by this single part of the URL. As a result, the keys within the routes dictionary are just the function name.

Each function to handle a request also follows a template. First it gets the parameters from the request, either GET or POST. If it is a POST request that sends the data in JSON, it converts it from a string into a dictionary. Then it tries to execute the functions to get the relevant data from the database, and update the session ID if it has not expired. It then catches any relevant errors that have been thrown specifically (custom exceptions for the project), and handles them as appropriate. This exception is normally a session expired error, in which case it gives a 403 error. It also sets the response headers depending on the data that is being sent to the client. Finally it returns the response, status code and the parameters in that order. This template goes as follows:

| def function\_name(self):  data = retrieve\_parameters()    try:  self.\_log.output\_message(" Session id: " + session\_id)  user\_id = sessions.get\_user\_id(session\_id)  sessions.update\_time(session\_id)  self.\_log.output\_message(" User id: " + str(user\_id))  response = object.function(parameters)  status = "200 OK"  response\_headers = [  ("Content-Type", "application/json"),  ("Content-Length", str(len(response)))  ]  except components.accounts.SessionExpiredError:  self.\_log.output\_message(" Session expired")  response = "false"  status = "403 forbidden"  response\_headers = [  ("Content-Type", "text/plain")  ]  return response, status, response\_headers |
| --- |

The SessionExpiredError will be thrown within the sessions.get\_user\_id function, if the given session ID is either expired or invalid. In these situations, the except clause runs instead, which gives a 403 forbidden status code, as the resource exists, but cannot be accessed as the used session ID is invalid. It then sets the response to being a string containing “false” as the request failed, and a response should be given for completeness. Consequently the Content-Type is set to plain text so it can be displayed correctly. If the session ID does correspond to a user ID, the get\_user\_id function returns the user ID. This is then logged, along with any other data from the request, and updates the session ID expiry time, so the time before it expires is reset. Then it executes the corresponding function(s) from the components module, and either returns the results from these, or ‘true’, if the request does not require a response. The status is set to 200 OK, as the request was made, and responded to correctly, and the headers are set to the Content-Type (either application/json for dictionary results, or text/plain for strings), and the Content-Length set to the length of the result, so that the data is sent to the client correctly, in the correct form to be used in the JavaScript. Finally, it returns the response, status and response\_headers in that order. It is essential that the order of these return values is the same for all of the handler functions, as they are taken within the \_\_call\_\_ method of the handler to send the result of the request to the client..

#### Accounts Handler

| **AccountsHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the AccountsHandler class. It first performs the superclass constructor to store the logger object, and log the creation of the handler. It then overrides the private routes variable to redirect the three possible requests that it handles to the corresponding function within the class. The corresponding function is stored as the key, without the brackets. Omitting the brackets means that it can be accessed and called from the dictionary, rather than calling the function when the dictionary is created. | | |

| **Sign in request handler (within AccountsHandler class)** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
|  | | |
| This function handles the sign in request. First it gets the user’s credentials from the request by getting the post data using the inherited retrieve\_post\_parameters function. This result is given as a string, but contains JSON data, which is then converted into a dictionary so that the data can be accessed and used. It then tries to get the corresponding user ID from these credentials. If the credentials are valid, the check\_credentials function gives the user ID. It then takes this and creates a new session. In this case, the message for the client should be “Signed in successfully”, as the credentials were correct, and the session ID was created. It also logs the user signing into the account, and the session ID.  If the credentials are incorrect, the check\_credentials function then throws the InvalidUserCredentialsError. This stops the processing in the try, and directly executes the except part of the statement. It then logs the incorrect sign in attempt, and sets the session ID to None, as a session ID can not be created without a user ID. Finally it logs that the session ID could not be given.  Following this try-except statement, it creates a dictionary, containing the message string, and the session ID, converting this into a JSON string. The status is also set to 200, as the request was successful. Finally, it sets the Content-Type header to JSON so it is processed correctly at the client, and the Content-Length header to ensure that all of the JSON is recieved by the client. Together they ensure that the response is received correctly. It then can return the response, status, and headers, so the response can be handled, and given correctly. | | |
| **JSON**  This method uses JSON, as it takes a JSON object from the client containing the login credentials, and convert it into a dictionary. It also responds to the client in JSON, converting a dictionary into a string. | | |
| **Exception Handling**  This method uses exception handling to detect and process when the given credentials are incorrect. It allows the invalid user credentials to be detected clearly, simply, and in a readable way, and the appropriate response to be given. | | |
| **Users must be able to log in to the website to access their personalised information.**  This code helps meet this objective, as it responds to the sign up request that is issued by the javascript. It checks that the credentials are correct, and creates a session if they are, or alerts the client that they were invalid. Once a session ID has been given, the user has signed in, and therefore, this method, in conjunction with the check\_credentials and create\_session functions, and the JavaScript, meets this objective. | | |

| **Sign out request handler (within AccoutHandlerClass)** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
|  | | |
| This method handles the sign out request. It first gets the session ID parameter, getting the session ID that is given in the POST request. It then closes this session, using the close session method. This does not throw any exceptions explicitly, so it does not need to catch any exceptions. It then sets the status to 200 OK, as the request was handled correctly, as well as setting a response to “true”. This response is given for completeness, as it is not required, or processed by the client. Finally, it sets the headers, setting the Content-Type to plain text, and the Content-Length to the length of the string. This allows the data to be received correctly, although this is not essential for the client. The response, status and response\_headers are returned. | | |
| **Users must be able to log in to the website to access their personalised information.**  This method helps to meet this objective. It is implied through this objective that users should also be able to sign out of the website, which is facilitated by this. It responds to the sign out request from the client, and closes the session that the client was using. This meets this objective, in conjunction with the JavaScript, and the close session procedure. | | |

| **Sign up request handler (within AccoutHandlerClass)** | | |
| --- | --- | --- |
| Users must be able to create their own accounts on the website. | Met |  |
|  | | |
| This function handles a sign up request. First, it gets the JSON data from the environ variable, using the inherited retrieve\_post\_parameters method. It then tries to create the account. This first performs the create\_user function to create a new record in the database, which, if successful, returns the user ID. Following this, it creates a new session, and assigns the message to ‘Account created successfully’, and logs the creation of the new account, and the session ID.  If the user cannot be created, the accounts.create\_user function throws the UserExistsError. This stops any subsequent processing of the try, and immediately starts running the except clause. It the logs the fail to create the account, and sets the message to ‘Username is already taken’. This message is set to this, as this exception only occurs if the desired username is already used within the database. Consequently, the session ID cannot be created, so the session\_id is set to None. It also then logs the session ID.  It then sets the response, to a JSON string, containing the message and the session ID. It also sets the status to 200 OK, regardless of whether it succeeds or fails, as the request was valid, and correct, regardless of whether the account can be created or not. It also sets the headers, with the Content-Type to JSON, and the Content-Length is set to the length of the JSON string. This ensures that the result is recieved correctly at the client, and can be used like a JSON object within the JavaScript. The response, status, and response\_headers are then returned so that the request can be responded to. | | |
| **Exception Handling**  This uses exception handling to detect whether the account was created, or whether the username is already in use. This allows the user to be created if possible, and if not, it then changes the message, and session ID as a clear way to detect whether the account was created at the client. | | |
| **Users must be able to create their own accounts on the website.**  This helps to meet this objective, as it responds to the create account request issued by the JavaScript. It creates a new account in the database, handling any issues with the request, and informing the client of them. Therefore, it meets this objective, in conjunction with the JavaScript and the sessions module. | | |

#### Reading lists Handler (My Books)

| **MyBooksHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the MyBooksHandler class. It first executes the superclass (Handler) constructor to store the logger using aggregation, and log the creation of the object with this logging object. Then it overrides the routes dictionary so it can link each of the individual requests that it handles to the corresponding function. Again, the functions are given as values, but are not given with brackets, to prevent them from executing, and rather allowing the dictionary to be accessed as self.\_routes[request\_name]() to then execute the function. This allows the correct function to be called when it is needed by the inherited \_\_call\_\_ method. | | |
| **Inheritance**  This method uses inheritance, as it uses the inherited constructor to help create the MyBooksHandler. | | |
| **Overriding**  This method uses overriding. It changes the routes dictionary that was created by the superclass constructor to set the specific URL to function mappings for this handler. | | |

| **Get reading list names request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
|  | | |
| This method handles the get\_lists request. First it gets the session ID from the get parameters dictionary, which is found using the inherited retrieve\_get\_parameters method. The session ID can be taken directly from this, as the GET parameters only contain a single parameter, the session ID. It then gets the user ID from the session ID. If it has expired, this throws the SessionExpiredError, otherwise it returns the user ID. It then updates the time of the session ID, which then resets the expiry time of the session ID. Following this, it logs the user ID to help track the request throguh the script. It then gets the reading lists for the user ID, and takes the result of this, converting it into a dictionary of the names, with the index in the result as the key. This acts as a priority queue. It also converts this dictionary to a JSON string, and sets it as the response. It also sets the status to being 200 OK as the request is successful, as well as the headers, with the Content-Type as JSON, and the length as the length of the response.  If the session has expired, the SessionExpiredError is thrown within the get\_user\_id function. This stops the execution of the try part immediately, and processes the except. It then logs that the session expired, and sets the response to false. The status code is given as 403 forbidden, as the user was not using a valid session ID. It then also sets the headers, setting the Content-Type to plain text. The response, status, and headers can then be returned. | | |
| **Exception Handling**  This method uses exception handling to handle a session having expired. It tries to get the user ID from the session ID, and if it is expired, it gives a 403 error code, which gives a simple, easy way to detect this, and handle it. If the session ID is still valid, it then gives the list names instead, within a JSON object | | |

| **Get list entries request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
|  | | |
| This function handles the get\_list\_entries request. First it gets the GET parameters using the inherited retrieve\_get\_parameters method. It logs the target session ID, and then tries to get the user ID from the session ID. If it is valid, it returns the user ID, and if it isn’t, such as an invalid or expired session ID, it throws the SessionExpiredError, which is subsequently caught. When the request is valid, it updates the session ID time, so it resets the expiry period for the session. It then also logs the user ID, and the list ID. It then gets the reading lists values from the database, with the get\_values function, creating a new dictionary, containing the books dictionary, button, and the target list ID if the move button should be shown. It also adds a meta item to the dictionary, which contains a string saying that there are no books in the list, so the user can be alerted to it. This dictionary is converted to a JSON string, and stored as the response. It also sets the status to 200 OK as the data was retrieved successfully, and the headers set, with the Content-Type to JSON and the length to that of the response JSON.  If the sessionID is expired, or invalid, the get\_user\_id function throws the SessionExpiredError. In this case, it gives a status of 403 forbidden, as the session ID used is invalid, and cannot be used. It also sets the response to false, and the headers to plain text.  The response, status and headers can then be returned so that they can be used when the handler is called, to respond to the request correctly. Their does not need to be a check for whether the target reading list exists. This is because it is only accessable through the client by explicitly clicking on links to each reading list, which is fetched from the database with the corresponding ID. This ensures that the reading list must exist, as the list ID is explicitly known to the client. | | |
| **Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list**  This code helps to meet this objective, as it gets all of the data stored in the database for the reading lists. This is then converted into a JSON, which is then sent back to the client, which can then be processed by the JavaScript. Between this, the components module, and the JavaScript, this objective is met. | | |
| **Reading lists, and the entries within them should be specific to the user.**  This code meets this objective, as it gets the user ID from the session ID given in the request from the JavaScript, and uses this to specify which user’s results to get. This ensures that the results given are for the single user that is signed in. | | |
| **Each entry into the Currently Reading reading list should supply a method to mark an entry as read and move it to the have read reading list.**  **Each entry into the Want to Read reading list should supply a way to the user to start reading the book and move the entry to the currently reading list.**  This code helps meet these objectives. For both of these, it adds the text that should be shown in the button that is used to move to a different reading list. It also gives the target list ID of the destination lsit, so it can be sent whenever the button is pressed, so it can be moved directly, without any additional queries, or processing. This simplifies moving items between lists. | | |

| **Delete entry request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Each entry into a reading list should allow the user to delete the entry. | Met |  |
|  | | |
| This function handles the request to remove a list entry. First, it gets the post parameters, and converts it from the JSON string, into a dictionary. It then logs the session ID, before trying to get the user ID from the session ID. If the session ID is valid, it returns the user ID, otherwise it throws a SessionExpiredError. It then updates the session ID expiry time, and logs the user ID, list ID and book ID of the request. It removes the list entry, using the function from the components.reading\_lists object. It sets the status to 200 OK, as the request was successful, and the response to true, as the request was successful, however the data is not required for the client, as it does not process the result. It then sets the headers to plain text, with the content length specified too, so the response is handled correctly.  If the SessionExpiredError is thrown when trying to get the user ID, it is either invalid or expired. In this case, it then handles it the same as in the previous function, setting the headers, and the status to 403 Forbidden.  Finally it returns the response, status and headers, so the response can be handled correctly, within the \_\_call\_\_ and the Middleware. As the book ID and list ID is sent directly to the client, and the buttons that perform this request use this data directly, it is not possible for these vallues to not exist, and therefore, it does not need to be handled. | | |
| **Each entry into a reading list should allow the user to delete the entry.**  This function helps meet this objective, as it responds to the request made by the JavaScript, when the delete entry button is pressed. It uses the delete list entry procedure from the components module to remove the entry. This therefore, in combination with the JavaScript, deletes a reading list entry from the database. | | |

| **Move an entry request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
|  | | |
| This function handles the request to move a list entry from one list to another. It first gets the post data using the inherited retrieve\_post\_parameters method. Like the other methods, it gets the user ID from the session ID, and if it throws a SessionExpiredError, it gives a 403 Forbidden status code, otherwise it updates the session ID expiry time. It then logs the user ID, list Id, book ID, and target list ID for the move, then moves the entry using the components.reading\_lists object’s move\_entry method. It then sets the headers, with the content type as plain text, before returning the response, status and headers.  It does not need to handle the list ID, book ID, and target list ID not being valid. This is because they are explicitly sent to the client, and can only be accessed through the buttons on the webpage, directly using the sent values. This ensures that all of the IDs are valid, so must exist. | | |
| **Each entry into the Currently Reading reading list should supply a method to mark an entry as read and move it to the have read reading list.**  **Each entry into the Want to Read reading list should supply a way to the user to start reading the book and move the entry to the currently reading list.**  This function helps meet these objectives, as it handles the move a reading list entry request, sent by the JavaScript. It takes the starting list, and the end list from the POST data, as well as the book ID, and moves the entry from the first to the second list. In combination with the JavaScript, this meets these objectives. | | |

| **Delete a reading list request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Users should be able to delete their own custom lists | Met |  |
|  | | |
| This function handles the request to remove a reading list. This follows the template function, getting the post parameters using the inherited retrieve\_post\_parameters() function. If the session ID is valid, it then removes the target list, using the remove\_list function from the reading\_lists object from the reading lists part of the components module. It does not need to check whether the user should be able to delete the list, as the JavaScript hides the button to delete the list for system-defined lists (‘Have Read’, ‘Want to Read’ and ‘Currently Reading’), so they cannot be deleted. The response is also set to ‘true’, and the Content-Type header to tet/plain, as the client needs a response, but the content of it does not matter, and ‘true’ is a logical response to use, as the request was successful. If the session ID is invalid, it uses exception handling, in the same way as in the template. Finally, it can return the response, status, and response\_headers, so that the response can then be made from the \_\_calll\_\_ part of the handler class. | | |
| **Users should be able to delete their own custom lists**  This function helps to meet this objective, as it takes the request to delete a reading list from the client, issued by the JavaScript, and then performs the database operations to delete the reading list from the database. Therefore, in combination with the JavaScript, and the components module, this code meets this objective, as it allows reading lists to be deleted. | | |

| **Create list request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Users should be able to create their own custom reading lists | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
|  | | |
| This function handles the request to create a new list, following the template function given previously. It first gets the POST request parameters using the inherited retireve\_post\_parameters function, and converting the resulting JSON string into a dictionary using Python’s JSON library. As in the template, it logs the session ID, and then converts it to the user ID. If unsuccessful, it raises the SessionExpiredError, which is then caught by the except statement. It this situation, it sets the response to ‘false’ as the request failed, and the status to 403 forbidden, as the user cannot access the data with the session ID, which has either expired or is invalid.  If the user ID valid, the get\_user\_id method returns the user ID, which is then logged, along with the user ID and the new list name. It then updates the expiry time of the used session ID, then uses the reading\_lists object from the components module to create the new list. In this case, it sets the response to ‘true’ as the request was successful, and the status code to 200 OK, as the request was valid, and performed successfully. The headers are then set, with the Content-Type being plain text, and the Content-Length being the length of the response string. These headers ensure that the response is received correctly at the client, although the response doesn’t matter, only success or failure is important.  Finally, it returns the response data, status, and headers, in this specific order so that it can be used within the Handler \_\_call\_\_ method. This return statement works for both for when the session ID is valid and when it isn’t. | | |
| **Users should be able to create their own custom reading lists**  This function helps to meet this objective, as it recieves the request to create a new, custom list, and uses the components module to then create the new list for the user. Consequently, in combination with the JavaScript issuing the request, and the reading lists part of the components module creating the new list, it meets this objective. | | |
| **Reading lists, and the entries within them should be specific to the user.**  This function helps to meet this objective, as it takes the session ID, whilst making sure it is valid, and then using the reading lists part of the components module to create the new list, assigning the user ID to the new list, to ensure that it is user specific. Consequently, in combination with the JavaScript sending the session ID, and the components module, it ensures that the reading lists are specific to the user. | | |

| **Get list names (specifiying if a specific book is in each list) request handler (within MyBooksHandler)** | | |
| --- | --- | --- |
|  | | |
| This function responds to the get request to get a user’s reading list names, and whether a specific book is within each of them. It follows the template explained previously. First it gets the GET parameters, logging the session ID. It then tries to get the user ID from this session ID, which throws the SessionExpiredError exception. This exception is caught using exception handling, and, as explained in the template, sets the status to 403 forbidden, as it cannot be accessed from the given session ID, and the response to ‘false’. It also sets the content type header to 403, as it cannot be accessed.  If the session ID is valid, however, the get\_user\_id gives returns the user ID, corresponding to the session ID. It then logs the user ID, and book ID, to help make the requests easier to track and debug. Following this, it then executes the get\_names\_check\_book\_in method from the reading\_lists object, from the reading lists part of the components module. This returns a dictionary containing all of the lists, with their name, ID, and whether the book given in the GET request is in each list. This is then assigned to the result, which is the converted into a JSON string, using the JSON dumps method from Python’s JSON module. The status is set to 200 OK, as the request was successful, and the headers are set, with the Content-Type to application/json, as the response is in JSON, and the Content-Length to the length of the JSON string. This ensures that the JSON is received correctly.  Following the try-except, it returns the response, status and headers. This works for both the success of the request, or if the session ID is invalid. | | |

| **Add a list entry handler (within MyBooksHandler)** | | |
| --- | --- | --- |
| Users should be able to record the books that they have read in the past. | Met |  |
| Users should be able to record the books that they are currently reading. | Met |  |
| Users should be able to record the books that they would like to read in the future. | Met |  |
| Users should be able to create their own custom reading lists | Met |  |
|  | | |
| This function handles the request to add a new entry to a reading list. It follows the response function template. First it gets the POST request parameters from the request, converting the JSON string into a dictionary using the loads function from Python’s JSON library. It then logs the session ID, before trying to get the user ID from it, with the get\_user\_id method. If the session ID is invalid, either if it has expired or is unknown, the SessionExpiredError is thrown. This error is then caught through exception handling, and as in the template, it gives the response of ‘false’, and the status code 403 forbidden, as well as setting the Content-Type header to be plain text.  If the session ID is valid, however, it then the get\_user\_id method returns the user ID. It then updates the session ID expiry time, and logs the user ID, book ID and the list ID. Following that, it uses the add\_entry method of the reading lists object, from the reading lists part of the components module to add the entry into the reading list. It then sets the response to ‘true’ as the request was successful, and also the status to 200 OK, as the request was made, and a response was given.  Finally, it returns the response, status and headers, regardless of whether the session ID was valid or not, so that the request can be responded to. | | |
| **Users should be able to record the books that they have read in the past.**  **Users should be able to record the books that they are currently reading.**  **Users should be able to record the books that they would like to read in the future.**  **Users should be able to create their own custom reading lists**  This code helps to meet these objectives, as it allows an entry to be made into any of a user’s reading lists. This allows the users to be able record the books they have read in the past, the ones they are currently reading, and the ones they would like to read within the reading lists that are automatically created when the user signs up. By allowing entries to be made into any of the user’s reading lists, it also allows them to add entries into their own, custom defined reading lists, so therefore it allows them to create their own custom lists. In combination with the JavaScript issuing the request, and the components module object adding the entry to the reading lists, this meets these objectives. | | |

#### Genre Handler

| **GenreHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the GenreHandler class. As with the other handlers, it executes the superclass constructor, and overrides the routes variable so the specific URLs correspond to the correct functions.  The genre handler only needs to process a single request. This means that it could have been completely rewritten so the \_\_call\_\_ method checked whether the URL was correct, then executed the function. This would be marginally faster than this approach, however does not allow for future requests to be added to the Handler class without a full rewrite of it. This would not be very maintainable, and makes the project difficult to expand. The very marginal performance benefit that rewriting it would bring was not worth the additional complexity, and lack of future expandability that it would bring. | | |
| **Inheritance**  This uses inheritance, as it uses the Handler superclass’s constructor to log the creation of the object, and store the logging object. | | |
| **Overriding**  This code uses overriding, as it overrides the routes dictionary so it maps the URLs for this handler to the correct function, as opposed to the empty dictionary that is created in the superclass’s constructor. | | |

| **About page request handler (within GenreHandler)** | | |
| --- | --- | --- |
| It should display the genre’s name | Met |  |
| It should show an about summary for the genre, if applicable | Met |  |
| It should show the books that are part of the genre | Met |  |
|  | | |
| This function responds to requests for a genre’s about data. It first gets the genre name from the GET request parameters. It does this by using the inherited retrieve\_get\_parameters function, and taking the genre name from the resulting dictionary. As the request only has a single get parameter, and does not need any others, it can be done on a single line, and just store the genre name. It then logs this name, to help track the request. Following this, it then uses get\_about\_data method on the genres object, from the class in the genres part of the components module, to get all of the relevant data for that genre from the database, as a dictionary. This includes the books it matches, the genre name, and the summary information for the genre. It then converts this dictionary to a JSON string, using Python’s JSON library, then logs this response and the status, which is set to 200 OK, as the genre exists, and the data is returned successfully. Finally, it sets the headers, with the Content-Type as JSON, and the Content-Length to the length of the JSON string, to ensure that it is displayed correctly, before returning the response, status and headers.  If the genre does not exist, the get\_about\_data method of the genres object throws the GenreNotFoundError. This is then caught using exception handling. If it does not exist, the data cannot be generated, and therefore gives a 404 Not Found error, which is logged, to help debugging. In this case, it creates an ErrorHandler instance, handling a 404 Error, and calling the error\_response function within it. This creates a HTML page to indicate that the page has not been found, and returns the response, status, and relevant headers, in the correct order. This also then allows the result of this function to be directly returned. | | |
| **Exception Handling**  This uses exception handling to handle whenever a requested genre does not exist. In this case, it gives a page explaining the page was not found, with a 404 error. This allows the appropriate response to be given, and for the script to continue running if it does not exist, as well as being in a readable manner. | | |
| **It should display the genre’s name**  **It should show an about summary for the genre, if applicable**  **It should show the books that are part of the genre**  This code helps to meet all of these objectives. When the request is made, it executes the get\_about\_data method on the genres object from the genres part of the components module. This returns a dictionary including the genre’s name, summary, and the books that are within that genre, and consequently gives these pieces of data for the about page. Therefore, in combination with the JavaScript making and processing the request, and the components module, it meets these objectives. | | |

#### Books Handler

| **BookHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the BookHandler class. As with the other handlers, it first executes the superclass constructor, and then overrides the routes dictionary with the specific URLs and their corresponding functions. | | |
| **Inheritance**  This function uses inheritance, as it calls the Handler superclass’s constructor, which logs the creation of the handler, and storing the log object. It also creates an empty routes dictionary, which needs to be overridden. | | |
| **Overriding**  This function uses overriding, as it changes the empty routes dictionary that is created from the the inherited constructor, and changes it to having the handler-specific URLs mapping to the different functions within the class. | | |

### 

| **Get about request handler (within BookHandler)** | | |
| --- | --- | --- |
| Reviews should be listed on the corresponding book about page. | Met |  |
| Reviews on the book should show the user’s username. | Met |  |
| The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book | Met |  |
| A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website. | Partially Met |  |
| There should be a list of books that are related to each other displayed. | Met |  |
|  | | |
| This function handles the request for a book’s about data. It does not follow the template for the request functions. First it retrieves the GET parameters using the inherited retrieve\_get\_parameters function, then logging the session ID, and the book ID, which are the only two values given within the request. It then tries to get the user ID from the session ID. If it is a valid session, the get\_user\_id method returns the user ID, and if it is not valid, either because it is not a valid session ID, or it has expired, it throws the SessionExpiredError. If the session ID is valid, it then stores the returned user ID, and updates the session ID’s expiry time. If the SessionExpiredError is thrown, it is an invalid session ID, it is caught through the try-except statement. In this case it sets the user ID to None, as there is not a user signed into the website, as well as logging that the session has expired, or does not exist.  It logs the user ID, to help with debugging before proceeding to get the books about data. It then tries to to get the books about data by using the get\_about\_data method on the books object from the books part of the components module. If the book does not exist, it throws the BookNotFoundError, which is then caught. When this occurs, it sets the status to 404 Not found, as the desired page does not exist, then logging it. Following this, it creates a new ErrorHandler instance, also giving 404 as the status code for it, then calling the error\_response() function within it. This handler only needs to be used once for this function, so can just be instantiated here. The function also gives the relevant response, status code, and headers as a return value. This allows for the result of the function to be returned directly.  If the BookNotFoundError is not thrown, the book exists. It then gets all of the data about the book by using the get\_about\_data on the books object, which includes the title, synopsis, reviews, cover image, author name, author about, number of followers, the number of followers the author has, and the link to buy the book. It also includes the current user’s review, if the user ID is known. This data is within a dictionary, so additional items can then be added to it later. It then gets the similar books to the book that the request is about, using the books object’s get\_similar\_items function on the book ID, adding it into the dictionary. The status is then set to 200 OK, as the data can be requested about the book. The dictionary is then converted to a JSON string, again using the Python JSON library. Finally it sets the headers, with the Content-Type as JSON, and the Content-Length to the length of the JSON string, to ensure that is received by the client correctly, and can be processed by the JavaScript as a JSON object too. The response, status and headers are then returned. | | |
| **Error Handling**  This function uses error handling, to get the user ID from a given session ID. If it is not given, it then sets the user ID as None, so it can be used within the subsequent functions. It also uses error handling for getting the book data. It handles the BookNotFoundError that is thrown if the book does not exist, and gives a 404 Error page instead. | | |
| **Reviews should be listed on the corresponding book about page.**  **Reviews on the book should show the user’s username.**  **The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book**  This code helps to meet these objectives, as when handling the get data request, it uses the books object’s get\_about\_data method, from the books part of the components module. This includes the reviews, including the username, and all of the relevant data, as well as the book’s title, author’s name, genres, synopsis, cover image, average rating, and the purchase link. Consequently, in combination with the JavaScript issuing the request and using the response to construct the about page, it meets these objectives. | | |
| **A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website.**  This code partially meets this objective. It gets a link through the get\_about\_data method, but does not web scrape the price, so this is not given. This wasn’t included due to time constraints. | | |
| **There should be a list of books that are related to each other displayed.**  This code helps to meet this objective, as it uses the get\_similar\_books method on the book that is being requested. The similar books are then added to the dictionary, so will be sent to the client. Consequently, in combination with the JavaScript inserting the similar books into the webpage, this meets this objective. | | |

### 

| **Delete review request handler (within BookHandler)** | | |
| --- | --- | --- |
| Users should be able to delete any reviews that they have left | Met |  |
|  | | |
| This function handles the request to delete a review. It follows the template function. It first gets the POST request data using the inherited retrieve\_post\_parameters method, and converting the JSON string that it returns into a dictionary so that the values can be accessed. It logs all of these parameters as well, to help make debugging it easier. As with the template, in then tries to get the corresponding user ID from the session ID. If the session ID has expired, or is invalid, it then throws the SessionExpiredError, which is then caught, and the response set to ‘false’ as it failed, the status to 403 forbidden, as it cannot be performed with the given session ID, and the Content-Type header to plain text, so the response is displayed correctly.  If the user ID is retrieved for the session ID, it then resets its expiry time to the current time, so it doesnt expire. It then uses the books object’s delete\_review method to remove the specified review, giving both the user ID, and the review ID, so that the review ID cannot just be guessed to delete it, the corresponding session ID must be known to do so. The response is then set to ‘true’ as the review was deleted successfully. It isn’t used or checked by the JavaScript, as a successful error code, like the 200 OK one that the status is set to, indicates that the review was deleted successfully. The headers are then set, with the Content-Type to being plain text, and the Content-Length being set to the length of the response string, which together ensure that the response is received correctly by the client.  Finally, the response, status and headers are returned so that the \_\_call\_\_ method on the handler can send the response to the client correctly. | | |
| **Users should be able to delete any reviews that they have left**  This code helps to meet this objective, as it responds to the request to delete a review issued by the JavaScript, and then deletes it from the database. It also needs a correct session ID, which corresponds the review that the user is trying to delete, to be able to remove the review, which ensures that only the user can delete their own reviews as the review ID would need to be guessed, and a correct session ID which is present in the database, and corresponding to the user, to be guessed as well to be able delete another user’s reviews. Consequently, in combination with the JavaScript, and the components module, which handles the database query to delete the review, this objective is met. | | |

### 

| **Leave a review request handler (within BookHandler)** | | |
| --- | --- | --- |
| Users should be able to leave reviews about each book | Met |  |
| Reviews must have an overall rating. | Met |  |
| Reviews can have a character rating, plot rating, review summary, and review body. | Met |  |
| Users should only be able to review a book if they have read it, marked by it being in the users’ *have read* list. | Met |  |
|  | | |
| This function handles the request to leave a review. It follows the function template. First it gets the POST request data, by calling the inherited retrieve\_post\_parameters function, and converting this into a dictionary, using the loads method from Python’s JSON library. This library is used as it makes converting between JSON and dictionaries simpler, and removes the need to consider None to null, and any other data type conversions. It then logs the session ID, and tries to get the user ID from it. If it cannot, the SessionExpiredError is thrown, which is then caught. It logs that the session is expired, sets the response to ‘false’ as it failed, and the status to 403 forbidden, as it could not be performed, and access was denied. It also sets the Content-Type header to plain text to match the response.  If the session ID does have a user ID, and has not expired, the get\_user\_id method returns the user ID. The session expiry time is updated, so the user can continue to use it until the cookie expires (if it was set), or the page is closed. It then logs the user ID, and the book ID to help with debugging the request. As it uses the logging class, like any other of the debug logs, it can be enabled through the configuration (as it defaults to disabled). It then uses the leave\_review behaviour of the books object, from the books part of the components module, to leave a review, passing the user ID, book ID, and the overall rating, character rating, plot rating, summary, and the user’s thoughts to the function, regardless of whether they are None or not. This then adds a new review for the book.  It also adds the book to the user’s Have Read reading list. This ensures that the user has read the book that they are reviewing, by adding it to the reading list. Automatically adding it is better than prompting the user to add it, as it makes the user experience appear smoother. To do this, it first gets the ID of the user’s ‘Have Read’ reading list, using the get\_list\_id method of the reading lists object (from the class in the components module). It then uses the add\_entry method on the same object to add an entry into the user’s reading list entry. This add\_entry method checks whether the book specified is already in the reading list, so this does not need to check beforehand. Once this has run, the review has been created successfully, and therefore the response is given as ‘true’. Again, the JavaScript does not check the response, as the success status code that it gives implies that the review was created successfully. The status is set to 200 OK as the review was successfully created, so the request was handled successfully. Finally, it sets the headers, with the Content-Type as plain text, as the respons is just a string, and the Content-Length to being the lenght of the response string.  It then returns the response, status, and headers. This is done regardless of whether the session ID was expired or not. This ensures that as long as the function runs it will give an output. If it does not run, there was an unexpected error, which should not have occured, and therefore should not be caught here. This is caught by the catchall statement in the inherited \_\_call\_\_ method, and a 500 Server Error response is given instead, without stopping the script from running. | | |
| **Reviews can have a character rating, plot rating, review summary, and review body.**  This code helps to meet this objective, as it takes these parameters from the POST body and passes them to the leave\_review method. This then adds them to the database. Consequently, in combination with the HTML, providing these fields for the user to enter, the JavaScript sending these fields, and the components module storing them in the database, and handling them not being specified, this objective is met. | | |
| **Users should only be able to review a book if they have read it, marked by it being in the users’ have read list.**  This code meets this objective, as when a user leaves a review, it automatically adds it to the user’s reading list. This then ensures that the book is marked as read if the user leaves a review. It does not add the book again if it is already in the list. Consequently, it ensures that a reviewed book is always in the user’s ‘Have Read’ list if they have left a review. | | |

#### Authors Handler

| **AuthorHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the AuthorHandler class. It uses the superclass's constructor, and then overrides the private routes dictionary with the URLs that each function should map to. | | |
| **Inheritance**  This code uses inheritance, as it uses the Handler superclass’s constructor. This logs the creation of the handler, and stores the logging object. | | |
| **Overriding**  This code uses overriding. This is because when the superclass’s constructor is called, it creates an empty routes dictionary. It is overridden so that it contains the URLs that the AuthorHandler class responds to, with the corresponding functions. | | |

### 

| **Follow an author request handler (within AuthorHandler)** | | |
| --- | --- | --- |
| Users should be able to follow each author. | Met |  |
|  | | |
| This function handles the request to follow an author, and it follows the template for the request response. It first gets the data from the POST request using the inherited retrieve\_post\_parameters function, and converts this from a JSON string into a dictionary so that the values in it can be accessed. It then logs the author ID and the session ID before getting the user ID from the session ID. If the session ID has expired, the get\_user\_id method throws the SessionExpiredError. This is then caught, and a 403 status code given, with a response of ‘false’ as the request failed, as the session ID was invalid.  If the session ID is valid, the get\_user\_id method returns the user ID. This is then logged to help debugging, and the session ID’s expiry time reset to the current time. Resetting the expiry time enables the session ID to be used until the cookie expires (if it was set), and be used until the website is opened, and as long as the website does not have one week of inactivity before it is closed, the session ID can still be used. It then uses the follow method on the authors object (an instance of the Authors class from the authors part of the components module), to mark that the user follows the author. It also sets t the number of followers the author now has after the user follows them as the response. This is done by using the get\_number\_followers method on the authors object, specifying the author ID. The result from this is cast to a string, so that it can be sent as text, so that it can be directly inserted into the webpage by the JavaScript. It then sets the status to 200 OK, as the request was made successfully. It also sets the headers, with the Content-Type being set to plain text, so the new number of followers can be directly inserted into the webpage, and the Content-Length to the length of the response, to ensure that the whole response is sent to the client.  Following the try-except, it returns the response, status, and headers. This returns either the 403 response if the session ID expired, or the 200 response if the session ID was valid. They are all returned, and in this order, so that the inherited \_\_call\_\_ method can then send the data to the client. | | |
| **Users should be able to follow each author.**  This code helps to meet this objective, as it responds to the request to follow an author. It handles the request, and adds the new relationship to the database, using the authors class from the components module. Consequently, it meets this objective, in conjunction with the JavaScript issuing the request, and the components module handling the database operations to record the user following the author. | | |

### 

| **Unfollow an author request handler (within AuthorHandler)** | | |
| --- | --- | --- |
| Users should be able to follow each author. | Met |  |
|  | | |
| This function handles the request to unfollow an author. It works in exactly the same way as the follow\_author function, except it uses the unfollow method of the authors object, rather than the follow method. Consequently it removes the follow relationship between the specified user and author ID | | |
| **Users should be able to follow each author.**  While not explicitly given by this objective, it is also implied that if the user can follow an author, they should also be able to unfollow them. Consequently, this code helps to meet this objective by responding to the request to unfollow an author, and in combination with the JavaScript issuing this request, it meets this objective. | | |

### 

| **Get about request handler (within AuthorHandler)** | | |
| --- | --- | --- |
| It should display the author’s average book rating. | Met |  |
| It should display all the genres that the author writes for. | Met |  |
| It should show all the books that the author has written. | Met |  |
| It should supply an about the author section. | Met |  |
| It should show the author’s name (or the name that they write under if applicable). | Met |  |
| The page should display how many followers each author has. | Met |  |
|  | | |
| This function handles the request to get an author’s about data. It first gets the GET parameters, using the inherited retrieve\_get\_parameters method. It then logs the author ID, before trying to get the author’s about data using the get\_about\_data function from the authors object (from the class in the components module). If the author does not exist, the get\_about\_data function throws the AuthorNotFoundError. This is then caught using exception handling. It then sets the status to 404 Not Found, as the author does not exist, so the resource that is being requested does not exist either, and logs this status to help with debugging. It also creates an ErrorHandler instance, to handle the ‘404 Not Found’ error. This is passed into it, so it handles the correct error. It then executes the error\_response method, which returns the response, status, and relevant headers. The order of the return values from this is the same as the one required for this function, so can then be returned directly, and as the ErrorHandler instance is not needed again, it can be returned directly.  If the get\_about\_data function does not throw the AuthorNotFoundError, the author exists, it will return the about data for the author, including the books that they have written, their name, their about, and their average rating. This dictionary is then converted to a JSON string so that it can be sent to the client, and processed by the JavaScript. It sets the status to 200 OK, as the request had a valid response, which is being sent correctly. The headers are set, with the Content-Type being set to JSON, so it is received as a JSON object, and can be used like one by the JavaScript. It also sets the Content-Length to the length of the JSON string, so that it is all sent to the client correctly. Finally it returns the response, status and headers, so that it can be used by the inherited \_\_call\_\_ method, and send the data to the client. | | |
| **Exception Handling**  This code uses exception handling to handle the AuthorNotFoundError that is thrown when the requested author does not exist. In this case it then returns the 404 Not Found page instead of the desired author about page. | | |
| **It should display the author’s average book rating.**  **It should display all the genres that the author writes for.**  **It should show all the books that the author has written.**  **It should supply an about the author section.**  **It should show the author’s name (or the name that they write under if applicable).**  **The page should display how many followers each author has.**  This code helps to meet these objectives, as the get\_about\_data method of the authors class includes all of this data. Consequently, between the JavaScript sending the data request to create the about page, and the get\_about\_data method of the authors class from the components module, these objectives are met. | | |

#### Diaries Handler

| **DiaryHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the DiaryHandler class. It uses the Handler superclass constructor to store the logging object, and log the creation of the object. It then overrides the routes dictionary to assign the URL to the corresponding function in the class. | | |

### 

| **Get entries request handler (within DiaryHandler )** | | |
| --- | --- | --- |
| The users’ diary entries must be available on a dedicated webpage on the website. | Met |  |
|  | | |
| This function handles the request to get a user’s diary entries. It follows the request template, by first getting the GET request parameters, using the inherited retrieve\_get\_parameters method. This returns a dictionary, but as it is only the session ID needed, it gets that value from it, and only stores that. As with the template, it then logs the session ID, and tries to get the user ID from it. If the session has expired or does not exist, it then throws the SessionExpiredError, which is then caught, and the response set to ‘false’, and the Content-Type to plain text to match this. The status is also set to 403 forbidden, as the request failed as the session ID could not be used, so access was denied.  If the session has not expired, however, it updates the session ID expiry time. It then creates a dictionary for the result. This dictionary contains the user’s diary entries which are found using the diaries object’s (from the class in the components module) get\_entries function, which gives a dictionary containing all of the user’s diary entries. It also gets a list of book summary dictionaries the user is currently reading, and also adds this to the result dictionary. This is given to create a list of all the books that the user can select when they add a new diary entry, as the book needs to be in the ‘Currently Reading’ reading list for a diary entry to be made. This dictionary is then converted to a JSON string, using Python’s JSON library’s dumps function. The status is then set to 200 OK, as the user’s diary entries were accessed correctly. It finally sets the headers, with the Content-Type header to being JSON so that the JavaScript can access the data in the dictionaries, and the Content-Length header to be the length of the JSON string, to ensure that the whole response is sent to the client.  Finally, it returns the response, status and headers which allows the inherited \_\_call\_\_ method to send the response to the client correctly. | | |
| **The users’ diary entries must be available on a dedicated webpage on the website.**  This code helps to meet this objective, as it gets all of the diary entries for a specific user, which can then be inserted into the dedicated diary entries webpage by JavaScript. Consequently, in combination with the JavaScript issuing the request and inserting the response into the webpage, this objective is met. | | |

### 

| **Delete an entry request handler (within DiaryHandler )** | | |
| --- | --- | --- |
| Users should be able to delete diary entries | Met |  |
|  | | |
| This function handles the request to delete a diary entry, which follows the template function given originally. It first gets the data from the POST request, using the inherited get\_post\_parameters method, and converting the JSON string it returns into a dictionary using the loads function of the JSON library. As with the template, it then gets the user ID from the session ID. If the session expired or is invalid, it then gives a response of ‘false’, and a 403 status code, as with the previous function. The headers are also set in the same way.  If the session ID is valid, the user ID is returned. It then updates the session ID’s expiry time, so that it can be used continuously, as well as logging the Entry ID and the session ID, and the user ID. It then uses the delete\_entry method of the diaries object to delete the entry. This removes the entry from the database. Following this, it sets the response to ‘true’ as it was successful, as well as the status to 200 OK as the response was given correctly. The Content-Type header is set to plain text, and the Content-Length is set to the length of the string. This ensures that the response is received correctly by the client. However, the client does not need this response, and only depends on success or failure, which is done through the status codes. It is given for completeness only.  Finally, it returns the response, status and the headers. This works for when the session ID is valid, or invalid, as it is outside of the try-catch statement. It can then use the single return to return the response and send it to the client. | | |
| **Users should be able to delete diary entries**  This code helps to meet this objective, as it handles the request to delete a diary entry. It deletes the entry from the database when the request is given, so consequently, in combination with the JavaScript issuing the request, it meets the objective. | | |

### 

| **Create a diary entry request handler (within DiaryHandler )** | | |
| --- | --- | --- |
| Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating. | Met |  |
| Users must specify the number of pages that they read. | Met |  |
| When the user finishes the book, the user should be given the option to make the diary entry available as a review. | Met |  |
|  | | |
| This function handles the request to create a diary entry, following the template. First it gets the POST parameters from the request, using the inherited retrieve\_post\_parameters request. It then takes the JSON string that this returns and converts it to a dictionary. It then tries to get the user ID from the session ID, using the get\_user\_id function, as done previously. If it fails, it throws the SessionExpiredError, which is then caught, as in the previous function, the response set to ‘false’, and the 403 Forbidden, as the request failed, as the session is invalid, so cannot be performed.  If the user ID can be accessed from the session ID, it then updates the session ID expiry time, so that it can be used continuously. It also logs the session ID, user ID and book ID to help to debug the request handling. This can be disabled through the configuration, so the logs are then not outputted. Following that, it uses the add\_entry method of the diaries object to then create a new diary entry. The parameters can be None from the request, and these are handled correctly by the add\_entry method, so this does not need to be checked before passing the values into the method. Executing this function then adds the entry to the database.  It also checks whether the book's completed value in the JSON is true. If it is true, it then adds the entry to the ‘Have Read’ reading list. This is done by getting the list ID of the user’s ‘Have Read’ list, and adding the entry to it. Within the add\_entry method, it removes the entry from the ‘Currently Reading’ list, so this acts to move the entry. It then also checks whether the as\_review value in the dictionary is true. This is done within the check for the book\_completed, as if the book is not completed, it should not be possible for the user to leave a review. This should be prevented through the JavaScript hiding the leave review button unless the book completed one is clicked, but this adds an additional layer to protect it from being performed. If the as\_review value is true, it then creates a review for the book, using the parameters given in the review, except the number of pages read. Finally, it sets the response to ‘true’ as it has been performed successfully. This does not affect the client, as the JavaScript bases the success or failure from the status codes, as if it gives a 200 OK response, the entry was created successfully. It is just given for completeness. It then sets the headers, with the Content-Type being plain text, as the response is a string, and the Content-Length header being set to the length of the response string.  It then returns the response, status and headers, regardless of whether the status code was 200 OK or 403 Forbidden. This allows the inherited \_\_call\_\_ method to then send the response to the client. | | |
| **Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating.**  This code helps to meet this objective, as it responds to the request to leave a new reading list entry. It takes the overall rating, character rating, plot rating, summary, thoughts and feelings, and the number of pages that the user read from the POST request, and then creates the diary entry. Consequently, in combination with the JavaScript sending the POST request, it meets this objective. | | |
| **Users must specify the number of pages that they read.**  This helps meet this objective, as it takes the number of pages that the user has read. Consequently, in combination with the JavaScript ensuring that it is given and including it in the request, this objective is met. | | |

#### Home Handler

| **HomeHandler class** | | |
| --- | --- | --- |
|  | | |
| This is the constructor for the HomeHandler class. Like the other handler classes, it runs the superclass’s constructor, and overrides the routes dictionary with the handler-specific URLs. Like the GenreHandler class, it only has one function within it, so consequently could be rewritten to only call that function, removing the need for the routes dictionary. This would require an entire rewrite of the class, which would reduce future expandability. The performance benefit of rewriting it would also be minimal, so it  This is the constructor for the AuthorHandler class. It uses the superclass's constructor, and then overrides the private routes dictionary with the URLs that each function should map to. | | |
| **Inheritance**  This code uses inheritance, as it uses the inherited constructor from the Handler superclass to log the creation of the handler, and store the logging object. | | |
| **Overriding**  This code uses overriding. This is because when the constructor from the superclass is run, it creates an empty routes dictionary. Consequently it needs to be overridden so that the | | |

### 

| **Get home data request handler (within HomeHandler)** | | |
| --- | --- | --- |
| The home page should give users a summary of their recommended books. | Met |  |
| The home page should give users a summary of what is in their want to read list. | Met |  |
| The home page should give users a summary of what is in their currently reading list. | Met |  |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. | Met |  |
| Book summaries should include the book’s cover image, author, and title. | Met |  |
| Each book summary should link the specific books about the page. | Met |  |
|  | | |
| This function handles the request to get the data from the home screen. It creates an empty dictionary for the result, beforing trying to get the session ID from the user ID. If it is not valid, either not known, or expired, the get\_user\_id method throws the SessionExpiredError, which is then caught. It then sets the recommended, currently\_reading, and want\_read parts of the dictionary to None. This is because there is not a user signed in these cannot be found, and setting them to None gives an easy way to detect this through the JavaScript. If the session ID is valid, however, it updates the expiry time of the session ID, so it can be used continuously. It then sets the recommended part of the results dictionary to the user’s recommendation summaries, limiting the results to the number of summaries on the home page constant. It also sets the currently\_reading and want\_read to the summaries from these lists, accessing them using the corresponding functions from the reading\_lists object. The number of summaries is also limited to the number of home summaries constant, before adding it to the dictionary.  Following this, regardless of whether the session ID is known or not, it sets the trending part of the dictionary to the trending books, accessed using the get\_popular method from the reading\_lists object, and the newest additions using the get\_newest method from the books object. Both of these already limit the number of results given, so it does not need to be limited here. They also give the summary dictionaries directly, so these can be used.  This dictionary is then converted into a JSON string using the dumps function from Python’s JSON module, so it can be sent to the client. It also sets the status to 200 OK, as the data was found successfully, regardless of whether the session ID is usable or not. The headers are also set, witht he Content-Type being JSON so it is recoeved by the JavaScript as a JSON object, and the Content-Length is set to the length of the string, so that it is all reieved by the client. Together this ensures that the data is recieved at the client correctly. Finally, it returns the response, status, and headers so that it can be used within the \_\_call\_\_ method, so the data can be sent to the client correctly. | | |
| **The home page should give users a summary of their recommended books.**  **The home page should give users a summary of what is in their want to read list.**  **The home page should give users a summary of what is in their currently reading list.**  **The home page should give users a list of trending books, which have the highest number of users currently reading them.**  **Book summaries should include the book’s cover image, author, and title.**  **Each book summary should link the specific books about the page.**  This objective helps to meet this objective, as it gets the recommended books, want to read books, and currently reading books if the session ID is valid. It also gets the trending books, and and the newest books in the database. Each of these are accessed using methods from the components module, which return the book summaries, containing the cover image, author and title. They also include the book ID, which then can be used by the JavaScript to link to the about page. Consequently, in combination with the JavaScript requesting and loading the data, and linking to the books about page, this meets the objective. | | |

#### Recommendations Handler

| **RecommendationsHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Get recommendations request handler (within RecommendationsHandler)** | | |
| --- | --- | --- |
| The users’ recommendations must be available on a dedicated webpage on the website. | Met |  |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
| Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre. | Met |  |
| Recommendation entries should have a method to move it to the want to read reading list. | Met |  |
| Recommendation entries should show the certainty of the recommendation | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Delete a recommendation request handler (within RecommendationsHandler)** | | |
| --- | --- | --- |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Move to recommendation to Want to Read request handler (within RecommendationsHandler)** | | |
| --- | --- | --- |
| Recommendation entries should have a method to move it to the *want to read* reading list. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Set initial preferences request handler (within RecommendationsHandler)** | | |
| --- | --- | --- |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Searching Handler

| **SearchingHandler class constructor** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Database search request handler (within SearchingHandler)** | | |
| --- | --- | --- |
| There must be a search bar that searches for books, authors and genres within the website. | Met |  |
| The search bar should be able to search by name or ISBN number | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

### 

| **Browse page data request handler (within SearchingHandler)** | | |
| --- | --- | --- |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website | Met |  |
| The browse page should display books related to the last one added to a reading list | Met |  |
| The browse page should display books related to the last read book | Met |  |
| The browse page should display books from the authors the user is following | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## Frontend

### Javascript

The JavaScript for the website handles all of the AJAX requests, in both making the request, and processing the response data. It also handles switching between pages on the website, without reloading the page. Finally, it handles formatting and cookies for remembering users. The majority of the actual website’s functionality comes from the JavaScript, which makes it a critical component of the project. It also forms a very large part of the project, and for that reason needs to be as simple as possible to do. To try and keep the complexity down, JQuery was used to simplify the process, especially for making AJAX requests, and for manipulating the DOM. Other JQuery-based modules were also used, namely one for cookies and debouncing function calls. These have been given in the requirements section of the technical solution. Due to the large size of this file, it has been explained in sections.

| **Global variables** | | |
| --- | --- | --- |
|  | | |
| This sets three global variables, which are then used throughout the code. The disablePopupCancel variable is used as a flag throughout the code, which specifies whether the sign in/sign up popups can be closed. There are situations that they cannot be closed, for example, when waiting for a response from the server, as to whether the sign in/sign up was successful, or if any errors occurred while processing it, like incorrect credentials, or existing username.  The session ID global variable is used to store the session ID for the session. It is used whenever the session ID is needed for the request. It defaults to null initially, as there is not a session ID in use when the script first runs. However, when the page loads, it checks whether there is a cookie being used, and if there is, it can then be changed then. Null is also easy to check for, as it allows for if (sessionID) {} to be used to check whether this is a specified session ID. It is also better than declaring the variable, without assigning a value, as it can easily be reverted to its initial state, which is important for signing out of the website.  The current page variable is declared here too. It is used to store the current page that the user is on. This allows for the pages to be loaded, without refreshing the page. | | |
| **Global variables**  While normally bad practice, these are needed through all of the functions in the code. Therefore, it uses global variables to allow them to be accessed, and overwritten from any part of the code. | | |

## 

| **Sign in from cookies and load the correct page data** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
| The website should be able to save a user’s login | Met |  |
|  | | |
| This code runs whenever the page is loaded. It gets the session ID cookie, using the js.cookie module. If there was a session ID cookie, it updates the session ID global variable, and also changes the account buttons, so that they are changed to the sign out, instead of the sign in. This does not occur if the cookie does not exist, which also keeps the sign in button. Following this, it also reloads the page, which performs the load function for the correct page, and also gets the relevant HTML file, and puts it within the page. | | |
| **Cookies**  This method uses cookies to store login details (session ID) on the website. This allows data to be stored on the client. It uses a module to simplify this process, as without it the complexity would increase, and would have made it harder to achieve this level of completion otherwise. | | |
| **Pages should be switched and navigated between without refreshing the page**  This meets this objective, as it gets the desired page using AJAX, through the reloadCurrentPage function, which does not require the page to refresh. | | |
| **The website should be able to save a user’s login**  This helps to meet this objective, as it loads a cookie when the page is loaded, to check for a session ID, which can then be used immediately. This allows the page to remember a user’s login, and automatically sign them in. | | |

## 

#### Utility Functions

| **Title case conversion** | | |
| --- | --- | --- |
|  | | |
| This adds a method to the String object to convert it to a title case, called like “”.ToTitleCase(). This was not coded, and was taken from Stackoverflow, and modified to function within this use case, removing additional features that were unneeded. | | |

| **Change the page URL** | | |
| --- | --- | --- |
|  | | |
| This function updates the URL to show the part of the website the user is on. It converts the link name to lowercase, replacing spaces with dashes, so that it can actually be displayed within the URL. For the home page, it changes it to an empty slash, which would then convert it back to being the root of the page, so it would show as domain-name, and if it is not home, as domain-name/linkName. Doing this helps make it clear to the user which page they are on, as well as adding the page to the user’s search history, so that they can navigate back using the return buttons on the browser. | | |

| **Get the page name from the URI** | | |
| --- | --- | --- |
|  | | |
| This function gets the link name from the URL. It reverts the changes that were made in the previously explained function to get the original link that was given to it. The slice within the URI is taken so that the first character (which will always be a slash) is removed. | | |
| **String manipulation**  This function uses string manipulation, as it converts the string to title case, as well as using slicing to remove the first character from the string. | | |

| **Add a parameter to a URL for a GET request** | | |
| --- | --- | --- |
|  | | |
| This function adds a get parameter to a given URL. This was taken from (Frisbie & Zach Tratar, 2020), as it handles all of the possible, valid cases simply. | | |

| **Fill in rating stars** | | |
| --- | --- | --- |
|  | | |
| This function fills the correct number of rating stars, given an average rating, as a decimal. It takes a JQuery selector as an input, so that the items can be accessed easily, and so that the function can be applied to any part of the website, without needing to consider the HTML classes, so it will work every time. Consequently, it is called, for example, changeElemStars($(".rating i"), 3.2). It does not work perfectly, as stars can only be full, half full or empty, and it does not round the values. This means that if the average rating is not a whole number, it will always fill as a half star, which is not ideal. To do it using rounding, or percentage fill of the stars, would have been significantly more complicated, with very little benefit as the exact average rating is displayed alongside them anyway.  First, it goes through all of the star elements, and sets their class to being a full star. This is done so that their starting class does not have an impact, which allows for the rating stars to be changed again, even if they are already full. Then it calculates the number of stars that should be full. This is done by truncating the average rating, so that it is the nearest whole number. Following this, if there's a difference between the number of full stars, and the average rating given, it sets the next available star to being half full. This is because any difference between the average rating, and the number of full stars means that the average rating has a decimal value. If a half star is added, it then increases the number of full stars variable by one, so that when the empty stars are set, it does not remove the half star. Finally it iterates through the remaining stars, and sets them to being empty. This is done by iterating through from the index of the number of full stars (because arrays are indexed from 0, this is then the one after the last edited star), to the last one, and setting their class to being an empty star. This changes the stars to the correct format, and as the JQuery selector is used throughout, it updates the stars directly. | | |

## 

#### Page switching/loading

| **Switch the content displayed on the page (with getting the name from a button)** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
|  | | |
| This changes the content displayed on the webpage. It can either take an element, which it can extract the text from (for use with buttons), or with a link name passed explicitly. If this link (found or passed) is the blank, it is set to home, as a blank string from the URL is the home page. The file name is constructed from this string, as the html files are called the same as the link name, with any spaces converted to underscores, converted to lowercase, with the /html/ folder prefix, and the .html file extension. This can then be directly passed into the changePageContent function, which issues the AJAX request. Finally, it updates the URI to reflect the new page. | | |
| **String manipulation**  This method uses string manipulation to find the file name to request from the server. It uses lowercase conversion to find the file name, and character replacing as well. It then uses concatenation to add the folder and the file type to it. | | |
| **Pages should be switched and navigated between without refreshing the page**  This procedure helps to meet this objective, as whenever it is called, the page should be switched. Then as part of that, it finds the html file it should access, and loads it, including adding it to the page, and updates the URL to reflect the page change. Consequently it helps meet this objective. | | |

## 

| **Reload the current page** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
|  | | |
| This function reloads the current page (or loads it initially). It gets the link name, which will be in the title case. First it splits this at the slashes, to get the component parts of the page. The first element of this array is then the page, and the second is either the name or ID of the thing that is to be loaded. If the page is a genre, author, or book, it uses the corresponding switch page method, as they involve more steps, including getting the HTML file, and requesting the about data separately. If it does not do meet these, it performs the standard page switch, with the original page URL, with the file left untouched. This will also load the current page, by calling that page's load function, through the switchPageContent function, then through the changePageContent function.  This is required, as the page does not load in the normal way. When the AJAX request is made, the content cannot be filled in using an event handler, and must be loaded when the request has been made. These pages also do not work directly, as they may need to make subsequent AJAX requests, and put the content onto the page. While these each have a function to do this, it is easier to have a single function that can be called to load the correct page. This does this correctly, and can be used for both refreshing and loading the page initially. | | |
| **Pages should be switched and navigated between without refreshing the page**  This function helps meet this objective, as it allows the page content to be loaded/reloaded whenever required. This is required as changing pages will not cause the content to change automatically, and needs to be inserted into the index.html template page. This handles this process. | | |

## 

| **Switch the page (making the AJAX request)** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
|  | | |
| This function is the one that actually changes the page. It issues the actual AJAX request, and changes the contents of the displayed page. It takes the file name (the file path, including folder name and file extension), whether it is asynchronous, and either the link name or the element that is to be used.  First it updates the currentPage global variable to the linkName passed into the function. This is so that the current page is known, and recorded. This is so that it can be accessed in the future, and the page the user is on is tracked, which is essential as the page does not physically change. Following this, it performs the GET request for the specified file, using the AJAX method from JQuery. This request also changes the asynchronous variable, so that the request can be done synchronously, or asynchronously, depending on the parameter.  When the request succeeds, it replaces the contents of the <main> section of the index.html file, with the webpage that was returned by the request. This automatically renders the new HTML content when it is inserted. However, if the request fails, it inserts the response text into the <main> section. For example, if a 500 error occurred, it would display 500 Error only, whilst keeping the header and title sections of the page visible. This should make it clear to the user the error that has occurred. It is important to note that it should not be possible for a 404 error to occur, as the page navigation only handles explicit links that are either pre-known or are found using data from the database.  Regardless of success, or failure, it changes the active link (the button in the navigation that is highlighted), moves the user back to the top of the page (in case they had scrolled down slightly), and reassigns handlers. Reassigning handlers is important as some pages have links that the user should be able to click, and reassigning these enables these links to work correctly. They are for navigating to the genre, author and book about pages. Finally, it clears the search bar, if the page was not the search. This ensures that it is wiped every time the user navigates to another page, which helps improve the user experience, and makes it more logical to use. However, if the page being switched to is the search page, this should not be performed, which is avoided through the use of the if statement, as if it was cleared there would not be an easy way to tell what the results being displayed were. | | |
| **AJAX and client-server model**  This function uses AJAX (asynchronous JavaScript and XML) to fetch data from the server. This allows for data to be accessed without a page refresh, which offers a superior user experience. Furthermore, by requesting data from a server, from a client, this is an example of a client-server model. | | |
| **Pages should be switched and navigated between without refreshing the page**  This function helps to meet this objective, as it requests the new page content from the server, and inserts the resulting HTML into the existing webpage. The AJAX gets the data without reloading the page, and by inserting this directly into the HTML, it changes the content of the page without refreshing the page. This should therefore give a much smoother user experience. | | |

## 

| **Change the active page button in the navigation bar** | | |
| --- | --- | --- |
| The website must be easy and intuitive to navigate for third parties. | Met |  |
|  | | |
| This procedure changes the active link in the navigation. The button to the current page should be highlighted in blue, in order to make it clear which page the user is on. Throughout this project, this is referred to as the active link. Consequently, when a user clicks on a different button to change the page, the active link needs to be changed to the corresponding page, which is what this procedure performs.  First it removes the active class from all of the buttons in the header. This removes the highlight from the previously selected button, without needing to know which button was selected. It takes two parameters, the element that was clicked, and the link content. Only one of these needs to be specified. If the element is specified (which takes a JQuery selector), it adds the active class to it. However, if the link content is specified, and the element is not (null is passed), it iterates through all of the buttons and changes the one whose text matches the specified link content to being active. The element approach is preferred as it is significantly faster, but isn’t always applicable. The element can be used within the onclick, as the target is known. However, when it changes the page to browse, when either the search is used, or a genre, book or author about page link is clicked, the element is not known, as the browse button should be highlighted, but the button will not be explicitly pressed. | | |
| **The website must be easy and intuitive to navigate for third parties.**  This function helps to meet this objective, as it ensures that the current page is highlighted. Doing this ensures that it is clear which page the user is currently looking at. This should help improve the navigation around the site, and make it simple and intuitive. Consequently, this is | | |

## 

| **Perform the function that loads the current page** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
|  | | |
| This procedure performs the function required to load the current page. It first checks whether the user is required to be signed in for the query, which also shows the sign in popup. When it is shown through this, it automatically redirects back to the home page. The correct page is then executed, which is found using a switch statement, matching the supplied link name. | | |
| **Switch statements**  This uses a switch statement to find the correct page function to load. This is more concise than using a long if-else selection, and is also more readable. For each of the possible cases that require a page load, it has the corresponding page function. Not all the possible pages in the project have an associated load function, so can be ignored, which is why they are not all present in the switch statement. The lack of a default option is because some of the pages do not require a custom loading method. | | |
| **Pages should be switched and navigated between without refreshing the page**  This method helps to meet this objective, as it allows the correct page load function to be executed only through knowing the link. This is required to meet this objective, as it cannot use an onload event handler, as there is only one JavaScript file for the whole site, which is loaded once, and the pages are not changed by changing the HTML file, but rather are done using AJAX, so the onload needs to be triggered manually, which is what this function performs. | | |

## 

| **Assign navigation bar page switching handlers** | | |
| --- | --- | --- |
| Pages should be switched and navigated between without refreshing the page | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
|  | | |
| This code assigns the onclick handler for all of the navigation links. For each of the buttons in the bottom half of the navigation bar (selected through the JQuery selector), it assigns the onclick function. When the button is clicked, it then calls the switch page content procedure, using the element to specify the target page. This also changes the active link highlight, through the switchPageContent procedure.  This does not need to be reassingend when the pages are switched between. This is run initially, when the website is first loaded, regardless of which page. This is because it runs as soon as the index.html file is loaded, and the script linked in it executed. When the page is then switched, the navigation section, including the buttons, is not changed. Consequently, the event handlers on each of the buttons are not lost when the pages are switched between, so it does not need to be reassigned. | | |
| **Event listeners**  This code uses event listeners, to switch the pages. It adds an on click event listener to each of the buttons in the navigation to change to the corresponding page on the website. | | |
| **Pages should be switched and navigated between without refreshing the page**  This code helps to meet this objective. It means that pages can be navigated too, adding functionality to the buttons in the navigation. To change pages, it also uses the switchPageContent procedure, which uses AJAX so does not refresh the page, so consequently meets this objective. | | |
| **The home page should have links to the other main sub-pages so that users can navigate easily around it.**  This code helps to meet this objective, as it adds the functionality to change the page that is being displayed on the page. This means that the links which are displayed on the home page, and also on any other page on the site, can direct users to any other main sub-page on the site. | | |

#### Account Management

| **Check whether a user needs to be signed in for the desired page** | | |
| --- | --- | --- |
| Recommendations and reading list entries must be kept private. | Met |  |
| Each user’s diary entries must be kept private | Met |  |
|  | | |
| This function checks whether a user needs to sign in to view a specific page. If the page is the My Books, Recommendations, or Diary entry page, the user needs to be signed in to view it. If the link supplied to the function is within the list of these page names, and the sessionID global variable is not specified (null), the user needs to be signed in. In this case, it displays the sign in popup to prompt the user to sign in. It also shows the sign in notice, which is inline element in each popup, which tells the user that they need to sign in (or sign up depending on the popup). This is shown to add explanation as to why the popup is shown. Following this, it returns true, as sign in is necessary. Otherwise, it returns false, because sign in is not necessary. This can occur, when the page is not one of the ones specified, or the session ID global variable is not null (the user is signed in).  Having this function run every time one of the pages which contain only user-specific content is visited, it ensures that the user is signed in. It also makes it harder for errors to occur within the backend, with fewer errors to handle, as it prevents the user trying to get data with a session ID of null. It also allows for a very quick and simple way to check whether the user should be switched back to the home page if they cancel the sign in. | | |
| **Array searching**  This function uses a searching algorithm to find whether the link appears in the string. It searches through an array containing the page names that require a sign in, to find the link given to the function. It uses the built-in search to determine whether the current page is in the array. | | |
| **Recommendations and reading list entries must be kept private.**  **Each user’s diary entries must be kept private**  This function helps meet these objectives. It ensures that the user cannot access the recommendations, my books (reading lists), and diary pages without being signed in. This allows for the content for these pages to be user specific, and limited depending on the signed in user. | | |

## 

| **Hide any sign in/sign up pop ups** | | |
| --- | --- | --- |
| Recommendations and reading list entries must be kept private. | Met |  |
| Each user’s diary entries must be kept private | Met |  |
|  | | |
| This procedure hides all sign in and sign up popups that are displayed. It is done as a seperate function, so the event handlers for clicking the cancel button, and clicking outside of the popup, can close the popup for both the sign in and sign up popups. It also checks whether the popup can be hidden.  First it checks whether the popup can be hidden. This is done by checking the global disablePopupCancel variable, which is set to true if the sign in and sign up popups cannot be hidden. This can be whilst the client is waiting from a response from the server, so that any issues while signing in or up, such as invalid credentials, can be shown. It also prevents the user from seeing the webpage before the user-specific content is loaded onto it. If it is false, and the popup should be closed, the procedure does nothing. This does not close the popup, and therefore the cancel button, and clicking outside of the popup does not have an effect. It does not effect the these methods, however, so they will work the next time they are pressed, assuming the disablePopupCancel variable changes to being false between the clicks.  If the disablePopupCancel variable is false, it can close the popup. It checks whether the current page requires a sign in, and whether the user has been signed in. These pages are the my books, recommendations, and diary entry pages. They only contain user specific content, so cannot be shown unless a user is signed in. If the current page is one of these pages, and the user is not signed in, they are redirected back to the home page using the switchPageContent procedure. This check is done first, as it allows the request for the home page to be sent before the popup is hidden, which should improve the responsiveness, as the page switching is done asynchronously. After switching the page, it hides both of the popups, using the JQuery hide method. This is done by selecting the window class on the page, and as both popups are within a div of this class, they are both hidden. Only one of these popups is actually shown at a time, but as the selector can select both, and performing the hide method on already hidden elements has no effect, it is done for both. This also allows this single procedure to hide both the sign in and sign up popup.  Following this it removes the any alerts shown within either popup, so they can be shown again later, without any alerts still showing, although this is very unlikely to happen. Finally, it hides the text which says that the user has to sign in to view the page. This again allows the popup to be shown directly.  Redirecting to the home page ensures that users cannot view the page if they are not signed in, and therefore cannot see the page if it does contain any data. This makes it easier to format the page, and simpler to use too. This redirection also has an additional benefit of making it impossible for the get\_data requests to be made with a session ID of null. Avoiding this reduces the amount of processing that needs to be done when the requests are made, as this does not need to be checked for, and responded to accordingly, as this JavaScript makes it impossible for it to happen. | | |
| **Recommendations and reading list entries must be kept private.**  **Each user’s diary entries must be kept private**  This function helps meet this objective, as it ensures that the user cannot access pages that only contain user-specific data on them without signing in to them. This helps keep recommendations, reading lists and diary entries private, and user specific. If they then try to cancel, and access the page without signing in, they are redirected to the home page, which further helps prevent it. | | |
| **Possible Improvements:**  If sign in was required for the page, and it is then hidden, without the user having signed in (the user has cancelled the sign in), they are redirected back to the home page. This is a suitable solution, however it could be improved. Instead of taking the user back to the home page, it could take the user back to the last page that they visited, which did not require a user to be signed into. This could improve the user experience. This was not done, as the benefit would be marginal, and was outside of the project requirements. If this were to be implemented, this procedure is the one that would need to be alterend to incorporate this. | | |

## 

| **Change sign in/sign out button visibility** | | |
| --- | --- | --- |
| Users must be able to create their own accounts on the website. | Met |  |
| Users must be able to log in to the website to access their personalised information. | Met |  |
|  | | |
| This procedure changes the visibility of the sign in, sign up and sign out buttons. When there is not a user signed in, the sign in and sign up buttons should be shown, and when there is a user signed in, the sign out button should be shown only. This procedure changes between these two possible states.  If the sessionID variable is not null, there is a user signed into the website. In this situation it hides the sign in and sign out buttons (which have the account-enter class), and shows the sign out button (which has the account-exit class). If the session ID variable is null, it then shows the sign in and sign up buttons (by removing the hidden class from them), and hiding the sign out button by adding the hidden class to it. For both of these, it does not check whether they are hidden or not initially, as adding and removing classes through JQuery does not have an effect if the changes that it would make are already applied. This enables the single procedure to toggle between the two, without any knowledge of their current state. | | |
| **Users must be able to create their own accounts on the website.**  This code helps to meet this objective, as it changes the visibility of the sign in, sign up and sign out buttons, so that where applicable, the users can access the form to create their own accounts. | | |
| **Users must be able to log in to the website to access their personalised information.**  This code helps to meet this objective, as it changes the visibility of the sign in and sign out buttons. It allows for the user use the sign in button when they are not signed in, and then access any of their personalised data using the session ID that they should recieve. It also then allows the sign out button to be displayed which enables them to sign out of the website, which, although not part of the objective explicitly, is implied. | | |

## 

| **Close a popup by clicking outside of the popup** | | |
| --- | --- | --- |
|  | | |
| This code closes both the sign up and sign in popups when the user clicks outside the popup. It does this by assigning an event listener, for when the user clicks within the window. Whenever the click occurs, it checks whether the target of the click is either the sign in window, or the sign up one. If it is, and the disablePopupCancel flag is false (so the popup can be hidden), it then hides the popup. This code was inspired by that from (w3schools, 2019). It was written here to use JQuery, as it is used extensively throughout the project, so can simplify the code, adds checks for whether the popups can safely be hidden, and allows for multiple popups to be processed using a single event handler. The concept, as given by (w3schools, 2019), is still the same however.  This checks whether the click is within either the sign in or sign up popup, and also uses the hideAllSignPopups procedure. This ensures that this works for both the sign up and sign in popup, without needing to create handlers for each of them separately, which is simpler, and likely to be faster, as there are fewer handlers firing every time the user clicks.  Furthermore, this runs when the page loads initially. This immediately sets the handler from when the page loads. When switching pages, the window is never completely refreshed, so the handler remains persistent for the entire time the page is loaded. Consequently, it does not need to be done as its own separate function, as it never needs to be reassigned. | | |
| **Event handler**  This uses event handling. Whenever the user clicks anywhere within the window, the function is then executed to find whether the popup should be closed or not. This is done using the JQuery onclick handler, which is attached to the window, and fires whenever the window is clicked anywhere. | | |

## 

| **Handle when a session has expired** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
| Recommendations and reading list entries must be kept private. | Met |  |
| Each user’s diary entries must be kept private | Met |  |
| The website should be able to save a user’s login | Met |  |
|  | | |
| This procedure handles when a session has expired, and should be executed whenever the session ID that is being used by the client has expired. The client finds this out when an AJAX request is made, which requires the session ID, and a 403 status code is given. In these scenarios this procedure should be called to handle it.  First it sets the sessionID global variable to null, to indicate that there is not a user signed into the page. Following that, it checks if the current page needs the user to be signed in, and if it does, it displays the sign in the popup. It also changes the account buttons, so that the sign in and sign up buttons are displayed instead of the sign out one. This gives the user a way to sign back into the website, and also makes it clear that they are not currently signed in. Finally, it removes the session ID cookie, so that next time they visit the page, they are not automatically signed in again next time they visit. | | |
| **Cookies**  This method uses cookies. It uses the js.cookie module to simplify their usage. Whenever the session expires, the recorded session ID cookie is removed, so that it can no longer be used, such as the next time the user signs into the website. It removes the cookie if it exists, and if it does not exist, it does not cause any errors. | | |
| **Users must be able to log in to the website to access their personalised information.**  This function helps to meet this objective, as it ensures that sessions are closed after a long time. This helps to ensure the account remains secure, so that it cannot be accessed indefinitely. | | |
| **Recommendations and reading list entries must be kept private.**  **Each user’s diary entries must be kept private**  This code helps to meet these objectives, as it ensures that sessions are closed after a long time of inactivity, which should help to ensure that the account remains secure. This will help to keep recommendations, reading lists, and diary entries private, and consequently helps to meet these objectives. | | |
| **The website should be able to save a user’s login**  This objective helps to meet this objective, as it allows a cookie to be used to store the session ID. If the session ID was able to be used indefinitely, it would be a security risk, so it expires after one week. The cookie expires by default after a week, but if it is used outside of this, | | |

| **Show and hide alerts when signing in or creating an account** | | |
| --- | --- | --- |
| Users must be able to create their own accounts on the website. | Met |  |
|  | | |
| These procedures show and hide alerts that should be shown to the user while they are signing up. Their purpose is to make it clear to the user what errors have occurred while they were trying to create an account. The following table shows the different errors that can occur, and the corresponding error messages that should be shown.   | The password, and the repeated password do not match | Passwords do not match | | --- | --- | | The username entered is already in use | Username is already taken | | A server error occurs (500 error) | Something went wrong |   Both of these procedures use generic JQuery selectors, not specifying whether the alert is within the sign in or sign up popup. This enables the single procedure to be used across both signing in and signing up, which was an active decision to reduce the amount of code required, and helping to avoid code repetition. It also helps to simplify handling both signing in and signing up, as they can use more similar code to each other to work.  **Showing the alert**  It first finds the alert elements and assigns it to a variable, so that it can subsequently be modified. It changes the content of the element to the given message, which can be any of the ones given in the table. The element is then shown, as it defaults to being hidden in the HTML. Finally it creates a procedure that executes on a timer. After 8 seconds from when the alert is created, it is then hidden, by fading it out over half a second. This is done using the built-in setTimeout method. Eight seconds was chosen for the timeframe for this timeout, as it gives enough time for the user to read the message, while being short enough to have hidden by the time they resubmit the form.  **Hiding the alert**  The second procedure hides the alert, and overrides the timeout. It does this by selecting the same element as used in the procedure to show the alert, through the JQuery selector, then hiding it immediately, using the JQuery hide method. If this is done within a timeout set within the show alert procedure, it hides the element anyway, and when the timeout triggers, it then does not have any effect, so does not need to be explicitly cancelled. | | |
| **Delays and timers**  This function uses a delay/timer to show the message for the user, with enough time for them to read it. After this eight second delay, it then hides the message. | | |
| **Users must be able to create their own accounts on the website.**  This helps to meet this objective, as it gives the appropriate error messages to the user while they are signing up. This enables them to know what any errors are with their account creation, which then allows them to create their own account following the required conditions simply and easily. | | |

##### Sign Up popup

| **Show the Sign Up popup** | | |
| --- | --- | --- |
| Users must be able to create their own accounts on the website. | Met |  |
|  | | |
| This procedure shows the sign up popup. It does this by hiding the sign in popup, and showing the sign up one instead. It hides the sign in popup, as the sign up popup could be shown by navigating through the sign in one, so therefore needs to be hidden. It can also be hidden directly, as the hide method does not throw any errors if the popup is already hidden. This removes the need to check for whether it is hidden or not, so makes the code significantly simpler, and can be performed every time. | | |
| **Users must be able to create their own accounts on the website.**  This procedure helps to meet this objective, as it shows the sign up popup, regardless of whether the user has navigated from the sign in popup, or from any of the other pages on the site. This then allows the user to insert their own data, and account credentials, so that they can actually create an account. | | |

| **Send sign up request** | | |
| --- | --- | --- |
| Users must be able to create their own accounts on the website. | Met |  |
| The website should be able to save a user’s login | Met |  |
|  | | |
| This code assigns a handler to the sign up form, which runs every time the user presses the sign up button at the bottom of the popup. As part of this, it handles all of the error checking (client side), and performs the AJAX request, handling both success and failure of the request. It uses JQuery to add an event handler to the form, so when the sign up form is submitted it runs the set code. This handler does not need to be reassigned, like the author navigation handlers, as the navigation bar is never reset when pages are switched between. This means that once this code has run, which it does as soon as the script is added to the page, the handler is created, and is persistent until the page is closed.  When the submit button is pressed (the contents of which is Sign Up), it first prevents the default response to the form submission. This needs to be prevented, as it would normally send a get request directly, without proccessing the user’s data. It then disables the ability for the user to close/cancel the popup by setting the global disablePopupCancel flag to true. This prevents it from being closed until a response has been received or it is known to be an invalid entry.  Having disabled the cancel methods, it then gets the passwords (the actual password, and the repetition of it. Following that it checks whether the two entered passwords are the same. If they are not, it alerts the user that they do not match, using the signUpAlert procedure, then enables the cancel methods by setting the disablePopupCancel flag to false. When it gets the passwords, it also gets whether the user should be remembered. It does this by getting the value of the check box, and converting it to a boolean value. This works because of the selector either gets the ‘Remember me’ if it is selected, or undefined if it is not selected. Consequently, casting this value to a boolean gives true if its selected, and false if it isn’t.  If the passwords do match, and if they are, it performs the AJAX request. Doing this within the else prevents the form being submitted if the passwords do not match, and also re-enables the cancel methods, which allows the form to be exited, and treated as if it were not submitted. This is why the AJAX request is within the else. The AJAX POST request is then made, with the URL for it specifying the WSGI script, and the accounts handler, and the sign up function. It sends the firstname, surname, username and the password within a JSON object too. It gets each of the components, by getting the value of each of the corresponding inputs.  If the sign up is successful, it re-enables the methods to close the popup by changing the disablePopupCancel flag to false. If the session ID part of the resulting JSON object is not null, the sign up was successful. It checks for this, and if it is given, it sets the global sessionID variable to the returned sessionID, so that it can be used for subsequent requests. It also changes thee account popups, so the sign in and sign up buttons are hidden, and replaced with a sign out button. It also reloads the current page, so that any user-specific data can be shown. Finally, it checks whether the rememeber me button was clicked. If it was, it creates a cookie that lasts for a 50 days, which is secure too. The duration of 50 dayswas chosen as it expires after a long time, but is not indefinite. Each time a session ID is used, its creation time is updated, and any sessions that have not been used in a week are then closed. The combination of the two allows sessions to be kept as long as they are regularly used, for upto 50 days after the user signed in. The cookie contains the session ID. Saving the session ID then allows for it to be used next time the page is loaded, and therefore saves the user’s login. If the session ID is not given in the result, it then shows the message given by the server, such as telling the user that their desired username is taken.  If the AJAX request fails, there must have been an internal server error, as the only possible status codes are 500, or 200 (404 cannot occur, as the URL is hardcoded, and the other status codes can only be given by the WSGI, which is hardcoded to give 200 OK). The 500 error should not occur, but it is handled in the event that it does, so the user knows what happened. It uses the signUpAlert procedure to alert the user than something went wrong, and re-enables the methods to close the popup. | | |
| **AJAX, HTTP requests and client-server model**  This code uses AJAX to make HTTP requests to the webserver, utilising a client-server model. The AJAX makes a POST request, sending the new account data to the server, so that a new record can be made in the database for it. It then responds with the new session ID which can then be used by the client. | | |
| **Event handling**  This code uses event handling to provide a custom response to the form submission, and issue the correct AJAX request, and also provide the correct user feedback if there are any errors with the sign up request. | | |
| **Users must be able to create their own accounts on the website.**  This code helps to meet this objective as it takes the data entered into the sign up form, and sends it to the webserver, which can then create a new account in the database. Consequently, between this, the HTML form, and the WSGI script handling the request, this objective is fully met. | | |
| **The website should be able to save a user’s login**  This objective helps to meet this objective, as it creates a cookie containing the session ID for that client, if the user wants to the website to remember them. The session ID then can be directly used in requests next time the user accesses the website, and therefore, allows the website to remember the user. | | |

##### Sign In popup

| **Show the sign in popup** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
|  | | |
| The first procedure adds an event handler, which triggers whenever a button with the ID sign-in-button is pressed. There are two buttons on the website that link to the sign in popup, and that is the one in the top navigation bar, and that at the top of the sign up popup. Both of these links are given this ID,, so that they can both be given the same handler. For this reason, the JQuery selector selects all of the sign in buttons on the website. For all of them, it adds an onclick event to them, which calls the showSignInPopup porcedure. These run when the script first loads, and as these buttons are never remvoved, or altered, so the handlers remain forever.  The showSignInPopup procedure shows the sign in popup. It does this by selecting the popup window through a JQuery selector, and using the show method to display it. When testing this, it was clear that the hide method is not required for the sign up one in this case, but is required for showing the signUpPopup correctly. | | |
| **Event handling**  This uses event handling to show the sign in popup whenever a sign in button is pressed by the user. This enables the form to be used, and the user to sign in correctly. | | |
| **Users must be able to log in to the website to access their personalised information.**  This code helps to meet this objective, as it shows the sign in popup whenever a sign in button is pressed, anywhere on the website. This allows them to sign in to the website, and therefore access their personalised information. Consequently, in combination with the form submission and the WSGI responding to the request, this objective is met. | | |

| **Send sign in request** | | |
| --- | --- | --- |
| Users must be able to log in to the website to access their personalised information. | Met |  |
| The website should be able to save a user’s login | Met |  |
|  | | |
| This code adds an event handler to the sign up form, which runs whenever it is submitted by pressing the sign in button at the bottom of the popup. This handler is created when the script runs initially, but as the form is never overwritten or modified, it doesn’t need to be reassigned later. It is always applied to the submit button.  When the submit button is pressed, it first preents the default behaviour of the form submission. If this was not done, it would perform a GET request. Following this, it disables the ability for the user to close the popup, by setting the disablePopupCancel flag to true. This is then changed once a response is recieved by the client. Following this, it gets whether the user’s login should be saved, in the same way as the sign up form.  Finally it issues uses AJAX to issue the POST request, directing it to the sign in function, as part of the accounts handler in the WSGI script through the URL, as explained in the documented design. Within the post request, it sends the username and password for the attempted login. The username and password are found by getting the value of the username and password inputs respectively. This is done by using a JQuery selector to select the corresponding input, and the .val() method is then used to get the contents of each input. They are then created into a JSON object, which is then converted to a string so it can be sent within the POST payload.  If the POST request is successful (the status code is not an error one, 200 codes), it re-enables the methods to close the popups, by setting the global disablePopupCancel variable to false. If the session ID part of the resulting JSON is not null, the sign in request was successful, as a session ID could be generated for the user. This also means that the credentials used were valid. If the session ID is specified in the result, it then sets the global sessionID variable to the transmitted session ID so that it can be used in other requests across the website. It then changes the account buttons, so the sign in and sign up buttons are hidden, and the sign out one is shown instead. Next, it hides the popups, and then reloads the current page, so that it displays any user-specific content. Finally, if the user had selected to remember their account, and sign them in automatically next time that they visit, it then creates a cookie to store the session ID. The cookie that it creates expires after 50 days, which, in combination with the sessions class in the backend, ensures that a session can be used for up to 50 days before a user has to sign in again, assuming that they use their account every week. However, if the session ID is null in the response, the credentials must be incorrect, so consequently alerts the user to the error, by displaying the error message given from the WSGI script using the signUpAlert method. This enables the user to know when their login attempt was invalid, and the error that occurred.  If the POST request fails due to an error, it alerts the user to this, by outputting Something went wrong in alert, using the signUpAlert procedure. This is only ever going to be triggered by a 500 error, as the URL is hardcoded, so will always exist, and any other status codes must be hard-coded into the WSGI, which they are not. The 500 error should not ever occur, as the backend is robust, and should handle any errors that can be thrown at it. However, this is included here in case an error did occur, so the user knows that an error has occured backend. Following the alert being shown, it then re-enables the cancel buttons by changing the disablePopupCancel global variable to false. | | |
| **AJAX, HTTP request and client-server model**  This code uses AJAX to issue a HTTP request to the webserver, which also uses a client-server model. The AJAX issues a post request, which sends the user’s credentials to the server, so that it can be checked against the credentials stored in the database, and generate a new session ID if it is valid, and give a message if it not. This session ID can then be used by the client in subsequent requests. | | |
| **Event handling**  This code uses event handling to override the default behaviour of a HTML form when it is submitted. It allows for the AJAX request to be made, without reloading or changing the page, targetting the correct function and handler, and handle the returned data correctly. | | |
| **Users must be able to log in to the website to access their personalised information.**  This code helps to meet this objective, as it allows users to send their login credentials to the webserver, and receive a session ID, which can be used to request data later. The credentials are checked on the server, and an appropriate message given if anything occurs that means that the user cannot sign in. Consequently, in combination with the WSGI responding the to request, this meets this objective | | |
| **The website should be able to save a user’s login**  This code helps meet this objective, as it creates a cookie that saves the session ID, if the user selects for the website to remember them. This cookie can be used when the page is loaded again in the future to make requests using it again, removing the need for the user to sign in. In combination with the WSGI, accounts part of the components module, and the maintenance script, this code meets this objective. | | |

##### Sign Out

| **Send sign out request** | | |
| --- | --- | --- |
| The website should be able to save a user’s login | Met |  |
|  | | |
| This code assigns the an event handler for when the sign out button is pressed. It runs initially, when the webpage first loads, which assigns the handler. The sign out button isn’t removed at any point after the page is loaded, or replaced, as only the content in the <main> section of the webpage is replaced. This means that once created, the handler stays applied to the button, so does not need to be reapplied, and consequently does not need to be made as a procedure.  First it uses AJAX to issue a POST request, to the sign out function within the accounts handler, within the WSGI script. It sends the session ID as the payload of the request, so that it can be closed. The request is sent so that the session can be removed from the database. This is not essential, so the client does not need to react to it, regardless of whether the request succeeds or fails. This is because it does not impact the client, or security, as once the client signs out, the session ID is lost. The request is made to the server so that the session ID can be removed from the database, which helps to keep the database maintainable. If it fails, it will be subsequently removed by the maintenance script in the future, which is why it does not matter to the client if it succeeds or fails. After making the HTTP request, it then sets the session ID global variable to null, to indicate that there is not a user signed in to the website. Following that it changes the account buttons to hide the sign out button, and replace it with the sign in and sign up ones, as there is no longer a user signed in, and therefore another needs to be able to sign in on the same client. Next it reloads the page. This removes any user-specific content on the page and prompts the user to sign back in if the page they were on when the sign out button was pressed requires sign in. If they cancel the sign in, they are redirected back to the home page automatically. Finally, it removes the session ID cookie, so that all record of the session ID is removed from the client.  The processing that occurs after the HTTP request is made using the AJAX is done outside of the JQuery ajax function, rather than using the complete option within it. This is because using the complete function within the AJAX method waits for a response from the server before running. This introduces delay, and as the response, success or faliure, is irrelevant, it can be done outside of the AJAX function for it to be faster, and improve the responsiveness of the site. | | |
| **AJAX, HTTP requests and client-server model**  This code uses AJAX, through JQuery, to issue HTTP requests to the webserver, through the client-server model used by the browser. The AJAX makes a POST request which sends the the session ID used by the client to the server so that it can be closed, and removed from the database. | | |
| **Event handling**  This code uses event handling, as it assigns a function to the sign out button, which is run whenever it is clicked. It sends a request to the server to close the session, and removes all user-specific data, and all record of the session ID from the client. This signs the user out of the webpage when the button is pressed. | | |
| Why the objective is met | | |

#### My Books (reading list) pages

| **Load the my books page** | | |
| --- | --- | --- |
| Users should be able to create their own custom reading lists | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| Users should be able to create their own custom reading lists | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
| The users’ reading lists must be available on a dedicated webpage on the website. | Met |  |
|  | | |
| This function loads the my books (reading lists) page. To do this, it needs to perform a series of different processes, which | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign event handlers for changing between reading lists** | | |
| --- | --- | --- |
| The user should not be able to delete the default *Have Read*, *Currently Reading*, and *Want to Read* reading lists | Met |  |
| Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list | Met |  |
| Each entry into a reading list should allow the user to delete the entry. | Met |  |
| Each entry into a reading list should supply a link to the books about page. | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign delete a reading list handlers** | | |
| --- | --- | --- |
| Users should be able to delete their own custom lists | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign delete reading list entry handlers** | | |
| --- | --- | --- |
| Each entry into a reading list should allow the user to delete the entry. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handlers to move entries to different lists** | | |
| --- | --- | --- |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. | Met |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. | Met |  |
| Users should be able to record the books that they have read in the past. | Met |  |
| Users should be able to record the books that they are currently reading. | Met |  |
|  | | |
| Want to Read -> Currently Reading  Currently Reading -> Have Read | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Genre about pages

| **Switch to a genre about page** | | |
| --- | --- | --- |
| Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| Each genre should have an about page | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign navigation handlers** | | |
| --- | --- | --- |
| Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system. | Met |  |
| Each genre should have an about page | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Book about pages

| **Assign navigation handlers to switch to an about page** | | |
| --- | --- | --- |
| Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system. | Met |  |
| Each related book should link to the corresponding about page. | Met |  |
| Each entry into a reading list should supply a link to the books about page. | Met |  |
| Each diary entry should supply a link to the corresponding book about page. | Met |  |
| Each book summary should link the specific books about the page. | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Switch to the about page for a book** | | |
| --- | --- | --- |
| The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book | Met |  |
| A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website. | Partially Met |  |
| It should have a method to add the book to the have read, currently reading, and want to read lists. | Met |  |
| There should be a list of books that are related to each other displayed. | Met |  |
| Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Change the number of summaries depending on the window size** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Add/move a book to a reading list** | | |
| --- | --- | --- |
| Users should be able to record the books that they have read in the past. | Met |  |
| Users should be able to record the books that they are currently reading. | Met |  |
| Users should be able to record the books that they would like to read in the future. | Met |  |
| Reading lists, and the entries within them should be specific to the user. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Hide the reading list popup** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign follow and unfollow author handlers** | | |
| --- | --- | --- |
| Users should be able to follow each author. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handler to delete a review** | | |
| --- | --- | --- |
| Users should be able to delete any reviews that they have left | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign review stars handlers for when they are clicked** | | |
| --- | --- | --- |
| Reviews must have an overall rating. | Met |  |
| Reviews can have a character rating, plot rating, review summary, and review body. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Submit a review for a book** | | |
| --- | --- | --- |
| Users should be able to leave reviews about each book | Met |  |
| Users should be able to delete any reviews that they have left | Met |  |
| Reviews should be listed on the corresponding book about page. | Met |  |
| Reviews must have an overall rating. | Met |  |
| Reviews can have a character rating, plot rating, review summary, and review body. | Met |  |
| Reviews on the book should show the user’s username. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Display an alert for when a review is left in an invalid format** | | |
| --- | --- | --- |
| Reviews must have an overall rating. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Author about pages

| **Simple overview** | | |
| --- | --- | --- |
| Each author should have a profile page. | Met |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Switch the page content to the author page** | | |
| --- | --- | --- |
| It should display the author’s average book rating. | Met |  |
| It should display all the genres that the author writes for. | Partially Met |  |
| It should show all the books that the author has written. | Met |  |
| It should supply an about the author section. | Met |  |
| It should show the author’s name (or the name that they write under if applicable). | Met |  |
| Users should be able to follow each author. | Met |  |
| The page should display how many followers each author has. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Diary page

| **Load a user’s diary entry page** | | |
| --- | --- | --- |
| The users’ diary entries must be available on a dedicated webpage on the website. | Met |  |
| Each diary entry should supply a link to the corresponding book about page. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handlers to delete diary entries** | | |
| --- | --- | --- |
| Users should be able to delete diary entries | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Hide the diary entry popup** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Show an alert for any issues with a new entry** | | |
| --- | --- | --- |
| Users must specify the number of pages that they read. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Leave a diary entry** | | |
| --- | --- | --- |
| Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating. | Met |  |
| Users must specify the number of pages that they read. | Met |  |
| When the user finishes the book, the user should be given the option to make the diary entry available as a review. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Home page

| **Add book summaries to DIVs on the home page** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Load the home page** | | |
| --- | --- | --- |
| The home page should give users a summary of their recommended books. | Met |  |
| The home page should give users a summary of what is in their want to read list. | Met |  |
| The home page should give users a summary of what is in their currently reading list. | Met |  |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. | Met |  |
| Book summaries should include the book’s cover image, author, and title. | Met |  |
| Each book summary should link the specific books about the page. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

| **Change the number of summaries shown in each row depending on window size** | | |
| --- | --- | --- |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Recommendations Page

| **Load recommendations** | | |
| --- | --- | --- |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
| Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre. | Met |  |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. | Met |  |
| Recommendation entries should have a method to move it to the *want to read* reading list. | Met |  |
| The users’ recommendations must be available on a dedicated webpage on the website. | Met |  |
| Recommendation entries should show the certainty of the recommendation | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Add a new author checkbox to the displayed list** | | |
| --- | --- | --- |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handler to send user’s initial perferences** | | |
| --- | --- | --- |
| When signing up for the first time, users should be prompted for their favourite authors and genres. | Partially Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handlers to delete a recommendation, and mark it as bad** | | |
| --- | --- | --- |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Assign handlers to move a recommendation to the Want to Read list** | | |
| --- | --- | --- |
| Recommendation entries should have a method to move it to the *want to read* reading list. | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Database search

| **Handler for when the search bar is used** | | |
| --- | --- | --- |
| There must be a search bar that searches for books, authors and genres within the website. | Met |  |
| The search bar should be able to search by name or ISBN number | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### Browse page

| **Switch to the browse page** | | |
| --- | --- | --- |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website | Met |  |
| The browse page should display books related to the last one added to a reading list | Met |  |
| The browse page should display books related to the last read book | Met |  |
| The browse page should display books from the authors the user is following | Met |  |
|  | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Simple overview** | | |
| --- | --- | --- |
| Objective | Met |  |
| Code | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

## 

| **Simple overview** | | |
| --- | --- | --- |
| Objective | Met |  |
| Code | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

#### 

### Proforma

## 

| **Simple overview** | | |
| --- | --- | --- |
| Objective | Met |  |
| Code | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |

| **Simple overview** | | |
| --- | --- | --- |
| Objective (12.62) | Partially Met?  (2.8) | 0.45 |
| Code screenshot | | |
| Explanation | | |
| Technique 1 | | |
| Why the objective is met | | |
| Improvements | | |

Rough outline

What is the objective

Did you achieve this → make it clear using explanation and discussion

How did you achieve it → make it clear → what techniques did you use

P1 - was the objective completed

* Write out the objective
* State if the objective was met or not
* main screen shot

P2 - what techniques were used

* Briefly explain how they were used

P3 - in depth first technique

* Screenshot to support the first technique
* Explain what the technique is
* How was this technique used
* Why did you use this technique

P4 - in depth second technique

* Screenshot to support the first technique
* Explain what the technique is
* How was this technique used
* Why did you use this technique

P5 - in depth third technique

* Screenshot to support the first technique
* Explain what the technique is
* How was this technique used
* Why did you use this technique

P6 - brief conclusion

* Round up your main parts

# Testing

To try and simplify the process of testing this project, the testing has been broken down into two parts. It has been split into backend components, and then the individual modules tested separately, as well as the front end components, which are then tested in combination with the backend. This also includes the wsgi.py file.

## Backend

When developing the project, unit tests were created for most of the modules. These were created to speed up the process of testing each individual part of the project. They do not work for all parts of the project, such as where database operations are used, but where applicable they were created. This significantly simplified the testing process, as many parts of the project had scripts that could be used to check the outputs of the code. Within the github repository, the testing folder contains these test files and any associated files required from them.

Any tests that are failed, and that can reasonably be rewritten are rewritten, and any amendments are then added to the technical solution chapter. They are then retested below the failed test.

### Test Database

This project needs a large amount of data to be useful, for the users. However, this is difficult to test and use, as to check the accuracy of the recommendations and the similarities between books, for example, a knowledge of all the genres needs to be known. This is not possible on a large scale project, so simple test database has been created. It features fewer genres, books and users. The MySQL files to create this can be found in Appendix D. Between each test, the database should be remade, so that the data is preserved between them. Some modules have additional data that is needed, which should also be added. Consequently, after each test the following should be executed within the database. The final command is optional, and is only needed for tests that have additional data.

| source /absolute/path/to/project/MySQL/create\_tables.sql;  source /absolute/path/to/projecttest\_data\_base.sql;  source /absolute/path/to/project/MySQL/module\_specific\_file; |
| --- |

| **Review** | **Book ID** | **User ID** | **Overall rating** | **Plot rating** | **Character Rating** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 5 | 5 | 5 |
| 2 | 1 | 3 | 2 | 3 | 1 |
| 3 | 2 | 1 | 3 | 2 | 3 |
| 4 | 2 | 2 | 4 | 2 | 5 |
| 5 | 2 | 4 | 3 | 3 | 4 |
| 6 | 3 | 2 | 4 | 3 | 4 |
| 7 | 3 | 3 | 1 | 1 | 2 |
| 8 | 3 | 4 | 3 | 2 | 4 |
| 9 | 4 | 1 | 3 | 2 | 5 |
| 10 | 4 | 2 | 3 | 3 | 4 |
| 11 | 4 | 3 | 1 | 2 | 3 |
| 12 | 4 | 4 | 4 | 3 | 5 |
| 13 | 5 | 1 | 1 | 2 | 1 |

Expected factors for each of the books and users are given in the following tables. These may very each time the model is fitted, but the relationships should remain similar to this. They should be approximately the same as these, but multiplied by a scale factor. They are generated randomly initially, so there will be a variation in them. The user factors are not stored, but the book\_factors are. These approximations are also given in the test data.

|  | **User 1** | **User 2** | **User 3** | **User 4** |
| --- | --- | --- | --- | --- |
| **Genre 1** | 0.0294956 | 0.09664164 | 0.26495274 | 1.25458025 |
| **Genre 2** | 1.54922344 | 0.27333539 | 0.12121422 | 1.088113 |
| **Genre 3** | 0.89430514 | 0.3921382 | -0.28712813 | -0.16836666 |
| **Genre 4** | 0.05814775 | 1.71102246 | 0.71086205 | -0.20507444 |
| **Genre 5** | 0.19796957 | -0.27786458 | 0.31624035 | 0.69514346 |
| **Genre 6** | 0.12207968 | 0.72508255 | -0.43121506 | -0.09972446 |
| **Genre 7** | -0.34043115 | -0.70024375 | 0.28556009 | -0.29876263 |
| **Genre 8** | 0.28192704 | -0.80572655 | 0.53219999 | -0.92930723 |
| **Genre 9** | -0.06565732 | 0.64759103 | 0.86344873 | 0.52843753 |
| **Genre 10** | 0.75296772 | 0.14208509 | -0.27026705 | -0.08855528 |

|  | **Book 1** | **Book 2** | **Book 3** | **Book 4** | **Book 5** |
| --- | --- | --- | --- | --- | --- |
| **Genre 1** | 0.09616839 | -0.11439516 | -0.34141417 | 0.89826349 | 0.83968715 |
| **Genre 2** | 0.13990453 | 0.4602393 | 0.74010668 | 1.67033071 | 0.3026473 |
| **Genre 3** | -0.19446672 | 0.31336525 | 0.874315 | 0.34160336 | -0.21344638 |
| **Genre 4** | 0.52190524 | 0.00427231 | 1.24445128 | -0.55935626 | 1.1563066 |
| **Genre 5** | 0.27685214 | 0.03817313 | -0.32515329 | 0.68218074 | 0.22312092 |
| **Genre 6** | -0.47304706 | -0.01841387 | 0.63977702 | -0.15577632 | 0.2789436 |
| **Genre 7** | 0.35159769 | -0.0327723 | -0.61130751 | -0.26834166 | -0.45117512 |
| **Genre 8** | 0.7532494 | 0.30054946 | -0.13086895 | -0.33200523 | -1.04237929 |
| **Genre 9** | 0.68760675 | -0.05232584 | 0.18113549 | 0.15145078 | 0.95588463 |
| **Genre 10** | -0.17612252 | 0.26632295 | 0.58969151 | 0.37327423 | -0.27979377 |

| **Configuration module (./backend/configuration.py)** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| Storing an integer  (positive) | Valid |  | value1 → 2  value2 → 5678923423  value3 → 10  value4 → 567 | value1 → 2  value2 → 5678923423  value3 → 10  value4 → 567 | Returns the correct values, and as integers. | ✓ | - |
| Storing an integer  (negative) | Valid |  | value1 → -2  value2 → -5678923423  value3 → -10  value4 → -567 | value1 → -2  value2 → -5678923423  value3 → -10  value4 → -567 | Returns the correct values, and as integers. | ✓ | - |
| Storing an integer  (positive expression) | Valid |  | value1 → 100  value2 → 100000  value3 → 32300  value4 → 10000000000000000000000000000000000000000000000 | ConfigInvalidDataForType Error | Evaluates any mathematical expression if it is written using the Python mathematical operators, and returns it as an integer. | ⛌ | ✓ |
| value1 → 100  value2 → 100000  value3 → 32300  value4 → 10000000000000000000000000000000000000000000000 | ✓ |
| Storing an integer  (negative expression) | Valid |  | value1 → -100  value2 → -100000  value3 → -32300  value4 → -10000000000000000000000000000000000000000000000 | ConfigInvalidDataForType Error | Evaluates any mathematical expression if it is written using the Python mathematical operator, and returns it as an integer | ⛌ | ✓ |
| value1 → -100  value2 → -100000  value3 → -32300  value4 → -10000000000000000000000000000000000000000000000 | ✓ |
| Storing an integer  (extreme positive) | Boundary |  | 1 \* 10302 | inf | Large integers in Python, around these values are taken as infinity. This means these fail, but as they are not required for the project, they are not an issue. | ⛌ | ⛌ |
| Storing an integer  (extreme negative) | Boundary |  | -1 \* 10302 | inf | ⛌ | ⛌ |
| Storing an integer  (string, boolean, decimals) | Erroneous |  | ConfigInvalidDataForType Error | ConfigInvalidDataForType Error | Any string value, boolean value, or floating point number, either positive or negative is ignored. | ✓ | - |
|
|
| Storing an string  (strings, string with non-alphabetical characters, booleans, integers, and decimals) | Valid |  | value1 → “string”  value2 → “string 2 with spaces & special characters!”  value3 → “450”  value4 → “-450”  value5 → “true”  value6 → “false”  value7 → ”6.901”  value8 → “-6.901” | value1 → “string”  value2 → “string 2 with spaces & special characters!”  value3 → “450”  value4 → “-450”  value5 → “true”  value6 → “false”  value7 → ”6.901”  value8 → “-6.901” | Returns the correct values, regardless of the form of the value of the field. Floats, integers and booleans are all treated as strings | ✓ | - |
|
|
|
| Storing an floating point  (positive and negative) | Valid |  | value1 → 1.02  value2 → 6002.30523  value3 → 10 value4 → -1.02  value5 → -6002.30523  value6 → -10 | value1 → 1.02  value2 → 6002.30523  value3 → 10.0 value4 → -1.02  value5 → -6002.30523  value6 → -10.0 | Returns the correct corresponding value as a float. Integers are also cast to floats as well. | ✓ | - |
|
| Storing an floating point  (positive expression) | Valid |  | value1 → 100.01  value2 → 1e-05  value3 → 0.000323  value4 → 1e-46 | ConfigInvalidDataForType Error | Returns the correct result from any expression using python operations as a value, and gives the result as a float | ⛌ | ✓ |
| value1 → 100.01  value2 → 1e-05  value3 → 0.000323  value4 → 1e-46 | ✓ |
| Storing an floating point  (negative expression) | Valid |  | value1 → -100.01  value2 → -1e-05  value3 → -0.000323  value4 → -1e-46 | ConfigInvalidDataForType Error | Returns the correct result from any expression using python operations as a value, and gives the result as a float | ⛌ | ✓ |
| value1 → -100.01  value2 → -1e-05  value3 → -0.000323  value4 → -1e-46 | ✓ |
| Storing an floating point  (integers) | Valid |  | value1 → 10032  value2 → 123  value3 → -10032  value4 → -123 | value1 → 10032.0  value2 → 123.0  value3 → -10032.0  value4 → -123.0 | Returns the correct result, to the corresponding key, as an integer. | ✓ | - |
| Storing an floating point  (extreme positive) | Boundary |  | 1.8 \* 10308 | inf | These values are very large, with Python storing them as infinity. This isn’t an issue for the project so can be ignored. | ⛌ | ⛌ |
| Storing an floating point  (extreme negative) | Boundary |  | -1.8 \* 10308 | inf | ⛌ | ⛌ |
| Storing an floating point  (strings and booleans) | Erroneous |  | ConfigInvalidDataForType Error | ConfigInvalidDataForType Error | Booleans and strings cannot be assigned as floats, which is correct | ✓ | - |
| Storing a binary string  (strings and booleans) | Valid |  | Corresponding value should be in binary  value1 → string  value2 → string 2 & special characters!  value3 → true  value4 → false | Corresponding value are in binary  value1 → string  value2 → string 2 & special characters!  value3 → true  value4 → false | Returns the value in binary to the corresponding key.  Any value that is given that is a string format (including booleans) are converted to binary and returned as binary | ✓ | - |
| Storing a binary string  (integers) | Valid |  | Corresponding value should be in binary  value1 → 1232  value2 → -1232 | Corresponding value are in binary  value1 → 1232  value2 → -1232 | Returns the value in binary to the corresponding key.  Integers, regardless of being positive or negative, are converted to binary | ✓ | - |
| Storing a binary string  (decimals) | Valid |  | Corresponding value should be in binary  value1 → 123.563  value2 → -123.563 | Corresponding value are in binary  value1 → 123.563  value2 → -123.563 | Returns the value in binary to the corresponding key.  Decimal numbers are converted to binary regardless of sign. | ✓ | - |
| Storing a boolean  (true and false) | Valid |  | value1 → true  value2 → false | value1 → True  value2 → False | Returns the corresponding boolean value for each parameter, as a boolean data type. | ✓ | - |
|
| Storing a boolean  (string) | Erroneous |  | ConfigInvalidDataForType Error | ConfigInvalidDataForType Error | Raises exception as boolean can only be given as | ✓ |  |
| Storing a boolean  (integer) | Erroneous |  | ConfigInvalidDataForType Error | ConfigInvalidDataForType Error | Raises exception as they are not valid values for the datatypes | ✓ | - |
| Storing a boolean  (decimal) | Erroneous |  | ConfigInvalidDataForType Error | ConfigInvalidDataForType Error | Raises exception as they are not valid values for the datatypes | ✓ | - |
| Storing an invalid data type | Erroneous |  | ConfigInvalidDataTypeError Error | ConfigInvalidDataTypeError Error | Raises exception as its an invalid datatype | ✓ | - |
| Hierarchy group  (headings and miscellaneous records) | Valid |  | header1 value1 → false  header1 value → true  header2 value1 → “string”  header2 value2 → 1.2  misc1 →”hello world” | header1 value1 → false  header1 value → true  header2 value1 → “string”  header2 value2 → 1.2  misc1 →”hello world” | Returns the corresponding value with the correct data type, with the correct lookup string. | ✓ | - |
| Hierarchy group  (heading then miscellaneous record then heading) | Valid |  | header1 value1 → false  header1 value → true  header2 value1 → “string”  header2 value2 → 1.2  misc1 →”hello world” | header1 value1 → false  header1 value → true  header2 value1 → “string”  header2 value2 → 1.2  misc1 →”hello world” | Returns the corresponding value with the correct data type, with the correct lookup string. | ✓ | - |
| Given default configuration  (known file) | Valid |  | header1 value1 → 4  header1 value2 → 3.5  header2 value1 → hello header2 value2 → true  misc1 → false | header1 value1 → 4  header1 value2 → 3.5  header2 value1 → hello header2 value2 → true  misc1 → false | Returns the corresponding value for each of the values, under the correct heading. The values are of the correct data type | ✓ | - |
| Given default configuration  (unknown file) | Erroneous | No file exists | FileNotFoundError | FileNotFoundError | Throws the correct error | ✓ | - |
| Given default configuration  (overridden values) | Valid |  | header1 value1 → 4  header1 value2 → -4203.01  header2 value1 → hello header2 value2 → true  misc1 → true | header1 value1 → 4  header1 value2 → -4203.01  header2 value1 → hello header2 value2 → true  misc1 → true | Values from the default do not change, and the values given in the second file are overridden.  The datatypes of each value is corresponds to the correct item | ✓ | - |
| Given default configuration  (overridden values) | Valid |  | header1 value1 → 4  header1 value2 → 3.5  header1 value3 → string  header2 value1 → hello header2 value2 → true  header2 value3 → -102  misc1 → false  misc2 → binary | header1 value1 → 4  header1 value2 → 3.5  header1 value3 → string  header2 value1 → hello header2 value2 → true  header2 value3 → -102  misc1 → false  misc2 → binary | Values from the default do not change, and the values given in the second file are added to the available options..  The datatypes of each value is corresponds to the correct item | ✓ | - |
| Configuration not found | Erroneous | No file exists | FileNotFoundError | FileNotFoundError | Throws the correct error | ✓ | - |

| **Environ variable manipulation module (./backend/environ\_manipulation.py)** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| Get target handler from URL | Valid | {"REQUEST\_URI": "/cgi-bin/handler\_name/target"} | handler\_name | handler\_name | Gets the handler name from the request URL | ✓ | - |
| Valid | {"REQUEST\_URI": "/cgi-bin/handler\_name"} | handler\_name | handler\_name | Gets the handler name from the request URL | ✓ | - |
| Erroneous | {"REQUEST\_URI": "/cgi-bin/} | None | None | Returns None (Python equivalent of null) as there is not a | ✓ | - |
| Get target function from URL | Valid | {"REQUEST\_URI": "/cgi-bin/handler/function\_name"} | function\_name | function\_name | Returns the function name | ✓ | - |
| Erroneous | {"REQUEST\_URI": "/cgi-bin/handler/"} | None | None | Returns None, as the function name is not given | ✓ | - |
| Erroneous | {"REQUEST\_URI": "/cgi-bin/"} | None | None | Returns None, as the function name is not given | ✓ | - |

| **Data Structures module (./backend/data\_structures.py)** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| Test order (first in first out)  (Queue) | Valid | 5, 9, 1, 2, 3, 4 | 5, 9, 1, 2, 3, 4 | 5, 9, 1, 2, 3, 4 | Follows the first in first out structure correctly | ✓ | - |
| Test overflow  (Queue) | Erroneous | Queue of with max length of 5  5,9,1,2,3,4 | QueueOverflowError | QueueOverflowError | Throws the QueueOverflowError, as the queue exceeded its maximum length | ✓ | - |
| Test underflow  (Queue) | Erroneous | Pop from empty queue | QueueUnderflowError | QueueUnderflowError | QueueUnderflowError is thrown, as the queue is empty, so cannot be popped. | ✓ | - |
| Test order (first in first out)  (Priority Queue) | Valid | 5 (priority 1),  9 (priority 4),  1 (priority 2),  2 (priority 2),  3 (priority 3),  4 (priority 1) | 9, 3, 1, 2, 5, 4 | 9, 3, 1, 2, 5, 4 | Pops the items in the expected order. It gives the highest priority (4) first, then priority (1) last. | ✓ | - |
| Test order using a priority function (first in first out)  (Priority Queue) | Valid | [5, 1],  [9, 4],  [1, 2],  [2, 2],  [3, 3],  [4, 1]  sort\_func = lambda x: -x[1] | [5, 1], [4, 1], [1, 2], [2, 2], [3, 3], [9, 4] | [5, 1], [4, 1], [1, 2], [2, 2], [3, 3], [9, 4] | Pops the items in the expected order, giving the entire list stored for each entry, as expected. | ✓ | - |
| Test overflow  (Priority Queue) | Erroneous | Queue of with max length of 5  5 (priority 1),  9 (priority 4),  1 (priority 2),  2 (priority 2),  3 (priority 3),  4 (priority 1) | QueueOverflowError | QueueOverflowError | Throws the QueueOverflowError, as the queue exceeded its maximum length | ✓ | - |
| Test underflow  (Priority Queue) | Erroneous | Pop from empty queue | QueueUnderflowError | QueueUnderflowError | QueueUnderflowError is thrown, as the queue is empty, so cannot be popped. | ✓ | - |
| Test in order traversal  (Binary Tree) | Valid | 5,9,1,2,3,4 | 1,2,3,4,5,9 | 1,2,3,4,5,9 | Traverses the tree in the correct order. | ✓ | - |
| Test in order traversal with access function  (Binary Tree) | Valid | [5, 1],  [9, 4],  [1, 2],  [2, 2],  [3, 3],  [4, 1]  access\_function = lambda x: -x[1] | [[9, 4], [3, 3], [1, 2], [2, 2], [5, 1], [4, 1]] | [[9, 4], [3, 3], [1, 2], [2, 2], [5, 1], [4, 1]] | Traverses the tree in the correct order, using the access function to compare the values. | ✓ | - |
| Test order (first in last out)  (Stack) | Valid | 5,9,1,2,3,4 | 4,3,2,1,9,5 | 4,3,2,1,9,5 | Follows the First in Last out structure of a stack | ✓ | - |
| Test overflow  (Stack) | Erroneous | Stack of with max length of 5  5,9,1,2,3,4 | StackOverflowError | StackOverflowError | Throws the StackOverflowError as the stack is full, so cannot have more items pushed into it | ✓ | - |
| Test underflow  (Stack) | Erroneous | Pop from empty stack | StackUnderflowError | StackUnderflowError | Throws the StackUnderflowError as the stack does not have any items in, so cannot be popped from | ✓ | - |

| **Logging module (./backend/logger.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Session Creation | Valid | None | Horziontal dashed line  New session created: date time  Horziontal dashed line  Written to file | Horziontal dashed line  New session created: date time  Horziontal dashed line  Written to file | Correctly creates a header which states when the session is created | ✓ | - |
| 2 | Write message to file  (without line wrapping) | Valid | Test message | [date time] test message  Written to file | [date time] test message  Written to file | Correctly writes the message to the file, with the date and the time that the it was logged | ✓ | - |
| 3 | Write message to file  (with line wrapping) | Valid | This is a very long test message. It should be long enough to wrap around lines in the log file. This message should demonstrate this ability clearly. | Message written to file with line wrap | Message written to file with line wrap | Correctly adds the message to the file, with the date and time, and wrapping the line around the line limit.  It does not work perfectly, when the line ends before a space. This is not a big issue, so can be overlooked (This happens with this test) | ✓ | - |
| 4 | Write message to file while not in debugging mode | Valid | Test message | No change to file | No change to file | The log file should not be changed, with old contents not being deleted. | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |

| **Machine learning utilities module (./backend/ml\_utilities.py)** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| Jaccard similarity | Valid | 3, 4, 10, 5, 230, 67, 90  7, 8, 10, 3, 4, 5, 100230 | 0.4 | 0.4 | Correctly calculates the jaccard similarity between the two sets of values  Does not have invalid inputs, outside of inputting invalid sets, but as this is a module does not need to be handled. | ✓ | - |

### Components module

| **Accounts module (./backend/components/accounts.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Get user ID  (known username) | Valid | user1 | 1 | 1 | Using test database | ✓ | - |
| 2 | Get user ID  (unknown username) | Erroneous | adasdasdasd | IndexError | IndexError | Using test database  Not an intuitive error, but is only used in locations where the error cannot be thrown, | ✓ | - |
| 3 | Create a user account  (available username) | Valid | First name: user  Surname: 5  Username: user5  Password: random\_password! | New record added to users table  Want to Read, Currently Reading and Have Read reading lists created for the user | New record added to users table  Want to Read, Currently Reading and Have Read reading lists created for the user | Using test database  This includes the creation of the system-defined reading lists. | ✓ | - |
| 4 | Create a user account  (username taken) | Erroneous | First name: user  Surname: 4  Username: user4  Password: password | UserExistsError | UserExistsError | Using test database | ✓ | - |
| 5 | Hash a password | Valid | T3st Passw0rd | 4b9f7c53850304f5fe3fb3014d1a331dc2df1195a443b1d714e62e36ef870cf1 | 4b9f7c53850304f5fe3fb3014d1a331dc2df1195a443b1d714e62e36ef870cf1 | Hashes the password as expected, when using sha256 with 100,000 passes, and a salt of +%E!mKZ(5%Z}k#pi(cPW!US8TU-J87. This is the default values, except the salt which is not given | ✓ | - |
| password | 5d557544916fde5c6b162cfcbce84181fb2cbe8798439b643edf96ee4c5826b4 | 5d557544916fde5c6b162cfcbce84181fb2cbe8798439b643edf96ee4c5826b4 |
| 6 | Check a user’s credentials  (valid credentials) | Valid | User1: password | 1 | 1 | Using test database  Gives the correct corresponding user ID | ✓ | - |
| User2: password | 2 | 2 |
| User4: password | 4 | 4 |
| 7 | Check a user’s credentials  (invalid username) | Erroneous | Username: asdassdjasjdsfbdsf  Password: asdasdsd | InvalidUserCredentialsError | InvalidUserCredentialsError | Using test database | ✓ | - |
| 8 | Check a user’s credentials  (invalid password) | Erroneous | Username: user1  Password: adasda | InvalidUserCredentialsError | InvalidUserCredentialsError | Using test database  Correct password for the user would be password | ✓ | - |
| 9 | Create a session ID | Valid | User ID: 2 | Creates new session in table | Creates new session in table | Using test database  New ID is 698b62181707071862 | ✓ | - |
| 10 | Update session ID expiry time  (valid session ID) | Valid | Session ID: asdhjaksnce1263872613 | Date added column updates for the single record | Date added column updates for the single record | Using test database  Date added changes to 2024-02-04 18:40:41 | ✓ | - |
| 11 | Update session ID expiry time  (unkown session ID) | Erroneous | random\_session\_id | No change to the sessions table | No change to the sessions table | Using test database | ✓ | - |
| 12 | Get user ID from session ID  (Session ID valid) | Valid | asdhjaksnce1263872613 | 1 | 1 | Using test database | ✓ | - |
| adqweqiueqw0812309812 | 1 | 1 |
| poipoqwerwrw983453453 | 3 | 3 |
| 13 | Get user ID from session ID  (session ID expired) | Erroneous | Session ID: sdfasdvnjtit987652678 | SessionExpiredError  Session removed from table | SessionExpiredError  Session removed from table | Using test database | ✓ | - |
| 14 | Get user ID from session ID  (session ID not known) | Erroneous | Session ID: unknown\_session\_id | SessionExpiredError | SessionExpiredError | Using test database | ✓ | - |
| 15 | Close a session ID  (known session ID) | Valid | Session ID: swcdecwftrbr132788943 | Session Removed from table | Session Removed from table | Using test database | ✓ | - |
| 16 | Close a session ID  (unknown session ID) | Erroneous | unkown\_session | No change to sessions table | No change to sessions table | Using test database | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 3 |  |  |
| 9 |  |  |
| 10 |  |  |
| 11 |  |  |
| 13 |  |  |
| 15 |  |  |
| 16 |  |  |

| **Authors module (./backend/components/authors.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Follow an author  (valid user ID and author ID) | Valid | Author ID: 1  User ID: 4 | New record added to author\_followers table | New record added to author\_followers table | Using test database | ✓ | - |
| 2 | Follow an author  (invalid user ID) | Erroneous | Author ID: 500  User ID: 1 | No change to the database | No change to the database | Using test database | ✓ | - |
| 3 | Follow an author  (invalid author ID) | Erroneous | Author ID: 1  User ID: 500 | No change to the database | mysql.connector.errors.IntegrityError: | Using test database | ⛌ | ✓ |
| No change to the database | ✓ |
| 4 | Unfollow an author  (valid user ID and author ID) | Valid | Author ID: 1  User ID: 1 | Removal of relationship from author\_followers table | Removal of relationship from author\_followers table | Using test database | ✓ | - |
| 5 | Unfollow an author  (valid user ID and author ID, with user not following author) | Erroneous | Author ID: 3  User ID: 1 | No change to database | No change to database | Using test database | ✓ | - |
| 6 | Unfollow an author  (invalid user ID) | Erroneous | Author ID: 1  User ID: 100 | No change to database | No change to database | Using test database | ✓ | - |
| 7 | Unfollow an author  (invalid author ID) | Erroneous | Author ID: 100  User ID: 1 | No change to database | No change to database | Using test database | ✓ | - |
| 8 | Get an authors number of followers | Valid | 1 | 2 | 2 | This can only be valid, as this is only used when the author ID is explicitly given. | ✓ | - |
| 3 | 0 | 0 | ✓ | - |
| 9 | Get authors about data  (known author ID) | Valid | 1 | {"name": "Author 1", "about": "<p>This is the first author's about.</p>", "author\_id": 1, "num\_followers": 2, "books": [{"id": 1, "title": "Book 1", "cover": ""}, {"id": 5, "title": "Book 5", "cover": ""}], "average\_rating": 2.67, "num\_ratings": 3, "genres": ['Genre 8', 'Genre 1', 'Genre 9', 'Genre 4', 'Genre 2', 'Genre 7']} | {"name": "Author 1", "about": "<p>This is the first author's about.</p>", "author\_id": 1, "num\_followers": 2, "books": [{"id": 1, "title": "Book 1", "cover": ""}, {"id": 5, "title": "Book 5", "cover": ""}], "average\_rating": 2.67, "num\_ratings": 3, "genres": ['Genre 8', 'Genre 1', 'Genre 9', 'Genre 4', 'Genre 2', 'Genre 7']} | Using test database  Gets all of the data from the database in the correct format. | ✓ | - |
| 3 | {"name": "Author 3", "about": "<p>This is the third author's about.</p>", "author\_id": 3, "num\_followers": 0, "books": [{"id": 3, "title": "Book 3", "cover": ""}], "average\_rating": 3.0, "num\_ratings": 3, "genres": ['Genre 9', 'Genre 6', 'Genre 8', 'Genre 5', 'Genre 10', 'Genre 3', 'Genre 4', 'Genre 2']} | {"name": "Author 3", "about": "<p>This is the third author's about.</p>", "author\_id": 3, "num\_followers": 0, "books": [{"id": 3, "title": "Book 3", "cover": ""}], "average\_rating": 3.0, "num\_ratings": 3, "genres": ['Genre 9', 'Genre 6', 'Genre 8', 'Genre 5', 'Genre 10', 'Genre 3', 'Genre 4', 'Genre 2']} | ✓ | - |
| 10 | Get authors about data  (unknown author ID) | Erroneous | 23 | AuthorNotFoundError | AuthorNotFoundError | Using test database | ✓ | - |
| 11 | Get list of author IDs | Valid | No parameters | [1, 2, 3] | [1, 2, 3] | Using test database | ✓ | - |
| 12 | Get list of author names | Valid | No parameters | [  {"id": 1, "name": "Author 1"},  {"id": 2, "name": "Author 2"},  {"id": 3, "name": "Author 3"}  ] | [  {"id": 1, "name": "Author 1"},  {"id": 2, "name": "Author 2"},  {"id": 3, "name": "Author 3"}  ] | Using test database | ✓ | - |
| 13 | Get books from the authors a user follows  (valid user ID) | Valid | 1 | {  0: {"author": "Author 2", "title": "Book 2", "book\_id": 2, "cover": ""},  1: {"author": "Author 1", "title": "Book 1", "book\_id": 1, "cover": ""},  2: {"author": "Author 2", "title": "Book 4", "book\_id": 4, "cover": ""},  3: {"author": "Author 1", "title": "Book 5", "book\_id": 5, "cover": ""}  } | {  0: {"author": "Author 2", "title": "Book 2", "book\_id": 2, "cover": ""},  1: {"author": "Author 1", "title": "Book 1", "book\_id": 1, "cover": ""},  2: {"author": "Author 2", "title": "Book 4", "book\_id": 4, "cover": ""},  3: {"author": "Author 1", "title": "Book 5", "book\_id": 5, "cover": ""}  } | Using test database | ✓ | - |
| 3 | {  0: {"author": "Author 2", "title": "Book 2", "book\_id": 2, "cover": ""},  1: {"author": "Author 2", "title": "Book 4", "book\_id": 4, "cover": ""}  } | {  0: {"author": "Author 2", "title": "Book 2", "book\_id": 2, "cover": ""},  1: {"author": "Author 2", "title": "Book 4", "book\_id": 4, "cover": ""}  } |
| 14 | Get books from the authors a user follows  (invalid user ID) | Erroneous | 40 | None | None | Using test database  This cannot be accessed from the client as the user ID is known. It does not cause errors if it does occur. | ✓ | - |
| 15 | Get books from the authors a user follows  (no authors followed) | Erroneous | 4 | None | None | Using test database  The user doesn’t follow any authors, so there are not any books given. | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |

| **Books module (./backend/components/books.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Get similar items  (known book) | Valid | 1 | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | Using test database  Number of similarities is reduced to work with the test database properly, from 8 to 2.  These tests failed as the similarity metric used to calculate the similarity did not give an accurate representation. It was switched to using dot product, which then gave a much better representation of the similarities. | ✓ | ✓ |
| 2 | [{'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}] | [{'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}] | ⛌ |
| 3 | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}] | ⛌ |
| 4 | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | [{'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}, {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}] | ⛌ |
| 5 | [{'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}] | [{'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}, {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}] | ⛌ |
| 2 | [{'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}] | [{'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}] | ✓ |
| 3 | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | ✓ |
| 4 | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | [{'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}] | ✓ |
| 5 | [{'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}] | [{'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}, {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}] | ✓ |
| 2 | Get similar items  (unknown book) | Erroneous | 500 | BookNotFoundError | KeyError | This doesn’t throw the logical error. However, it is caught before this function is called, so can never occur. This means it doesn’t need changing. | ⛌ | ⛌ |
| 3 | Get book summary data by book ID  (known book) | Valid | 1 | {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''} | {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''} | Using test database | ✓ |  |
| 2 | {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''} | {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''} | ✓ |
| 3 | {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''} | {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''} | ✓ |
| 4 | {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''} | {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''} | ✓ |
| 5 | {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''} | {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''} | ✓ |
| 4 | Get book summary data by book ID  (unknown book ID) | Erroneous | 500 | BookNotFoundError | BookNotFoundError | Using test database | ✓ | - |
| 5 | Get book summary data by book ISBN  (known book) | Valid | 0111111111 | {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''} | {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''} | Using test database | ✓ | - |
| 0222222222 | {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''} | {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''} | ✓ |
| 0333333333 | {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''} | {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''} | ✓ |
| 0444444444 | {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''} | {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''} | ✓ |
| 0555555555 | {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''} | {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''} | ✓ |
| 6 | Get book summary data by book ISBN  (unknown book ID) | Erroneous | 1111111111 | BookNotFoundError | BookNotFoundError | Using test database | ✓ | - |
| 7 | Get newest books from the database | Valid | INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (1, "temp 1", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (1, "temp 2", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (1, "temp 3", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (2, "temp 4", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (3, "temp 5", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (1, "temp 6", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (2, "temp 7", "", "", "", "", 1, "2022-2-2", "1234567890");  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES (3, "temp 8", "", "", "", "", 1, "2022-2-2", "1234567890"); | {  0: {'author': 'Author 3', 'title': 'temp 8', 'book\_id': 13, 'cover': ''},  1: {'author': 'Author 2', 'title': 'temp 7', 'book\_id': 12, 'cover': ''},  2: {'author': 'Author 1', 'title': 'temp 6', 'book\_id': 11, 'cover': ''},  3: {'author': 'Author 3', 'title': 'temp 5', 'book\_id': 10, 'cover': ''},  4: {'author': 'Author 2', 'title': 'temp 4', 'book\_id': 9, 'cover': ''},  5: {'author': 'Author 1', 'title': 'temp 3', 'book\_id': 8, 'cover': ''},  6: {'author': 'Author 1', 'title': 'temp 2', 'book\_id': 7, 'cover': ''},  7: {'author': 'Author 1', 'title': 'temp 1', 'book\_id': 6, 'cover': ''}  } | {  0: {'author': 'Author 3', 'title': 'temp 8', 'book\_id': 13, 'cover': ''},  1: {'author': 'Author 2', 'title': 'temp 7', 'book\_id': 12, 'cover': ''},  2: {'author': 'Author 1', 'title': 'temp 6', 'book\_id': 11, 'cover': ''},  3: {'author': 'Author 3', 'title': 'temp 5', 'book\_id': 10, 'cover': ''},  4: {'author': 'Author 2', 'title': 'temp 4', 'book\_id': 9, 'cover': ''},  5: {'author': 'Author 1', 'title': 'temp 3', 'book\_id': 8, 'cover': ''},  6: {'author': 'Author 1', 'title': 'temp 2', 'book\_id': 7, 'cover': ''},  7: {'author': 'Author 1', 'title': 'temp 1', 'book\_id': 6, 'cover': ''}  } | Insert new books into the test database, with a delay between, then check the output is in the correct order. | ✓ | - |
| 8 | Get a book’s about data  (known book) | Valid | Book ID: 1  User ID: 1 | {'title': 'Book 1', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0111111111', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 2, 'genres': ['Genre 8', 'Genre 9', 'Genre 4', 'Genre 7', 'Genre 5', 'Genre 2', 'Genre 1', 'Genre 10'], 'author\_id': 1, 'average\_rating': 3.5, 'num\_ratings': 2, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 1, 'num\_1\_star': 0, 'current\_user\_review': {'review\_id': 1, 'overall\_rating': 5, 'plot\_rating': 5, 'character\_rating': 5, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 2, 'overall\_rating': 2, 'plot\_rating': 3, 'character\_rating': 1, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}]} | {'title': 'Book 1', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0111111111', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 2, 'genres': ['Genre 8', 'Genre 9', 'Genre 4', 'Genre 7', 'Genre 5', 'Genre 2', 'Genre 1', 'Genre 10'], 'author\_id': 1, 'average\_rating': 3.5, 'num\_ratings': 2, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 1, 'num\_1\_star': 0, 'current\_user\_review': {'review\_id': 1, 'overall\_rating': 5, 'plot\_rating': 5, 'character\_rating': 5, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 2, 'overall\_rating': 2, 'plot\_rating': 3, 'character\_rating': 1, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}]} | Using test database  Tested with a known user ID, so will give the reviews if available. | ✓ | - |
| Book ID: 2  User ID: 1 | {'title': 'Book 2', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0222222222', 'author': 'Author 2', 'author\_about': "<p>This is the second author's about.</p>", 'author\_number\_followers': 3, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 3, 'genres': ['Genre 2', 'Genre 3', 'Genre 8', 'Genre 10', 'Genre 5', 'Genre 4', 'Genre 6', 'Genre 7'], 'author\_id': 2, 'average\_rating': 4.0, 'num\_ratings': 3, 'num\_5\_stars': 1, 'num\_4\_stars': 1, 'num\_3\_stars': 1, 'num\_2\_stars': 0, 'num\_1\_star': 0, 'current\_user\_review': {'review\_id': 3, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 3, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 5, 'overall\_rating': 4, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 4, 'overall\_rating': 5, 'plot\_rating': 2, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} | {'title': 'Book 2', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0222222222', 'author': 'Author 2', 'author\_about': "<p>This is the second author's about.</p>", 'author\_number\_followers': 3, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 3, 'genres': ['Genre 2', 'Genre 3', 'Genre 8', 'Genre 10', 'Genre 5', 'Genre 4', 'Genre 6', 'Genre 7'], 'author\_id': 2, 'average\_rating': 4.0, 'num\_ratings': 3, 'num\_5\_stars': 1, 'num\_4\_stars': 1, 'num\_3\_stars': 1, 'num\_2\_stars': 0, 'num\_1\_star': 0, 'current\_user\_review': {'review\_id': 3, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 3, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 5, 'overall\_rating': 4, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 4, 'overall\_rating': 5, 'plot\_rating': 2, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} | ✓ | - |
| Book ID: 3  User ID: 1 | {'title': 'Book 3', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0333333333', 'author': 'Author 3', 'author\_about': "<p>This is the third author's about.</p>", 'author\_number\_followers': 0, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 3, 'genres': ['Genre 4', 'Genre 3', 'Genre 2', 'Genre 6', 'Genre 10', 'Genre 9', 'Genre 8', 'Genre 5'], 'author\_id': 3, 'average\_rating': 3.0, 'num\_ratings': 3, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 1, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': None, 'author\_following': False, 'reviews': [{'id': 8, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 7, 'overall\_rating': 1, 'plot\_rating': 1, 'character\_rating': 2, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 6, 'overall\_rating': 5, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} | {'title': 'Book 3', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0333333333', 'author': 'Author 3', 'author\_about': "<p>This is the third author's about.</p>", 'author\_number\_followers': 0, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 3, 'genres': ['Genre 4', 'Genre 3', 'Genre 2', 'Genre 6', 'Genre 10', 'Genre 9', 'Genre 8', 'Genre 5'], 'author\_id': 3, 'average\_rating': 3.0, 'num\_ratings': 3, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 1, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': None, 'author\_following': False, 'reviews': [{'id': 8, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 7, 'overall\_rating': 1, 'plot\_rating': 1, 'character\_rating': 2, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 6, 'overall\_rating': 5, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} |  |  |
| Book ID: 4  User ID: 1 | {'title': 'Book 4', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0444444444', 'author': 'Author 2', 'author\_about': "<p>This is the second author's about.</p>", 'author\_number\_followers': 3, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 4, 'genres': ['Genre 2', 'Genre 1', 'Genre 5', 'Genre 10', 'Genre 3', 'Genre 9', 'Genre 6', 'Genre 7'], 'author\_id': 2, 'average\_rating': 2.75, 'num\_ratings': 4, 'num\_5\_stars': 0, 'num\_4\_stars': 1, 'num\_3\_stars': 2, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': {'review\_id': 9, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 5, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 12, 'overall\_rating': 4, 'plot\_rating': 3, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 11, 'overall\_rating': 1, 'plot\_rating': 2, 'character\_rating': 3, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 10, 'overall\_rating': 3, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} | {'title': 'Book 4', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0444444444', 'author': 'Author 2', 'author\_about': "<p>This is the second author's about.</p>", 'author\_number\_followers': 3, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 4, 'genres': ['Genre 2', 'Genre 1', 'Genre 5', 'Genre 10', 'Genre 3', 'Genre 9', 'Genre 6', 'Genre 7'], 'author\_id': 2, 'average\_rating': 2.75, 'num\_ratings': 4, 'num\_5\_stars': 0, 'num\_4\_stars': 1, 'num\_3\_stars': 2, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': {'review\_id': 9, 'overall\_rating': 3, 'plot\_rating': 2, 'character\_rating': 5, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': [{'id': 12, 'overall\_rating': 4, 'plot\_rating': 3, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user4'}, {'id': 11, 'overall\_rating': 1, 'plot\_rating': 2, 'character\_rating': 3, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 10, 'overall\_rating': 3, 'plot\_rating': 3, 'character\_rating': 4, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user2'}]} |  |  |
| Book ID: 5  User ID: 1 | {'title': 'Book 5', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0555555555', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 1, 'genres': ['Genre 4', 'Genre 9', 'Genre 1', 'Genre 2', 'Genre 6', 'Genre 5', 'Genre 3', 'Genre 10'], 'author\_id': 1, 'average\_rating': 1.0, 'num\_ratings': 1, 'num\_5\_stars': 0, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': {'review\_id': 13, 'overall\_rating': 1, 'plot\_rating': 2, 'character\_rating': 1, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': []} | {'title': 'Book 5', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0555555555', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 1, 'genres': ['Genre 4', 'Genre 9', 'Genre 1', 'Genre 2', 'Genre 6', 'Genre 5', 'Genre 3', 'Genre 10'], 'author\_id': 1, 'average\_rating': 1.0, 'num\_ratings': 1, 'num\_5\_stars': 0, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 0, 'num\_1\_star': 1, 'current\_user\_review': {'review\_id': 13, 'overall\_rating': 1, 'plot\_rating': 2, 'character\_rating': 1, 'summary': None, 'rating\_body': None}, 'author\_following': True, 'reviews': []} | ✓ | - |
| 9 | Get a book’s about data  (no user ID with known book) | Valid | Book ID: 1  User ID: None | {'title': 'Book 1', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0111111111', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 2, 'genres': ['Genre 8', 'Genre 9', 'Genre 4', 'Genre 7', 'Genre 5', 'Genre 2', 'Genre 1', 'Genre 10'], 'author\_id': 1, 'average\_rating': 3.5, 'num\_ratings': 2, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 1, 'num\_1\_star': 0, 'current\_user\_review': None, 'author\_following': False, 'reviews': [{'id': 2, 'overall\_rating': 2, 'plot\_rating': 3, 'character\_rating': 1, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 1, 'overall\_rating': 5, 'plot\_rating': 5, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user1'}]} | {'title': 'Book 1', 'cover\_image': '', 'synopsis': '<p>This book does not have a synopsis</p>', 'purchase\_link': '', 'release\_date': '02/02/2022', 'isbn': '0111111111', 'author': 'Author 1', 'author\_about': "<p>This is the first author's about.</p>", 'author\_number\_followers': 2, 'num\_want\_read': 0, 'num\_reading': 0, 'num\_read': 2, 'genres': ['Genre 8', 'Genre 9', 'Genre 4', 'Genre 7', 'Genre 5', 'Genre 2', 'Genre 1', 'Genre 10'], 'author\_id': 1, 'average\_rating': 3.5, 'num\_ratings': 2, 'num\_5\_stars': 1, 'num\_4\_stars': 0, 'num\_3\_stars': 0, 'num\_2\_stars': 1, 'num\_1\_star': 0, 'current\_user\_review': None, 'author\_following': False, 'reviews': [{'id': 2, 'overall\_rating': 2, 'plot\_rating': 3, 'character\_rating': 1, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user3'}, {'id': 1, 'overall\_rating': 5, 'plot\_rating': 5, 'character\_rating': 5, 'summary': None, 'rating\_body': None, 'date\_added': '12/02/2024', 'username': 'user1'}]} | Using test database  As this works for this book, it will work for any of the others, as proven by previous test. | ✓ | - |
| 10 | Get a book’s about data  (unknown book) | Erroneous | Book ID: 500  User ID: None | BookNotFoundError | BookNotFoundError | Using test database  User ID does not matter. This may succeed outside of the test data, as the | ✓ | - |
| 11 | Leave a review  (all fields given) | Valid | User ID: 2  Book ID: 1  Overall rating: 5  Plot rating: 3  Character rating: 2  Summary: This is the review summary  Body: This is the main review body | Review added to the database | Review added to the database | Using test database | ✓ | - |
| 12 | Leave a review  (plot rating, character rating, summary and thoughts excluded) | Valid | User ID: 4  Book ID: 1  Overall rating: 4  Plot rating: None  Character rating: None  Summary: None  Body: None | Review added to the database | Review added to the database | Using test database | ✓ | - |
| 13 | Leave a review  (plot rating, summary and thoughts excluded) | Valid | User ID: 3  Book ID: 2  Overall rating: 3  Plot rating: None  Character rating: 2  Summary: None  Body: None | Review added to the database | Review added to the database | Using test database | ✓ | - |
| 14 | Leave a review  (character rating, summary and thoughts excluded) | Valid | User ID: 1  Book ID: 3  Overall rating: 3  Plot rating: 3  Character rating: None  Summary: None  Body: None | Review added to the database | Review added to the database | Using test database | ✓ | - |
| 15 | Leave a review  (summary and thoughts excluded) | Valid | User ID: 2  Book ID: 5  Overall rating: 3  Plot rating: 3  Character rating: 2  Summary: None  Body: None | Review added to the database | Review added to the database | Using test database | ✓ | - |
| 16 | Leave a review  (with existing review from the user for the book) | Valid | User ID: 1  Book ID: 1  Overall rating: 3  Plot rating: 3  Character rating: 2  Summary: summary  Body: main | Old review removed from the database  Review added to the database | Old review removed from the database  Review added to the database | Using test database | ✓ | - |
| 17 | Delete a review  (valid user ID and review ID) | Valid | Added temporary review  Review ID: 14  User ID: 2 | No change to the database | No change to the database | Using test database | ✓ | - |
| 18 | Delete a review  (valid user ID and invalid review ID) | Erroneous | Added temporary review  Review ID: 50  User ID: 2 | No change to the database | No change to the database | Using test database | ✓ | - |
| 19 | Delete a review  (invalid user ID and valid review ID) | Erroneous | Added temporary review  Review ID: 14  User ID: 5895 | No change to the database | No change to the database | Using test database | ✓ | - |
| 20 | Get highest rated books | Valid | No parameters | {0: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}, 1: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}, 2: {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}, 3: {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, 4: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}} | {0: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}, 1: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}, 2: {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': ''}, 3: {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': ''}, 4: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}} | Using test database  Gives the expected order of ratings from the test database. | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |
| 15 |  |  |
| 16 |  |  |
| 17 |  |  |
| 18 |  |  |
| 19 |  |  |

| **Diaries module (./backend/components/diaries.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Add an diary entry  (all fields given) | Valid | User ID: 3  Book ID: 3  Overall rating: 4  Character rating: 2  Plot rating: 4  Summary: Diary entry summary  Thoughts: Diary entry thoughts  Pages read: 52 | Entry added to the database | Entry added to the database | Using test database | ✓ | - |
| 2 | Add an diary entry  (plot rating, character rating, summary and thoughts excluded) | Valid | User ID: 4  Book ID: 5  Overall rating: 5  Character rating: None  Plot rating: None  Summary: None  Thoughts: None  Pages read: 1 | Entry added to the database | Entry added to the database | Using test database | ✓ | - |
| 3 | Add an diary entry  (plot rating, summary and thoughts excluded) | Valid | User ID: 4  Book ID: 1  Overall rating: 3  Character rating: 2  Plot rating: None  Summary: None  Thoughts: None  Pages read: 3 | Entry added to the database | Entry added to the database | Using test database | ✓ | - |
| 4 | Add an diary entry  (character rating, summary and thoughts excluded) | Valid | User ID: 1  Book ID: 2  Overall rating: 5  Character rating: None  Plot rating: 5  Summary: None  Thoughts: None  Pages read: 28 | Entry added to the database | Entry added to the database | Using test database | ✓ | - |
| 5 | Add an diary entry  (summary and thoughts excluded) | Valid | User ID: 2  Book ID: 2  Overall rating: 5  Character rating: 3  Plot rating: 5  Summary: None  Thoughts: None  Pages read: 19 | Entry added to the database | Entry added to the database | Using test database | ✓ | - |
| 6 | Delete a diary entry  (valid user ID and entry ID) | Valid | User ID: 1  Entry ID: 1 | Entry removed from the database |  | Using test database | ✓ | - |
| 7 | Delete a diary entry  (invalid user ID and valid entry ID) | Erroneous | User ID: 5  Entry ID: 1 | No change to the database |  | Using test database | ✓ | - |
| 8 | Delete a diary entry  (valid user ID and invalid entry ID) | Erroneous | User ID: 1  Entry ID: 10 | No change to the database |  | Using test database | ✓ | - |
| 9 | Get diary entries  (known user ID) | Valid | User ID: 1 | {0: {'entry\_id': 1, 'book\_id': 1, 'overall\_rating': 5, 'character\_rating': 3, 'plot\_rating': 2, 'summary': 'Summary', 'thoughts': '<p>Thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 10, 'cover\_image': '', 'title': 'Book 1', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 3.5, 'number\_ratings': 2}, 1: {'entry\_id': 2, 'book\_id': 1, 'overall\_rating': 1, 'character\_rating': 2, 'plot\_rating': 5, 'summary': 'Entry summary', 'thoughts': '<p>Entry thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 21, 'cover\_image': '', 'title': 'Book 1', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 3.5, 'number\_ratings': 2}, 2: {'entry\_id': 3, 'book\_id': 2, 'overall\_rating': 5, 'character\_rating': 3, 'plot\_rating': 2, 'summary': 'A summary', 'thoughts': '<p>Thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 11, 'cover\_image': '', 'title': 'Book 2', 'author\_id': 2, 'author\_name': 'Author 2', 'average\_rating': 4.0, 'number\_ratings': 3}} | {0: {'entry\_id': 1, 'book\_id': 1, 'overall\_rating': 5, 'character\_rating': 3, 'plot\_rating': 2, 'summary': 'Summary', 'thoughts': '<p>Thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 10, 'cover\_image': '', 'title': 'Book 1', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 3.5, 'number\_ratings': 2}, 1: {'entry\_id': 2, 'book\_id': 1, 'overall\_rating': 1, 'character\_rating': 2, 'plot\_rating': 5, 'summary': 'Entry summary', 'thoughts': '<p>Entry thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 21, 'cover\_image': '', 'title': 'Book 1', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 3.5, 'number\_ratings': 2}, 2: {'entry\_id': 3, 'book\_id': 2, 'overall\_rating': 5, 'character\_rating': 3, 'plot\_rating': 2, 'summary': 'A summary', 'thoughts': '<p>Thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 11, 'cover\_image': '', 'title': 'Book 2', 'author\_id': 2, 'author\_name': 'Author 2', 'average\_rating': 4.0, 'number\_ratings': 3}} | Using test database | ✓ | - |
| User ID: 2 | {0: {'entry\_id': 4, 'book\_id': 5, 'overall\_rating': 2, 'character\_rating': 4, 'plot\_rating': 1, 'summary': 'Entry summary', 'thoughts': '<p>Entry thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 2, 'cover\_image': '', 'title': 'Book 5', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 1.0, 'number\_ratings': 1}, 1: {'entry\_id': 5, 'book\_id': 4, 'overall\_rating': 4, 'character\_rating': 5, 'plot\_rating': 3, 'summary': 'Short entry summary', 'thoughts': '<p>Long entry thoughts.</p>', 'date\_added': '12-02-2024', 'pages\_read': 5, 'cover\_image': '', 'title': 'Book 4', 'author\_id': 2, 'author\_name': 'Author 2', 'average\_rating': 2.75, 'number\_ratings': 4}} | {0: {'entry\_id': 4, 'book\_id': 5, 'overall\_rating': 2, 'character\_rating': 4, 'plot\_rating': 1, 'summary': 'Entry summary', 'thoughts': '<p>Entry thoughts</p>', 'date\_added': '12-02-2024', 'pages\_read': 2, 'cover\_image': '', 'title': 'Book 5', 'author\_id': 1, 'author\_name': 'Author 1', 'average\_rating': 1.0, 'number\_ratings': 1}, 1: {'entry\_id': 5, 'book\_id': 4, 'overall\_rating': 4, 'character\_rating': 5, 'plot\_rating': 3, 'summary': 'Short entry summary', 'thoughts': '<p>Long entry thoughts.</p>', 'date\_added': '12-02-2024', 'pages\_read': 5, 'cover\_image': '', 'title': 'Book 4', 'author\_id': 2, 'author\_name': 'Author 2', 'average\_rating': 2.75, 'number\_ratings': 4}} | ✓ |
| 10 | Get diary entries  (No entries given) | Boundary | User ID: 3 | {} | {} | Using test database | ✓ | - |
| User ID: 4 | {} | {} | ✓ |
| 11 | Get diary entries  (Unknown user) | Erroneous | User ID: 125 | {} | {} | Using test database | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |

| **Genres module (./backend/components/genres.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Get about data  (known genre) | Valid | Genre 1 | {'name': 'Genre 1', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | {'name': 'Genre 1', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | Using test database | ✓ | - |
| Genre 2 | 'name': 'Genre 2', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | 'name': 'Genre 2', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | ✓ |
| Genre 3 | {'name': 'Genre 3', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 3, 'title': 'Book 3', 'author': 'Author 3', 'cover': ''}}} | {'name': 'Genre 3', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 3, 'title': 'Book 3', 'author': 'Author 3', 'cover': ''}}} | ✓ |
| Genre 4 | {'name': 'Genre 4', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 3, 'title': 'Book 3', 'author': 'Author 3', 'cover': ''}}} | {'name': 'Genre 4', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 3, 'title': 'Book 3', 'author': 'Author 3', 'cover': ''}}} | ✓ |
| Genre 5 | {'name': 'Genre 5', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | {'name': 'Genre 5', 'about': '<p>This genre does not have an about</p>', 'books': {0: {'id': 4, 'title': 'Book 4', 'author': 'Author 2', 'cover': ''}}} | ✓ |
| 2 | Get about data  (unknown genre) | Erroneous | Genre 100 | GenreNotFoundError | GenreNotFoundError | Using test database | ✓ | - |
| 3 | Get genre name from ID  (known genre) | Valid | 1 | Genre 1 | Genre 1 | Using test database | ✓ | - |
| 2 | Genre 2 | Genre 2 | ✓ |
| 3 | Genre 3 | Genre 3 | ✓ |
| 4 | Genre 4 | Genre 4 | ✓ |
| 5 | Genre 5 | Genre 5 | ✓ |
| 4 | Get genre name from ID  (unknown genre) | Erroneous | 100 | GenreNotFoundError | IndexError | Using test database  This error is illogical, but it is not worth the extra complexity to catch, and handle logically. It is only used where the ID is explicitly known to be valid from previous queries, so this cannot be thrown. This means catching it and throwing a different one is unnecessary, so this is not amended. | ⛌ | ⛌ |

This testing should have additional items added into test database, as there are too few items to test it properly. It is included within an additional test file. These were not used for any other tests, as this number of items would make the other parts difficult to test, as well as changing some of the expected outputa from functions.

| INSERT INTO authors (first\_name, surname, about, clean\_name) VALUES  ('Kristin', 'Hannah', 'This author does not have an about', 'kristin hannah'),  ('B.A.', 'Paris', 'This author does not have an about', 'ba paris'),  ('George', 'Orwell', ' This author does not have an about', 'george orwell'),  ('Mary', 'E. Pearson', 'This author does not have an about', 'mary e pearson'),  ('Kurt', 'Vonnegut Jr.', 'This author does not have an about', 'kurt vonnegut jr'),  ('Rick', 'Riordan', 'This author does not have an about', 'rick riordan'),  ('Madeleine', "L'Engle", 'This author does not have an about', 'madeleine lengle'),  ('Alyson', 'Noel', 'This author does not have an about', 'alyson noel'),  ('Jay', 'Asher', 'This author does not have an about', 'jay asher'),  ('Marie', 'Kondo', 'This author does not have an about', 'marie kondo'),  ('William', 'Goldman', 'This author does not have an about', 'william goldman'),  ('Barbara', 'Kingsolver', 'This author does not have an about', 'barbara kingsolver'),  ('Susan', 'Ee', 'This author does not have an about', 'susan ee'),  ('Gregory', 'Maguire', 'This author does not have an about', 'gregory maguire'),  ('Bill', 'Bryson', 'This author does not have an about', 'bill bryson'),  ('Jennifer', 'L. Armentrout', 'This author does not have an about', 'jennifer l armentrout'),  ('Kristin', 'Cashore', 'This author does not have an about', 'kristin cashore'),  ('Aziz', 'Ansari', 'This author does not have an about', 'aziz ansari'),  ('Abbi', 'Glines', 'This author does not have an about', 'abbi glines');  INSERT INTO books (author\_id, title, clean\_title, synopsis, cover\_image, purchase\_link, fiction, release\_date, isbn) VALUES  (4, "The Nightingale", "the nightingale", "", "", "", 1, "2015-01-01", "41125521"),  (3, "Animal Farm", "animal farm", "", "", "", 1, "2003-01-01", "2207778"),  (7, "The Kiss of Deception (The Remnant Chronicles, #1)", "the kiss of deception the remnant chronicles 1", "", "", "", 1, "2014-01-01", "22617247"),  (9, "The Sea of Monsters (Percy Jackson and the Olympians, #2)", "the sea of monsters percy jackson and the olympians 2", "", "", "", 1, "2006-01-01", "43554"),  (9, "The Son of Neptune (The Heroes of Olympus, #2)", "the son of neptune the heroes of olympus 2", "", "", "", 1, "2011-01-01", "14406312"),  (9, "The Last Olympian (Percy Jackson and the Olympians, #5)", "the last olympian percy jackson and the olympians 5", "", "", "", 1, "2009-01-01", "4551489"),  (9, "The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)", "the sword of summer magnus chase and the gods of asgard 1", "", "", "", 1, "2015-01-01", "21400019"),  (9, "The Red Pyramid (Kane Chronicles, #1)", "the red pyramid kane chronicles 1", "", "", "", 1, "2010-01-01", "346572"),  (11, "Evermore (The Immortals, #1)", "evermore the immortals 1", "", "", "", 1, "1990-01-01", "4021549"),  (13, "The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing", "the lifechanging magic of tidying up the japanese art of decluttering and organizing", "", "", "", 1, "2014-01-01", "41711738"),  (15, "The Poisonwood Bible", "he poisonwood bible", "", "", "", 1, "2005-01-01", "810663"),  (16, "Angelfall (Penryn & the End of Days, #1)", "angelfall penryn the end of days 1", "", "", "", 1, "2013-01-01", "16435765"),  (18, "A Walk in the Woods", "a walk in the woods", "", "", "", 1, "1990-01-01", "613469"),  (19, "Onyx (Lux, #2)", "onyx lux 2", "", "", "", 1, "2012-01-01", "18211575"),  (19, "Opal (Lux, #3)", "opal lux 3", "", "", "", 1, "2012-01-01", "18591132"),  (19, "Origin (Lux, #4)", "origin lux 4", "", "", "", 1, "2013-01-01", "19259997"),  (20, "Graceling (Graceling Realm, #1)", "graceling graceling realm 1", "", "", "", 1, "2008-01-01", "3270810"),  (21, "Modern Romance", "modern romance", "", "", "", 1, "2015-01-01", "43014915"); |
| --- |

The following table gives the expected unique words for the test data (including the addition of the data above), and their corresponding IDF values. These values are what is expected in the unique\_words table in the database. These values are inserted with the information retrieval data part of the test data dump, so when testing these values, they are removed from the database beforehand. This is done within the function to generate the data, but as the function works, the output should be the same, so it would not be possible to tell if the data was wiped, or if the data was not changed

| jr | 1.74036 |  | glines | 1.74036 |  | end | 1.74036 |  | olympians | 1.43933 |  | 5 | 1.26324 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 0.895265 |  | aziz | 1.43933 |  | author | 0.78612 |  | marie | 1.43933 |  | book | 1.04139 |
| 3 | 1.04139 |  | decluttering | 1.74036 |  | e | 0.138303 |  | asher | 1.74036 |  | of | 0.962211 |
| nightingale | 1.74036 |  | lengle | 1.74036 |  | gregory | 1.74036 |  | alyson | 1.43933 |  | a | 0.117113 |
| realm | 1.74036 |  | mary | 1.43933 |  | penryn | 1.74036 |  | magic | 1.74036 |  | heroes | 1.74036 |
| paris | 1.74036 |  | maguire | 1.74036 |  | bill | 1.43933 |  | onyx | 1.74036 |  | poisonwood | 1.74036 |
| japanese | 1.74036 |  | genre | 0.740363 |  | lux | 1.26324 |  | magnus | 1.74036 |  | chase | 1.74036 |
| red | 1.74036 |  | william | 1.74036 |  | jay | 1.74036 |  | noel | 1.43933 |  | son | 0.740363 |
| l | 0.360151 |  | monsters | 1.74036 |  | remnant | 1.74036 |  | origin | 1.74036 |  | sword | 1.74036 |
| sea | 1.74036 |  | olympus | 1.74036 |  | last | 1.74036 |  | asgard | 1.74036 |  | bible | 1.74036 |
| chronicles | 1.43933 |  | organizing | 1.74036 |  | orwell | 1.74036 |  | hannah | 1.43933 |  | pearson | 1.43933 |
| 6 | 1.74036 |  | walk | 1.74036 |  | and | 1.1383 |  | 4 | 1.26324 |  | angelfall | 1.74036 |
| percy | 1.43933 |  | 7 | 1.74036 |  | deception | 1.74036 |  | Neptune | 1.74036 |  | vonnegut | 1.74036 |
| woods | 1.74036 |  | jackson | 1.43933 |  | ansari | 1.43933 |  | lifechanging | 1.74036 |  | riordan | 0.962211 |
| kingsolver | 1.43933 |  | opal | 1.74036 |  | olympian | 1.43933 |  | jennifer | 1.1383 |  | graceling | 1.74036 |
| he | 0.626419 |  | art | 1.74036 |  | 10 | 1.74036 |  | ba | 1.26324 |  | summer | 1.74036 |
| immortals | 1.74036 |  | tidying | 1.74036 |  | 8 | 1.74036 |  | the | 0.69897 |  | kristin | 1.1383 |
| evermore | 1.74036 |  | cashore | 1.43933 |  | kurt | 1.74036 |  | bryson | 1.43933 |  | 9 | 1.74036 |
| ee | 1.43933 |  | pyramid | 1.74036 |  | madeleine | 1.74036 |  | in | 0.69897 |  | kondo | 1.43933 |
| george | 1.74036 |  | days | 1.74036 |  | modern | 1.74036 |  | abbi | 1.74036 |  | kane | 1.74036 |
| armentrout | 1.1383 |  | susan | 1.43933 |  | kiss | 1.74036 |  | 1 | 0.69897 |  | rick | 0.962211 |
| romance | 1.74036 |  | farm | 1.74036 |  | gods | 1.74036 |  | up | 1.74036 |  | goldman | 1.74036 |
| barbara | 1.43933 |  | animal | 1.74036 |  |  | | | | | | | |

| **Searching module (./backend/components/information\_retrieval.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Generating unique words | Valid | Test database, with the additional data added | Unique words in above tables added to the database | Unique words in above tables added to the database | Using the test database, with the added information in it. | ✓ | - |
| 2 | Generating IDF values | Valid | Test database, with the additional data added | Data in above tables in unique\_words table | Data in above tables in unique\_words table | Using the test database, with the added information in it. | ✓ | - |
| 3 | Searching the database  (known ISBN number) | Valid | 0111111111 | {0: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': '', 'type': 'b', 'certainty': 100.0}} | {0: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': '', 'type': 'b', 'certainty': 100.0}} | Using the test database, with the added information in it. | ✓ | - |
| 3270810 | {0: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 100.0}} | {0: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 100.0}} | Using the test database, with the added information in it. | ✓ |
| 4 | Searching the database  (unknown ISBN number) | Erroneous | 1234567890 | {} | {} | Using the test database, with the added information in it. | ✓ | - |
| 5 | Searching the database  (known author) | Valid | Kristin Hannah | {0: {'name': 'Kristin Hannah', 'type': 'a', 'author\_id': 4, 'certainty': 100.0}, 1: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 100.0}, 2: {'name': 'Kristin Cashore', 'type': 'a', 'author\_id': 20, 'certainty': 62.0}, 3: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 62.0}} | {0: {'name': 'Kristin Hannah', 'type': 'a', 'author\_id': 4, 'certainty': 100.0}, 1: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 100.0}, 2: {'name': 'Kristin Cashore', 'type': 'a', 'author\_id': 20, 'certainty': 62.0}, 3: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 62.0}} | Using the test database, with the added information in it.  Tests that the order of words do not matter, and the punctuation and capitalisation does not have an effect  Tests that the search for an author also gives the books that they have written (if applicable) | ✓ | - |
| Kurt | {0: {'name': 'Kurt Vonnegut Jr.', 'type': 'a', 'author\_id': 8, 'certainty': 100.0}} | {0: {'name': 'Kurt Vonnegut Jr.', 'type': 'a', 'author\_id': 8, 'certainty': 100.0}} | ✓ |
| l'engle | {0: {'name': "Madeleine L'Engle", 'type': 'a', 'author\_id': 10, 'certainty': 100.0}} | {0: {'name': "Madeleine L'Engle", 'type': 'a', 'author\_id': 10, 'certainty': 100.0}} | ✓ |
| Armentrout, Jennifer | {0: {'name': 'Jennifer L. Armentrout', 'type': 'a', 'author\_id': 19, 'certainty': 100.0}, 1: {'author': 'Jennifer L. Armentrout', 'title': 'Onyx (Lux, #2)', 'book\_id': 19, 'cover': '', 'type': 'b', 'certainty': 100.0}, 2: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': '', 'type': 'b', 'certainty': 100.0}, 3: {'author': 'Jennifer L. Armentrout', 'title': 'Origin (Lux, #4)', 'book\_id': 21, 'cover': '', 'type': 'b', 'certainty': 100.0}} | {0: {'name': 'Jennifer L. Armentrout', 'type': 'a', 'author\_id': 19, 'certainty': 100.0}, 1: {'author': 'Jennifer L. Armentrout', 'title': 'Onyx (Lux, #2)', 'book\_id': 19, 'cover': '', 'type': 'b', 'certainty': 100.0}, 2: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': '', 'type': 'b', 'certainty': 100.0}, 3: {'author': 'Jennifer L. Armentrout', 'title': 'Origin (Lux, #4)', 'book\_id': 21, 'cover': '', 'type': 'b', 'certainty': 100.0}} | ✓ |
| 6 | Searching the database  (unknown author) | Erroneous | Bob | {} | {} | Using the test database, with the added information in it. | ✓ | - |
| Arbitrary writer | {} | {} | ✓ |
| 7 | Searching the database  (known genre) | Valid | Genre 1 | {0: {'name': 'Genre 1', 'type': 'g', 'certainty': 100.0}, 1: {'name': 'Genre 2', 'type': 'g', 'certainty': 72.7}, 2: {'name': 'Genre 3', 'type': 'g', 'certainty': 72.7}, 3: {'name': 'Genre 4', 'type': 'g', 'certainty': 72.7}, 4: {'name': 'Genre 5', 'type': 'g', 'certainty': 72.7}, 5: {'name': 'Genre 6', 'type': 'g', 'certainty': 72.7}, 6: {'name': 'Genre 7', 'type': 'g', 'certainty': 72.7}, 7: {'name': 'Genre 8', 'type': 'g', 'certainty': 72.7}, 8: {'name': 'Genre 9', 'type': 'g', 'certainty': 72.7}, 9: {'name': 'Genre 10', 'type': 'g', 'certainty': 72.7}, 10: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 68.6}, 11: {'name': 'Author 1', 'type': 'a', 'author\_id': 1, 'certainty': 68.6}, 12: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': '', 'type': 'b', 'certainty': 68.6}, 13: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': '', 'type': 'b', 'certainty': 68.6}, 14: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 68.6}, 15: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 68.6}, 16: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 68.6}, 17: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 68.6}, 18: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 68.6}} | {0: {'name': 'Genre 1', 'type': 'g', 'certainty': 100.0}, 1: {'name': 'Genre 2', 'type': 'g', 'certainty': 72.7}, 2: {'name': 'Genre 3', 'type': 'g', 'certainty': 72.7}, 3: {'name': 'Genre 4', 'type': 'g', 'certainty': 72.7}, 4: {'name': 'Genre 5', 'type': 'g', 'certainty': 72.7}, 5: {'name': 'Genre 6', 'type': 'g', 'certainty': 72.7}, 6: {'name': 'Genre 7', 'type': 'g', 'certainty': 72.7}, 7: {'name': 'Genre 8', 'type': 'g', 'certainty': 72.7}, 8: {'name': 'Genre 9', 'type': 'g', 'certainty': 72.7}, 9: {'name': 'Genre 10', 'type': 'g', 'certainty': 72.7}, 10: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 68.6}, 11: {'name': 'Author 1', 'type': 'a', 'author\_id': 1, 'certainty': 68.6}, 12: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': '', 'type': 'b', 'certainty': 68.6}, 13: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': '', 'type': 'b', 'certainty': 68.6}, 14: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 68.6}, 15: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 68.6}, 16: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 68.6}, 17: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 68.6}, 18: {'author': 'Kristin Cashore', 'title': 'Graceling (Graceling Realm, #1)', 'book\_id': 22, 'cover': '', 'type': 'b', 'certainty': 68.6}} | Using the test database, with the added information in it. | ✓ | - |
| Genre 2 | {0: {'name': 'Genre 2', 'type': 'g', 'certainty': 100.0}, 1: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 77.1}, 2: {'author': 'Jennifer L. Armentrout', 'title': 'Onyx (Lux, #2)', 'book\_id': 19, 'cover': '', 'type': 'b', 'certainty': 77.1}, 3: {'name': 'Author 2', 'type': 'a', 'author\_id': 2, 'certainty': 77.1}, 4: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': '', 'type': 'b', 'certainty': 77.1}, 5: {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': '', 'type': 'b', 'certainty': 77.1}, 6: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 77.1}, 7: {'name': 'Genre 1', 'type': 'g', 'certainty': 63.7}, 8: {'name': 'Genre 3', 'type': 'g', 'certainty': 63.7}, 9: {'name': 'Genre 4', 'type': 'g', 'certainty': 63.7}, 10: {'name': 'Genre 5', 'type': 'g', 'certainty': 63.7}, 11: {'name': 'Genre 6', 'type': 'g', 'certainty': 63.7}, 12: {'name': 'Genre 7', 'type': 'g', 'certainty': 63.7}, 13: {'name': 'Genre 8', 'type': 'g', 'certainty': 63.7}, 14: {'name': 'Genre 9', 'type': 'g', 'certainty': 63.7}, 15: {'name': 'Genre 10', 'type': 'g', 'certainty': 63.7}} | {0: {'name': 'Genre 2', 'type': 'g', 'certainty': 100.0}, 1: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 77.1}, 2: {'author': 'Jennifer L. Armentrout', 'title': 'Onyx (Lux, #2)', 'book\_id': 19, 'cover': '', 'type': 'b', 'certainty': 77.1}, 3: {'name': 'Author 2', 'type': 'a', 'author\_id': 2, 'certainty': 77.1}, 4: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': '', 'type': 'b', 'certainty': 77.1}, 5: {'author': 'Author 2', 'title': 'Book 4', 'book\_id': 4, 'cover': '', 'type': 'b', 'certainty': 77.1}, 6: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 77.1}, 7: {'name': 'Genre 1', 'type': 'g', 'certainty': 63.7}, 8: {'name': 'Genre 3', 'type': 'g', 'certainty': 63.7}, 9: {'name': 'Genre 4', 'type': 'g', 'certainty': 63.7}, 10: {'name': 'Genre 5', 'type': 'g', 'certainty': 63.7}, 11: {'name': 'Genre 6', 'type': 'g', 'certainty': 63.7}, 12: {'name': 'Genre 7', 'type': 'g', 'certainty': 63.7}, 13: {'name': 'Genre 8', 'type': 'g', 'certainty': 63.7}, 14: {'name': 'Genre 9', 'type': 'g', 'certainty': 63.7}, 15: {'name': 'Genre 10', 'type': 'g', 'certainty': 63.7}} |  |  |
| Genre 3 | {0: {'name': 'Genre 3', 'type': 'g', 'certainty': 100.0}, 1: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': '', 'type': 'b', 'certainty': 81.5}, 2: {'name': 'Author 3', 'type': 'a', 'author\_id': 3, 'certainty': 81.5}, 3: {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': '', 'type': 'b', 'certainty': 81.5}, 4: {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': '', 'type': 'b', 'certainty': 81.5}, 5: {'name': 'Genre 1', 'type': 'g', 'certainty': 57.9}, 6: {'name': 'Genre 2', 'type': 'g', 'certainty': 57.9}, 7: {'name': 'Genre 4', 'type': 'g', 'certainty': 57.9}, 8: {'name': 'Genre 5', 'type': 'g', 'certainty': 57.9}, 9: {'name': 'Genre 6', 'type': 'g', 'certainty': 57.9}, 10: {'name': 'Genre 7', 'type': 'g', 'certainty': 57.9}, 11: {'name': 'Genre 8', 'type': 'g', 'certainty': 57.9}, 12: {'name': 'Genre 9', 'type': 'g', 'certainty': 57.9}, 13: {'name': 'Genre 10', 'type': 'g', 'certainty': 57.9}} | {0: {'name': 'Genre 3', 'type': 'g', 'certainty': 100.0}, 1: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': '', 'type': 'b', 'certainty': 81.5}, 2: {'name': 'Author 3', 'type': 'a', 'author\_id': 3, 'certainty': 81.5}, 3: {'author': 'Author 3', 'title': 'Book 3', 'book\_id': 3, 'cover': '', 'type': 'b', 'certainty': 81.5}, 4: {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': '', 'type': 'b', 'certainty': 81.5}, 5: {'name': 'Genre 1', 'type': 'g', 'certainty': 57.9}, 6: {'name': 'Genre 2', 'type': 'g', 'certainty': 57.9}, 7: {'name': 'Genre 4', 'type': 'g', 'certainty': 57.9}, 8: {'name': 'Genre 5', 'type': 'g', 'certainty': 57.9}, 9: {'name': 'Genre 6', 'type': 'g', 'certainty': 57.9}, 10: {'name': 'Genre 7', 'type': 'g', 'certainty': 57.9}, 11: {'name': 'Genre 8', 'type': 'g', 'certainty': 57.9}, 12: {'name': 'Genre 9', 'type': 'g', 'certainty': 57.9}, 13: {'name': 'Genre 10', 'type': 'g', 'certainty': 57.9}} |  |  |
| 8 | Searching the database  (unknown genre) | Erroneous | Arbitrary Genre-name | {} | {} | Using the test database, with the added information in it.  It needs to be hyphenated so that it does not match all of the other Genre’s as they include genre in their names. | ✓ | - |
| 9 | Searching the database  (known book) | Valid | The animal | {0: {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': '', 'type': 'b', 'certainty': 92.8}, 1: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 37.3}, 2: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 37.3}, 3: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 37.3}, 4: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 37.3}, 5: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 37.3}, 6: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 37.3}, 7: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 37.3}, 8: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 37.3}, 9: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 37.3}, 10: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 37.3}, 11: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 37.3}} | {0: {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': '', 'type': 'b', 'certainty': 92.8}, 1: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 37.3}, 2: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 37.3}, 3: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 37.3}, 4: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 37.3}, 5: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 37.3}, 6: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 37.3}, 7: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 37.3}, 8: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 37.3}, 9: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 37.3}, 10: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 37.3}, 11: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 37.3}} | Using the test database, with the added information in it.  Demonstrates that inexact matches work as well | ✓ | - |
| The olympians | {0: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 95.0}, 1: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 95.0}, 2: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 43.7}, 3: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 43.7}, 4: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 43.7}, 5: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 43.7}, 6: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 43.7}, 7: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 43.7}, 8: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 43.7}, 9: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 43.7}, 10: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 43.7}} | {0: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 95.0}, 1: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 95.0}, 2: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 43.7}, 3: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 43.7}, 4: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 43.7}, 5: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 43.7}, 6: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 43.7}, 7: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 43.7}, 8: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 43.7}, 9: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 43.7}, 10: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 43.7}} | ✓ |
| The sword of summer magnus chase | {0: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 95.8}, 1: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 36.7}, 2: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 36.7}, 3: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 36.7}, 4: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 34.6}, 5: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 34.6}, 6: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 21.6}, 7: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 21.6}, 8: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 21.6}, 9: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 21.6}, 10: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 21.6}} | {0: {'author': 'Rick Riordan', 'title': 'The Sword of Summer (Magnus Chase and the Gods of Asgard, #1)', 'book\_id': 12, 'cover': '', 'type': 'b', 'certainty': 95.8}, 1: {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': '', 'type': 'b', 'certainty': 36.7}, 2: {'author': 'Susan Ee', 'title': 'Angelfall (Penryn & the End of Days, #1)', 'book\_id': 17, 'cover': '', 'type': 'b', 'certainty': 36.7}, 3: {'author': 'Rick Riordan', 'title': 'The Son of Neptune (The Heroes of Olympus, #2)', 'book\_id': 10, 'cover': '', 'type': 'b', 'certainty': 36.7}, 4: {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': '', 'type': 'b', 'certainty': 34.6}, 5: {'author': 'Mary E. Pearson', 'title': 'The Kiss of Deception (The Remnant Chronicles, #1)', 'book\_id': 8, 'cover': '', 'type': 'b', 'certainty': 34.6}, 6: {'author': 'Bill Bryson', 'title': 'A Walk in the Woods', 'book\_id': 18, 'cover': '', 'type': 'b', 'certainty': 21.6}, 7: {'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': '', 'type': 'b', 'certainty': 21.6}, 8: {'author': 'Rick Riordan', 'title': 'The Last Olympian (Percy Jackson and the Olympians, #5)', 'book\_id': 11, 'cover': '', 'type': 'b', 'certainty': 21.6}, 9: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': '', 'type': 'b', 'certainty': 21.6}, 10: {'author': 'Alyson Noel', 'title': 'Evermore (The Immortals, #1)', 'book\_id': 14, 'cover': '', 'type': 'b', 'certainty': 21.6}} | ✓ |
| 10 | Searching the database  (unknown book) | Erroneous | An arbitrary book's title | {} | {} | Using the test database, with the added information in it. | ✓ | - |

| ID | Before | After |
| --- | --- | --- |
| 1 |  |  |
| 2 |  |  |

For the reading lists tests, there is an additional MySQL file to add additional data to the project. This is done as there is insufficient books in the base test data to be added to all of the reading lists. It also includes some custom lists, and also adding values into the system-defined lists. These new entries in these lists would interfere with the other tests for other modules, so are included within a separate test data file.

| **Reading lists module (./backend/components/reading\_lists.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Get most popular books | Valid | None | {0: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, 1: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}, 2: {'author': 'Barbara Kingsolver', 'title': 'The Poisonwood Bible', 'book\_id': 16, 'cover': ''}, 3: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}, 4: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}, 5: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': ''}} | {0: {'author': 'Author 1', 'title': 'Book 5', 'book\_id': 5, 'cover': ''}, 1: {'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}, 2: {'author': 'Barbara Kingsolver', 'title': 'The Poisonwood Bible', 'book\_id': 16, 'cover': ''}, 3: {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}, 4: {'author': 'Author 2', 'title': 'Book 2', 'book\_id': 2, 'cover': ''}, 5: {'author': 'Jennifer L. Armentrout', 'title': 'Opal (Lux, #3)', 'book\_id': 20, 'cover': ''}} | Using test database with additional data | ✓ | - |
| 2 | Get list ID from user ID and name  (valid user ID and list name) | Valid | List name: Currently Reading  User ID: 1 | 2 | 2 | Using test database with additional data | ✓ | - |
| List name: Have Read  User ID: 3 | 9 | 9 | ✓ |
| List Name: Test list  User ID: 1 | 13 | 13 | ✓ |
| 3 | Get list ID from user ID and name  (valid user ID and invalid list name) | Erroneous | List Name: Random list  User ID: 3 | ListNotFoundError | ListNotFoundError | Using test database with additional data | ✓ | - |
| 4 | Get list ID from user ID and name  (invalid user ID and valid list name) | Erroneous | List Name: Have Read  User ID: 100 | ListNotFoundError | ListNotFoundError | Using test database with additional data | ✓ | - |
| 5 | Get user’s list names  (known user ID) | Valid | 1 | Queue object with items:  1. {'id': 1, 'name': 'Want to Read'}  2. {'id': 2, 'name': 'Currently Reading'}  3. {'id': 3, 'name': 'Have Read'}  4. {'id': 13, 'name': 'Test list'} | Queue object with items:  1. {'id': 1, 'name': 'Want to Read'}  2. {'id': 2, 'name': 'Currently Reading'}  3. {'id': 3, 'name': 'Have Read'}  4. {'id': 13, 'name': 'Test list'} | Using test database with additional data | ✓ | - |
| 2 | Queue object with items:  1. {'id': 4, 'name': 'Want to Read'}  2. {'id': 5, 'name': 'Currently Reading'}  3. {'id': 6, 'name': 'Have Read'}  4. {'id': 14, 'name': 'Test list'} | Queue object with items:  1. {'id': 4, 'name': 'Want to Read'}  2. {'id': 5, 'name': 'Currently Reading'}  3. {'id': 6, 'name': 'Have Read'}  4. {'id': 14, 'name': 'Test list'} | ✓ |
| 3 | Queue object with items:  1. {'id': 7, 'name': 'Want to Read'}  2. {'id': 8, 'name': 'Currently Reading'}  3. {'id': 9, 'name': 'Have Read'} | Queue object with items:  1. {'id': 7, 'name': 'Want to Read'}  2. {'id': 8, 'name': 'Currently Reading'}  3. {'id': 9, 'name': 'Have Read'} | ✓ |
| 4 | Queue object with items:  1. {'id': 10, 'name': 'Want to Read'}  2. {'id': 11, 'name': 'Currently Reading'}  3. {'id': 12, 'name': 'Have Read'} | Queue object with items:  1. {'id': 10, 'name': 'Want to Read'}  2. {'id': 11, 'name': 'Currently Reading'}  3. {'id': 12, 'name': 'Have Read'} | ✓ |
| 6 | Get user’s list names  (unknown user ID) | Erroneous | 400 | [] | [] | Using test database with additional data  An empty queue is not necessarily logical. However, it is never used outside of processes where the user ID is already checked. Therefore it does not need to be checked. | ✓ | - |
| 7 | Get books currently reading  (known user ID) | Valid | 1 | [{'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}] | [{'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}] | Using test database with additional data | ✓ | - |
| 2 | [{'author': 'Barbara Kingsolver', 'title': 'The Poisonwood Bible', 'book\_id': 16, 'cover': ''}, {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}] | [{'author': 'Barbara Kingsolver', 'title': 'The Poisonwood Bible', 'book\_id': 16, 'cover': ''}, {'author': 'Author 1', 'title': 'Book 1', 'book\_id': 1, 'cover': ''}] | ✓ |
| 5 | [] | [] | ✓ |
| 8 | Get books currently reading  (unknown user ID) | Erroneous | 400 | [] | [] | Using test database with additional data  This is not immediately logical, but as this is only used after validating the ID it is fine. As it also is used for the home page, regardless of whether the user exists, or there aren’t any items within the list, the section should be hidden, so having the same result for both is easier to test for. | ✓ | - |
| 9 | Get books that want to read  (known user ID) | Valid | 1 | [{'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': ''}, {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': ''}] | [{'author': 'Kristin Hannah', 'title': 'The Nightingale', 'book\_id': 6, 'cover': ''}, {'author': 'Rick Riordan', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'book\_id': 9, 'cover': ''}] | Using test database with additional data | ✓ | - |
| 2 | [{'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}, {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': ''}, {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': ''}] | [{'author': 'Rick Riordan', 'title': 'The Red Pyramid (Kane Chronicles, #1)', 'book\_id': 13, 'cover': ''}, {'author': 'Author 3', 'title': 'Animal Farm', 'book\_id': 7, 'cover': ''}, {'author': 'Marie Kondo', 'title': 'The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing', 'book\_id': 15, 'cover': ''}] | ✓ |
| 3 | [] | [] | ✓ |
| 10 | Get books that want to read  (unknown user ID) | Erroneous | 400 | [] | [] | Using test database with additional data  As with the unknown user ID fro the get currently reading, this is not that logical, but as it is checked before calling, shouldn’t ever happen | ✓ | - |
| 11 | Get reading list entries with  (valid list ID and user ID) | Valid | List ID: 1  User ID: 1 | ({0: {'id': 6, 'cover': '', 'title': 'The Nightingale', 'synopsis': '<p></p>', 'author': 'Kristin Hannah', 'author\_id': 4, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 0.0, 'num\_reviews': 0}, 1: {'id': 9, 'cover': '', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'synopsis': '<p></p>', 'author': 'Rick Riordan', 'author\_id': 9, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 0.0, 'num\_reviews': 0}}, 'Start Reading', 2) | ({0: {'id': 6, 'cover': '', 'title': 'The Nightingale', 'synopsis': '<p></p>', 'author': 'Kristin Hannah', 'author\_id': 4, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 0.0, 'num\_reviews': 0}, 1: {'id': 9, 'cover': '', 'title': 'The Sea of Monsters (Percy Jackson and the Olympians, #2)', 'synopsis': '<p></p>', 'author': 'Rick Riordan', 'author\_id': 9, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 0.0, 'num\_reviews': 0}}, 'Start Reading', 2) | Using test database with additional data | ✓ | - |
| List ID: 12  User ID: 4 | ({0: {'id': 2, 'cover': '', 'title': 'Book 2', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 2', 'author\_id': 2, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 4.0, 'num\_reviews': 3}, 1: {'id': 3, 'cover': '', 'title': 'Book 3', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 3', 'author\_id': 3, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 3.0, 'num\_reviews': 3}, 2: {'id': 4, 'cover': '', 'title': 'Book 4', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 2', 'author\_id': 2, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 2.75, 'num\_reviews': 4}}, None, None) | ({0: {'id': 2, 'cover': '', 'title': 'Book 2', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 2', 'author\_id': 2, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 4.0, 'num\_reviews': 3}, 1: {'id': 3, 'cover': '', 'title': 'Book 3', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 3', 'author\_id': 3, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 3.0, 'num\_reviews': 3}, 2: {'id': 4, 'cover': '', 'title': 'Book 4', 'synopsis': '<p>This book does not have a synopsis</p>', 'author': 'Author 2', 'author\_id': 2, 'date\_added': '14-02-2024', 'genres': ['Genre 1', 'Genre 2', 'Genre 3', 'Genre 4', 'Genre 5', 'Genre 6', 'Genre 7', 'Genre 8'], 'average\_rating': 2.75, 'num\_reviews': 4}}, None, None) | ✓ |
| List ID: 14  User ID: 2 | ({}, None, None) | ({}, None, None) | ✓ |
| 11 | Get reading list entries  (valid list ID and invalid user ID) | Erroneous | List ID: 1  User ID: 400 | ListNotFoundError | IndexError | Using test database with additional data  This does not pass the test. However, it is cannot get to this pointwithout validating the user ID. | ⛌ | ⛌ |
| 12 | Get reading list entries  (invalid list ID and valid user ID) | Erroneous | List ID: 14  User ID: 2 | ListNotFoundError | IndexError | Using test database with additional data  This does not pass the test, but as the buttons explicitly link to the list ID, it is not an issue, as the invalid list ID cannot be sent. | ⛌ | ⛌ |
| 13 | Add reading list entry  (valid user ID, list ID and book ID) | Valid | User ID: 1  List ID: 1  Book ID: 1 | Book removed from their Have Read list (list ID 3), and added to their Want to Read list (list ID 1) | Removed from their Have Read list  Added to list ID 1 | Using test database with additional data  This removes the entry from the Have Read list. This is because if the user wants to read the book, it can be assumed that they will need to re-add it to the have read list in future, so must be removed first. | ✓ |  |
| 14 | Add reading list entry  (valid user ID, invalid list ID and valid book ID) | Erroneous | User ID: 1  List ID: 4  Book ID: 1 | No change | Entry is added | Using test database with additional data  The entry is added to the other user’s list  When accessing the entries, and managing them the user ID is included within it, so it will not give this result. Furthermore, the webpage prevents this from happening, and as it does not need to deal with requests from an API, it is not an issue. | ⛌ | ⛌ |
| 15 | Add reading list entry  (valid user ID, valid list ID and invalid book ID) | Erroneous | User ID: 1  List ID: 1  Book ID: 400 | No change | Foreign key constraint fails error | Using test database with additional data | ⛌ | ✓ |
| No change | ✓ |
| 16 | Add reading list entry  (invalid user ID, valid list ID and valid book ID) | Erroneous | User ID: 400  List ID: 1  Book ID: 1 | No Change | No Change | Using test database with additional data | ✓ | - |
| 17 | Remove list entry  (valid list ID, user ID and book ID) | Valid | Insert book ID 1, into list ID 1  User ID: 1  List ID: 1  Book ID: 1 | Entry removed | Entry Removed | Using test database with additional data | ✓ | - |
| 18 | Remove list entry  (valid list ID, valid user ID and invalid book ID) | Erroneous | User ID: 1  List ID: 1  Book ID: 400 | No Change | No Change | Using test database with additional data | ✓ | - |
| 19 | Remove list entry list entries  (valid list ID, invalid user ID) | Erroneous | User ID: 1  List ID: 4  Book ID: 1 | No Change | No Change | Using test database with additional data | ✓ | - |
| 20 | Remove reading list  (valid user ID and list ID) | Valid | User ID: 1  List ID: 1 | Removal of all entries in list ID 1 from reading\_lists  Removal of list ID 1 from reading\_list\_names | Removal of all entries in list ID 1 from reading\_lists  Removal of list ID 1 from reading\_list\_names | Using test database with additional data | ✓ | - |
| 21 | Remove reading list  (invalid user ID and valid list ID) | Erroneous | User ID: 2  List ID: 2 | No change | No change | Using test database with additional data | ✓ | - |
| 22 | Remove reading list  (valid user ID and invalid list ID) | Erroneous | User ID: 400  List ID: 400 | No change | No change | Using test database with additional data | ✓ | - |
| 23 | Create a reading list | Valid | User ID: 3  List name: Testing list name | Record added to the reading\_list\_names table | Record added to the reading\_list\_names table | Using test database with additional data | ✓ | - |
| 24 | Get the most recent book read  (Books have been read) | Valid | 1 | (1, 'Book 1') | (1, 'Book 1') | Using test database with additional data | ✓ | - |
| 2 | (2, 'Book 2') | (2, 'Book 2') | ✓ |
| 3 | (1, 'Book 1') | (1, 'Book 1') | ✓ |
| 25 | Get the most recent book read  (No books have been read) | Boundary | 5 | None | None | Using test database with additional data | ✓ | - |
| 26 | Get the most recent book added to any list thats not Have Read  (Books have been added to lists) | Valid | 1 | (6, 'The Nightingale') | (6, 'The Nightingale') | Using test database with additional data | ✓ | - |
| 2 | (13, 'The Red Pyramid (Kane Chronicles, #1)') | (13, 'The Red Pyramid (Kane Chronicles, #1)') | ✓ |
| 3 | (7, 'Animal Farm') | (7, 'Animal Farm') | ✓ |
| 27 | Get the most recent book added to any list thats not Have Read  (No books have been added to lists) | Boundary | 5 | None | None | Using test database with additional data | ✓ | - |

For these screenshots, some of the tables have too many records in them, so cannot be comfortably displayed in the table. In these instances, the number of rows in the table is given before and after the test, so it is clear that the only addition is the one is that is expected, and shown in the screenshot of the table under the where clause. This makes it clear if there are records added that should not be outside of the shown records. An asterix next to the ID refers to the amended solution for the test, and its output.

| ID | Before | After |
| --- | --- | --- |
| 13 |  |  |
| 14 |  |  |
| 15\* |  |  |
| 16 |  |  |
| 17 |  |  |
| 18 |  |  |
| 19 |  |  |
| 20 |  |  |
| 21 |  |  |
| 22 |  |  |
| 23 |  |  |

| **Recommendations module (./backend/components/recommendations.py)** | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Test | Category | Data | Expected result | Actual Result | Comments | Passed? | Actioned? |
| 1 | Fitting the model | Valid |  |  |  |  | ✓ | - |
|  | Generating the recommendations | Valid |  |  |  |  |  |  |
|  | Delete a recommendation  (Good recommendation) | Valid |  |  |  |  |  |  |
|  | Delete a recommendation  (Bad recommendation) | Valid |  |  |  |  |  |  |
|  | Delete a recommendation  (valid user ID and invalid book ID) | Erroneous |  |  |  |  |  |  |
|  | Delete a recommendation  (invalid user ID and valid book ID) | Erroneous |  |  |  |  |  |  |
|  | Get list of bad recommendations  (known user ID) | Valid |  |  |  |  |  |  |
|  | Get list of bad recommendations  (known user ID with no marked bad recommendations) | Boundary |  |  |  |  |  |  |
|  | Get list of bad recommendations  (unknown user ID) | Erroneous |  |  |  |  |  |  |
|  | Get user recommendations  (known user ID) | Valid |  |  |  |  |  |  |
|  | Get user recommendations  (unknown user ID) | Erroneous |  |  |  |  |  |  |
|  | Get user recommendation summaries  (known user ID) | Valid |  |  |  |  |  |  |
|  | Get user recommendation summaries  (unknown user ID) | Erroneous |  |  |  |  |  |  |
|  | Add user to model  (known user ID and author IDs) | Valid |  |  |  |  |  |  |
|  | Add user to model  (known user ID and unknown author IDs) |  |  |  |  |  |  |  |
|  | Add user to model  (unknown user ID and known author IDs) |  |  |  |  |  |  |  |

| ID | Before | After |
| --- | --- | --- |
| 1 |  |  |

### WSGI script testing

The wsgi.py fiile is what responds to any HTTP requests that are made to the server. Consequently it needs to be tested to ensure that it is responding to, and handling the requests in the correct manner. Doing this outside of the website narrows down any errors that occur when making requests to the WSGI, rather than either the WSGI, JavaScript or HTML. This can be tested using the curl command through the command line. It is important to note that for these tests, wsgi.py unpacks the request data, and performs the relevant function, which has already been tested above. This means the outputs can be assumed to be correct, and as long as a response is given, it is working correctly. The following commands can be used to perform the request: The first performs a get request to the URL, which should include the GET parameters, and the second performs a POST request. With both commands the && echo adds an additional line after the resulting data just to ensure that the output is more readable, as it does not contain a new line at the end of the response data.

| curl <url> && echo  curl -H “<content type>” -d ‘<data>’ -X POST <url> && echo |
| --- |

These tests are made using additional data. For many of the response screenshots, the command is too long to fit in the available space, so has been wrapped to fit properly. As stated in the acceptable limitations, it only needs to support requests that can be made through the client, and therefore the invalid requests are mitigated through the JavaScript and HTML, limiting the user’s controls. Using the curl commands to test the WSGI bypasses this, so the requests reflect what is possible through the webpage. These following tests demonstrate that the WSGI script can respond correctly to all of the valid HTTP requests, either POST or GET that are issued by the JavaScript, returning the expected data, and performing any relevant alterations to the database

| ID | Test | Command | Response | Pass? |
| --- | --- | --- | --- | --- |
| 1 | Server error | curl localhost:8080/cgi-bin/my\_books/get\_list\_entries?session\_id=asdhjaksnce1263872613&list\_id=None |  | ✓ |
| 2 | 404 Error  (unknown handler | curl localhost:8080/cgi-bin/unknown\_handler/url && echo |  | ✓ |
| 3 | 404 Error  (known handler and unknown function) | curl localhost:8080/cgi-bin/account/url && echo |  | ✓ |
| 4 | Sign In  (Correct details) | curl -H "application/json" -d '{"username": "user1", "password": "password"}' -X POST localhost:8080/cgi-bin/account/sign\_in && echo |  | ✓ |
| 5 | Sign In  (invalid details) | curl -H "application/json" -d '{"username": "random", "password": "password"}' -X POST localhost:8080/cgi-bin/account/sign\_in && echo |  | ✓ |
| curl -H "application/json" -d '{"username": "user1", "pa  ssword": "incorrect"}' -X POST localhost:8080/cgi-bin/account/sign\_in && echo |  | ✓ |
| 6 | Sign out | curl -H "text/plain" -d 'asdhjaksnce1263872613' -X POST  localhost:8080/cgi-bin/account/sign\_out && echo |  | ✓ |
| 7 | Sign Up  (valid) | curl -H "application/json" -d '{"username": "test", "pas  sword": "new password!", "first\_name": "test", "surname": "user"}' -X POST localhost:8080/cgi  -bin/account/sign\_up && echo |  | ✓ |
| 8 | Sign Up  (username taken) | curl -H "application/json" -d '{"username": "user1", "pa  ssword": "new password!", "first\_name": "test", "surname": "user"}' -X POST localhost:8080/cg  i-bin/account/sign\_up && echo |  | ✓ |
| 9 | Get lists | curl localhost:8080/cgi-bin/my\_books/get\_lists?session\_id=asdhjaksnce1263872613 && echo |  | ✓ |
| 10 | Get list entries | curl 'localhost:8080/cgi-bin/my\_books/get\_list\_entries?session\_id=asdhjaksnce1263872613&list\_id=1' && echo |  | ✓ |
| 11 | Remove entry | curl -H "application/json" -d '{"list\_id": 9, "book\_id": 1, "user\_id": 3, "session\_id": "poipoqwerwrw983453453"}' -X POST localhost:8080/cgi-bin/my\_books/remove\_list\_entry && echo |  | ✓ |
| 12 | Add entry | curl -H "application/json" -d '{"list\_id": 3, "session\_id": "asdhjaksnce1263872613", "book\_id": 3}' -X POST localhost:8080/cgi-bin/my\_books/add\_list\_entry && echo |  | ✓ |
| 13 | Move entry | curl -H "application/json" -d '{"list\_id": 1, "target\_list\_id": 2, "session\_id": "asdhjaksnce1263872613", "book\_id": 6}' -X POST localhost:8080/cgi-b  in/my\_books/move\_list\_entry && echo |  | ✓ |
| 14 | Remove list | curl -H "application/json" -d '{"list\_id": 1, "session\_id": "asdhjaksnce1263872613"}' -X POST localhost:8080/cgi-bin/my\_books/remove\_list && echo |  | ✓ |
| 15 | Create list | curl -H "application/json" -d '{"list\_name": "new list", "session\_id": "asdhjaksnce1263872613"}' -X POST localhost:8080/cgi-bin/my\_books/create\_list && echo |  | ✓ |
| 16 | Get reading lists, and if a book is in them | curl 'localhost:8080/cgi-bin/my\_books/get\_lists\_book\_target?session\_id=asdhjaksnce1263872613&book\_id=1' && echo |  | ✓ |
| 17 | Get genre about data  (known) | curl localhost:8080/cgi-bin/genres/about\_data?genre\_name=Genre%201 && echo |  | ✓ |
| 18 | Get genre about data  (unknown) | curl localhost:8080/cgi-bin/genres/about\_data?genre\_name=Unknown%20Genre && echo |  | ✓ |
| 19 | Get book about data with signed in user  (known) | curl 'localhost:8080/cgi-bin/books/about\_data?book\_id=1&session\_id=asdhjaksnce1263872613' && echo |  | ✓ |
| 20 | Get book about data without signed in user  (known) | curl 'localhost:8080/cgi-bin/books/about\_data?book\_id=1&session\_id=null' && echo |  | ✓ |
| 21 | Get book about data  (unknown) | curl 'localhost:8080/cgi-bin/books/about\_data?book\_id=100&session\_id=null' && echo |  | ✓ |
| 22 | Delete a review | curl -H "application/json" -d '{"review\_id": 3, "session\_id": "asdhjaksnce1263872613"}' -X POST localhost:8080/cgi-bin/books/delete\_review && echo |  | ✓ |
| 23 | Leave review | curl -H "application/json" -d '{"session\_id": "asdhjaksnce1263872613", "book\_id": 7, "overall\_rating": 4, "plot\_rating": 5, "character\_rating": 3, "summary": "This is the review summary", "thoughts": "This is the review body"}' -X POST localh  ost:8080/cgi-bin/books/add\_review && echo |  | ✓ |
| 24 | Follow author | curl -H "application/json" -d '{"session\_id": "swcdecwftrbr132788943", "author\_id": 1}' -X POST localhost:8080/cgi-bin/authors/follow\_author && echo |  | ✓ |
| 25 | Unfollow an author | curl -H "application/json" -d '{"session\_id": "swcdecwftrbr132788943", "author\_id": 2}' -X POST localhost:8080/cgi-bin/authors/unfollow\_author && echo |  | ✓ |
| 26 | Get author about data  (known) | curl localhost:8080/cgi-bin/authors/about\_data?author\_id=1 && echo |  | ✓ |
| 27 | Get author about data  (unknown) | curl localhost:8080/cgi-bin/authors/about\_data?author\_id=100 && echo |  | ✓ |
| 28 | Get diary entries | curl localhost:8080/cgi-bin/diary/get\_entries?session\_id=adqweqiueqw0812309812 && echo |  | ✓ |
| 29 | Delete a diary entry | curl -H "application/json" -d '{"session\_id": "zxmcabvzxcn1231231235", "entry\_id": 2}' -X POST localhost:8080/cgi-bin/diary/delete\_entry && echo |  | ✓ |
| 30 | Add a diary entry | curl -H "application/json" -d '{"session\_id": "zxmcabvzxcn1231231235", "book\_id": 2, "overall\_rating": 5, "plot\_rating": 5, "character\_rating": 4, "pages\_read": 15, "summary": "Diary entry summary", "thoughts": "Diary entry main body", "book\_completed": false, "as\_review": false}' -X POST localhost:8080/cgi-bin/diary/add\_entry && echo |  | ✓ |
| 31 | Get home data  (user signed in) | curl localhost:8080/cgi-bin/home/get\_data?session\_id=poipoqwerwrw983453453 && echo |  | ✓ |
| 32 | Get home data  (user signed out) | curl localhost:8080/cgi-bin/home/get\_data?session\_id=null && echo |  | ✓ |
| 33 | Get user recommendations | curl localhost:8080/cgi-bin/recommendations/get\_recommendations?session\_id=asdhjaksnce1263872613 && echo |  | ✓ |
| 34 | Remove recommendation | curl -H "application/json" -d '{"session\_id": "asdhjaksn  ce1263872613", "book\_id": 1}' -X POST localhost:8080/cgi-bin/recommendations/remove\_recommend  ation && echo |  | ✓ |
| 35 | Move to want to read list | curl -H "application/json" -d '{"session\_id": "asdhjaksnce1263872613", "book\_id": 1, "list\_id": 1}' -X POST localhost:8080/cgi-bin/recommendations/ad  d\_list\_entry && echo |  | ✓ |
| 36 | Set user preferences | curl -H "application/json" -d '{"session\_id": "asdhjaksnce1263872613", "authors": ["1", "2", "3"]}' -X POST localhost:8080/cgi-bin/recommendations/set\_user\_preferences && echo |  | ✓ |
| 37 | Search the database | curl localhost:8080/cgi-bin/search/search?query=rick%20olympians && echo |  | ✓ |
| 38 | Get browse page data  (no user) | curl localhost:8080/cgi-bin/search/get\_browse\_data?session\_id=null && echo |  | ✓ |
| 39 | Get browse page data  (user signed in) | curl localhost:8080/cgi-bin/search/get\_browse\_data?session\_id=zxmcabvzxcn1231231235 && echo |  | ✓ |

| ID | Before | After |
| --- | --- | --- |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |
| 15 |  |  |
| 22 |  |  |
| 23 |  |  |
| 24 |  |  |
| 25 |  |  |
| 29 |  |  |
| 30 |  |  |
| 34 |  |  |
| 35 |  |  |
| 36 |  |  |

## Frontend

As the website is interactive, screenshots do not adequately demonstrate and prove the functionality of the website. Consequently a video has been used to prove this, the link for which is below. Through this it demonstrates that the website follows the design given in the documented design, that the JavaScript requests and displays the relevant data correctly, the requests be received, processed, and logged by the webserver, and the website loading and changing content dynamically, without page refreshes. This is done using the dump.sql test data.

# Evaluation

| **It must use a website** | |
| --- | --- |
| The website should have a modern design. |  |
| The website must be easy and intuitive to navigate for third parties. |  |
| Each book stored in the database should have a dedicated about page, which displays all the available information about it in the system. |  |
| Each genre stored in the database should have a dedicated *about* page, which displays all the available information within the system. |  |
| Each author stored in the database should have a dedicated *about* page, which displays all the available information within the system. |  |
| The users’ reading lists must be available on a dedicated webpage on the website. |  |
| The users’ recommendations must be available on a dedicated webpage on the website. |  |
| The users’ diary entries must be available on a dedicated webpage on the website. |  |
| The website must use a hierarchical website structure, with all pages being able to be navigated to from the home page, or further pages in a logical format. |  |
| The website’s colour theme should be white, black, navy, grey and red colour scheme. |  |
| The website should be able to support at least five concurrent users. |  |
| Pages should be switched and navigated between without refreshing the page |  |
| **The website’s home page should present users with a list of frequently accessed information.** | |
| The home page should give users a summary of their recommended books. |  |
| The home page should give users a summary of what is in their want to read list. |  |
| The home page should give users a summary of what is in their currently reading list. |  |
| The home page should give users a list of trending books, which have the highest number of users currently reading them. |  |
| Book summaries should include the book’s cover image, author, and title. |  |
| Each book summary should link the specific books about the page. |  |
| The home page should have links to the other main sub-pages so that users can navigate easily around it. |  |
| **Users should have their own accounts on the website.** | |
| Users must be able to create their own accounts on the website. |  |
| Users must be able to log in to the website to access their personalised information. |  |
| Recommendations and reading list entries must be kept private. |  |
| Each user’s diary entries must be kept private |  |
| The website should be able to save a user’s login |  |
| **Personal information that is stored within the database should use appropriate data protection methods.** | |
| All passwords should be hashed before being stored in the database. |  |
| **Users should be able to create collections of books called reading lists** | |
| Users should be able to record the books that they have read in the past. |  |
| Users should be able to record the books that they are currently reading. |  |
| Users should be able to record the books that they would like to read in the future. |  |
| *Have Read*, *Currently Reading*, and *Want to Read* reading lists should be automatically created when a user signs up. |  |
| Users should be able to create their own custom reading lists |  |
| The user should not be able to delete the default *Have Read*, *Currently Reading*, and *Want to Read* reading lists |  |
| Users should be able to delete their own custom lists |  |
| Each entry into a reading list should show the book’s cover image, title, synopsis, genres, and authors, title, the average rating, and the date it was added to the list |  |
| Each entry into a reading list should allow the user to delete the entry. |  |
| Each entry into the *have read* reading list should also supply a link to all the user’s diary entries for that book. |  |
| Each entry into a reading list should supply a link to the books about page. |  |
| Reading lists, and the entries within them should be specific to the user. |  |
| Each entry into the *Currently Reading* reading list should supply a method to mark an entry as read and move it to the *have read* reading list. |  |
| Each entry into the *Want to Read* reading list should supply a way to the user to start reading the book and move the entry to the *currently reading* list. |  |
| **The system should generate recommendations for the users.** | |
| The system should generate recommendations that are unique to each individual user’s tastes. |  |
| When signing up for the first time, users should be prompted for their favourite authors and genres. |  |
| Recommendations displayed on the dedicated recommendations page should show the cover image, author, synopsis, average rating, date recommended and the genre. |  |
| Recommendation entries should have a method to remove the recommendation, and mark it as a bad recommendation. |  |
| Recommendation entries should have a method to move it to the *want to read* reading list. |  |
| Recommendations should include whether the user follows an author |  |
| Recommendations should include whether the book is in a reading list |  |
| Recommendations should include the user’s diary entries |  |
| Recommendations should be regenerated daily |  |
| Recommendation entries should show the certainty of the recommendation |  |
| **Users should be able to have a diary to record what they have read.** | |
| Users should be able to create entries about each book, giving their ratings for the characters and plot, their thoughts, and feelings about the part of the book they have read and their overall rating. |  |
| Users must specify the number of pages that they read. |  |
| When the user finishes the book, the user should be given the option to make the diary entry available as a review. |  |
| Each diary entry should supply a link to the corresponding book about page. |  |
| Users should be able to delete diary entries |  |
| **Each book should have an about page, which gives all the related information about the book.** | |
| The page should show the title, author, genres, synopsis, cover image, average rating, user ratings, and a link to buy the book |  |
| A link to buy the book should be provided, and the current price of the book should be displayed, web scraped from the website. |  |
| It should have a method to add the book to the have read, currently reading, and want to read lists. |  |
| There should be a list of books that are related to each other displayed. |  |
| Each related book should link to the corresponding about page. |  |
| **Each author should have a profile page.** | |
| It should display the author’s average book rating. |  |
| It should display all the genres that the author writes for. |  |
| It should show all the books that the author has written. |  |
| It should supply an about the author section. |  |
| It should show the author’s name (or the name that they write under if applicable). |  |
| Users should be able to follow each author. |  |
| The page should display how many followers each author has. |  |
| **Each author should have a profile page.** | |
| It should display the author’s average book rating. |  |
| It should display all the genres that the author writes for. |  |
| It should show all the books that the author has written. |  |
| It should supply an about the author section. |  |
| It should show the author’s name (or the name that they write under if applicable). |  |
| Users should be able to follow each author. |  |
| The page should display how many followers each author has. |  |
| **Users should be able to leave reviews about books that they have read.** | |
| Users should be able to leave reviews about each book |  |
| Users should be able to delete any reviews that they have left |  |
| Reviews should be listed on the corresponding book about page. |  |
| Reviews must have an overall rating. |  |
| Reviews can have a character rating, plot rating, review summary, and review body. |  |
| Reviews on the book should show the user’s username. |  |
| Users should only be able to review a book if they have read it, marked by it being in the users’ *have read* list. |  |
| **Each genre should have an about page** | |
| It should display the genre’s name |  |
| It should show an about summary for the genre, if applicable |  |
| It should show the books that are part of the genre |  |
| **There should be a quick way for users to find books, authors and genres.** | |
| There should be a browse page on the website to display popular books, highest rated, and the newest additions the website |  |
| The browse page should display books related to the last one added to a reading list |  |
| The browse page should display books related to the last read book |  |
| The browse page should display books from the authors the user is following |  |
| There must be a search bar that searches for books, authors and genres within the website. |  |
| The search bar should be able to search by name or ISBN number |  |
| **There should be a simple way to configure the system** | |
| Parameters that can affect performance should be configurable |  |
| A default configuration should be used, which can be updated as needed based upon available hardware |  |
| Recommendations should be configurable based upon the books and data stored in the database |  |
| Issues with the configuration should be clear, easy to understand, and simple to debug |  |
| Configurable parameters should be modified through an external configuration file, which is in a clear and readable format |  |

This project was initially thought of as an isolated recommendation system for books. Reading is one of my interests, and therefore this became something that I was significantly interested in, leading to me considering features and functionality into the project that I would personally find beneficial, through this adding the reading diaries and reading lists features. However, through conversations with end users, the importance, and demand for these was emphasised, and their importance was increased. This then led to their inclusion within the project. Furthermore through conversations with the end users, desire for additional features was expressed, including book reviews, about pages for books, authors and genres, similar books, and the ability to follow other users. Consequently the scope of the project continued to grow. Through my lack of experience with projects of this nature, and project management in general, I did not know whether it was achievable or not. As a result, I ended up working extremely hard to meet the objectives that I had set at the start of the project. This project did end up significantly exceeding the expected scope of an A-level project, and even though many parts of the project were fairly simple to develop, it was still very time consuming. This combined lack of knowledge and requests from end users led to the project becoming this large.

—----------------------------------

Issues:

* Goes against GDPR
  + Users cannot access their data
  + Data is not all encrypted
  + Users cannot delete accounts and all associated data
* Cookies are used but there is not any way to decline them, or any message saying they are used.
* Books can only have one author
* Users can create lists of the same name - will not work for situations where it is checking for a system-created list
* Genre about pages will list almost every book in the database
* Loading book pages is slow because all reviews left are shown on the page
* Loading reading lists can be slow, because all entries are loaded at once
* Loading diaries is slow as they are all loaded at once
* There is no way to sort reviews or access the reviews of a specific star
* Does not work with spam-clicks
* Does not work properly with small displays (860px wide or less)
* Back and forwards buttons do not work

Improvements

* The components objects are not easy to create as some are dependent on others to be created
* Server needs to be restarted when a new genre is created
* There is not an easy way to add new books or authors
* Users should be able to create and share lists between multiple people, so they have collaborative ones
* Some reviews may contain spoilers. There should be a setting to mark as to whether a review has a spoiler, so it can be hidden
* Books can be added to reading lists from book about pages, but cannot be removed from lists from the about page.
* If a sign in is required, exiting the popup takes the user back to the home screen. It would be more logical for the user to be taken back to the last visited page instead, as this would be clearer. This would not work with the search feature, however.
* It is not very well optimised. Creating the pages client side would be more efficient
* No password rules or checks.

—----------------------------------

ToDo

* Copy and paste objectives with discussion about each one
* Indepnedant feedback - contact end users
* **Reflect on project size and scope**
* Improvements and issues

# Bibliography

Alake, R. (2021, November 3). *Understanding Cosine Similarity And Its Application*. Medium. https://towardsdatascience.com/understanding-cosine-similarity-and-its-application-fd42f585296a

Anderson, S. (2018, January 10). *How Fast Should Your Web Page Load & How To Speed Up Your Website*. Hobo. https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/

Apache. (2013). *mod\_fcgid - FastCGI interface module for Apache 2 - The Apache HTTP Server Project*. Httpd.apache.org. https://httpd.apache.org/mod\_fcgid/

Apache. (2015). *Welcome! - The Apache HTTP Server Project*. Apache.org. https://httpd.apache.org/

baeldung. (2023, May 24). *What Is the Difference Between GET and POST Methods?* Baeldung. https://www.baeldung.com/cs/http-get-vs-post

Bard, A. (2016, August 26). *Python Web Applications: The basics of WSGI — SitePoint*. Www.sitepoint.com. https://www.sitepoint.com/python-web-applications-the-basics-of-wsgi/

Blaisdell, M. (n.d.). *Book title: What’s a good length? - article*. Author Learning Center. Retrieved January 1, 2023, from https://www.authorlearningcenter.com/publishing/preparation/w/choosing-a-title/2214/book-title-what-s-a-good-length---article

Brans, P. (2022, August). *What is a Fuzzy Search?* WhatIs.com. https://www.techtarget.com/whatis/definition/fuzzy-search

BYJU'S. (n.d.). *What advantages do tuples have over lists?* Byjus.com. Retrieved December 9, 2023, from https://byjus.com/question-answer/what-advantages-do-tuples-have-over-lists/

Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N. L. U., John, R. S., Constant, N., Guajardo-Céspedes, M., Yuan, S., Tar, C., Sung, Y., Strope, B., & Kurzweil, R. (2018). *Universal Sentence Encoder*. Google Research. https://research.google/pubs/pub46808/

Cloudflare. (n.d.). Why use HTTPS. *Cloudflare*. Retrieved April 28, 2023, from https://www.cloudflare.com/en-gb/learning/ssl/why-use-https/

Cloudflare. (2022). What Is DNS? | How DNS Works | Cloudflare UK. *Cloudflare*. https://www.cloudflare.com/en-gb/learning/dns/what-is-dns/

Cloudflare. (2023). What is HTTPS? | Cloudflare UK. *Cloudflare*. https://www.cloudflare.com/en-gb/learning/ssl/what-is-https/

Donovan, M. (2019, January 29). *The Benefits of Keeping a Reading Journal | Writing Forward*. Writing Forward. https://www.writingforward.com/better-writing/reading-journal

Eby, Phillip. J. (2010, August 26). *PEP 3333 – Python Web Server Gateway Interface v1.0.1 | peps.python.org*. Peps.python.org. https://peps.python.org/pep-3333/

Engati. (n.d.). *Bag of words Model*. Engati. Retrieved December 30, 2022, from https://www.engati.com/glossary/bag-of-words

EnoVision. (2021). *Google Chrome access using self-signed certificate on Macos*. Enovision.net. https://www.enovision.net/google-chrome-access-using-self-signed-certificate-macos

Entrust. (n.d.). *What is a Self-Signed Certificate*. Www.entrust.com. Retrieved April 30, 2023, from https://www.entrust.com/resources/faq/what-is-a-self-signed-certificate

Ethen. (2018, April 7). *1\_ALSWR*. Ethen8181.Github.io. https://ethen8181.github.io/machine-learning/recsys/1\_ALSWR.html

Fielding, R., Nottingham, M., & Reschke, J. (2022, June). *RFC 9110: HTTP Semantics*. Www.rfc-Editor.org. https://www.rfc-editor.org/rfc/rfc9110

Frisbie, M., & Zach Tratar. (2020). *Professional JavaScript for Web developers*. Wrox, A Wiley Brand.

Ganesan, K. (2014, October 19). *What are Stop Words?* Kavita Ganesan, PhD. https://kavita-ganesan.com/what-are-stop-words/#.ZE6LvnbMJD8

GeeksforGeeks. (2018, November 30). *Python Program for Binary Search (Recursive and Iterative) - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/python-program-for-binary-search/

Gillis, A. S. (2020, July). *What is a Web Server and How Does it Work?* WhatIs.com. https://www.techtarget.com/whatis/definition/Web-server

Goodreads. (2018). *Goodreads*. Goodreads. https://www.goodreads.com

Google. (n.d.). *TensorFlow Hub*. Tfhub.dev. Retrieved January 1, 2023, from https://tfhub.dev/google/collections/universal-sentence-encoder/1

Google for Developers. (2022a, July 18). *Collaborative Filtering | Recommendation Systems*. Google Developers. https://developers.google.com/machine-learning/recommendation/collaborative/basics

Google for Developers. (2022b, July 18). *Content-based Filtering | Recommendation Systems*. Google Developers. https://developers.google.com/machine-learning/recommendation/content-based/basics

Google for Developers. (2022c, July 18). *Deep Neural Network Models | Machine Learning*. Google Developers. https://developers.google.com/machine-learning/recommendation/dnn/softmax

Hernandez, J. (2019, May 8). *What is Apache? In-Depth Overview of Apache Web Server*. Sumo Logic. https://www.sumologic.com/blog/apache-web-server-introduction/

IBM. (2017, July 18). *What is Ajax?* Www.ibm.com. https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=page-asynchronous-javascript-xml-ajax-overview

IBM. (2022, December 29). *CGI*. Www.ibm.com. https://www.ibm.com/docs/en/i/7.4?topic=functionality-cgi

ICO. (2019, October 4). *Security*. Ico.org.uk; ICO. https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/security/

IONOS. (2020, September 14). *What is CGI?* IONOS Digitalguide. https://www.ionos.co.uk/digitalguide/websites/web-development/what-is-a-cgi/

Khan Academy. (n.d.). *The World Wide Web (article) | Web protocols*. Khan Academy. Retrieved April 28, 2023, from https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:web-protocols/a/the-world-wide-web

Kinsta. (2022, January 26). *What Is Nginx? A Basic Look at What It Is and How It Works*. Kinsta Managed WordPress Hosting. https://kinsta.com/knowledgebase/what-is-nginx/

Learn the Web. (n.d.). *Website file organization · Web Dev Topics · Learn the Web*. Learn the Web. Retrieved April 28, 2023, from https://learntheweb.courses/topics/organization/

LibraryThing. (2011). *LibraryThing | Catalog your books online*. Librarything.com. https://www.librarything.com/

Lighttpd. (n.d.-a). *Mod fastcgi - Lighttpd - lighty labs*. Redmine.lighttpd.net. Retrieved November 25, 2023, from https://redmine.lighttpd.net/projects/lighttpd/wiki/Mod\_fastcgi

Lighttpd. (n.d.-b). *Mod rewrite - Lighttpd - lighty labs*. Redmine.lighttpd.net. Retrieved November 25, 2023, from https://redmine.lighttpd.net/projects/lighttpd/wiki/Mod\_rewrite

Lighttpd. (n.d.-c). *WikiStart - Lighttpd - lighty labs*. Redmine.lighttpd.net. Retrieved November 25, 2023, from https://redmine.lighttpd.net/projects/lighttpd/wiki/WikiStart#Documentation

Lighttpd. (2023, February 10). *Home - Lighttpd - fly light*. Www.lighttpd.net. https://www.lighttpd.net/

Mahapatra, S. (2019, March 30). *Use-cases of Google’s Universal Sentence Encoder in Production*. Medium. https://towardsdatascience.com/use-cases-of-googles-universal-sentence-encoder-in-production-dd5aaab4fc15

Manning, C. D., Raghavan, P., Hinrich Schütze, & University Of Cambridge. (2009). *Introduction to information retrieval*. Cambridge University Press.

Meltem, S., Turan, E., Barker, W., Burr, L., Chen, Locke, G., & Gallagher, P. (2010). *NIST Special Publication 800-132 Recommendation for Password-Based Key Derivation Part 1: Storage Applications*. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

Menzli, A. (2021, June 24). *Tokenization in NLP: Types, Challenges, Examples, Tools*. Neptune.ai. https://neptune.ai/blog/tokenization-in-nlp

Morris, W. (2022, June 3). *What Is Lighttpd Web Server?* Elegant Themes Blog. https://www.elegantthemes.com/blog/wordpress/lighttpd-web-server

Mozilla. (n.d.). *Synchronous and asynchronous requests - Web APIs | MDN*. Developer.mozilla.org. Retrieved April 27, 2023, from https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous\_and\_Asynchronous\_Requests

Mozilla. (2019, February 19). *An overview of HTTP*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Mozilla. (2023a, April 10). *GET - HTTP | MDN*. Developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

Mozilla. (2023b, April 10). *POST*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

Nginx. (n.d.). *What is NGINX?* NGINX. Retrieved May 2, 2023, from https://www.nginx.com/resources/glossary/nginx/

Omniconvert. (2023, May 11). *What is Dynamic Content? Definition & Examples | Omniconvert - Omniconvert*. Omniconvert.com. https://www.omniconvert.com/what-is/dynamic-content/

ORO. (n.d.). *Available HTTP Methods - OroCommerce, OroCRM and OroPlatform Documentation*. Oro Documentation. Retrieved April 28, 2023, from https://doc.oroinc.com/api/http-methods/

*Parsing the Request - Post — WSGI Tutorial*. (n.d.). Wsgi.tutorial.codepoint.net. https://wsgi.tutorial.codepoint.net/parsing-the-request-post

Porter, M. F. (2006). An algorithm for suffix stripping. *Program*, *40*(3), 211–218. https://doi.org/10.1108/00330330610681286

Prakriti. (2022, May 13). *Code Studio*. Www.codingninjas.com. https://www.codingninjas.com/codestudio/library/smoothing-in-nlp

Prasad, Pratiksha. R. (2018, January 12). *A Detailed Study of WSGI: Web Server Gateway Interface for Python - DZone*. Dzone.com. https://dzone.com/articles/a-detailed-study-of-wsgi-web-server-gateway-interf-1

PythonAnywhere. (n.d.). *Host, run, and code Python in the cloud: PythonAnywhere*. Www.pythonanywhere.com. Retrieved April 27, 2023, from https://www.pythonanywhere.com/

Rastogi, K. (2022, January 31). *Text Cleaning Methods in NLP*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/01/text-cleaning-methods-in-nlp/

Rees, M. (2016, April 26). *The Open Source Developers’ Conference Papers» WSGI - Gateway or Glue?* Web.archive.org. https://web.archive.org/web/20160426075107/http://osdcpapers.cgpublisher.com/product/pub.84/prod.21/m.1

Rosenthal, E. (2016, January 9). *Explicit Matrix Factorization: ALS, SGD, and All That Jazz | Ethan Rosenthal*. Www.ethanrosenthal.com. https://www.ethanrosenthal.com/2016/01/09/explicit-matrix-factorization-sgd-als/

Saha, B., Garcia, E., G. C, S., & Li, Y. (2020, August 3). *Artificial Intelligence Project Identifying phishing emails targeting a higher education institution*. Research Gate. https://www.researchgate.net/publication/343399598\_Artificial\_Intelligence\_Project\_Identifying\_phishing\_emails\_targeting\_a\_higher\_education\_institution

Simha, A. (2021, October 7). *Understanding TF-IDF for Machine Learning*. Capital One. https://www.capitalone.com/tech/machine-learning/understanding-tf-idf/

Smirnova, S. (2023, September 1). *Understanding MySQL Triggers: Exploring How Triggers Impact MySQL Memory Allocation*. Percona Database Performance Blog. https://www.percona.com/blog/how-triggers-may-significantly-affect-the-amount-of-memory-allocated-to-your-mysql-server/

Sopha. (2022, May 24). *URI vs URL: Differences and When to Use Them*. Hostinger Tutorials. https://www.hostinger.co.uk/tutorials/uri-vs-url

StackPath. (2023). *What is dynamic content?* Stackpath.com. https://www.stackpath.com/edge-academy/what-is-dynamic-content/

Stein, C. (2019, September 5). *Basic Website Folder/File Structure – MMP 240 Web Design*. Basic Website Folder/File Structure. https://openlab.bmcc.cuny.edu/mmp-240-fall-19-stein/2019/09/05/basic-website-folder-file-structure/

superhosting.bg. (2016, April 27). *What is CGI, FastCGI?* Help.superhosting.bg. https://help.superhosting.bg/en/cgi-common-gateway-interface-fastcgi.html

Tutorials Point. (2023). *Python - Binary Tree - Tutorialspoint*. Www.tutorialspoint.com. https://www.tutorialspoint.com/python\_data\_structure/python\_binary\_tree.htm

w3schools. (2019). *How To Create a Sign Up Form*. W3schools.com. https://www.w3schools.com/howto/howto\_css\_signup\_form.asp

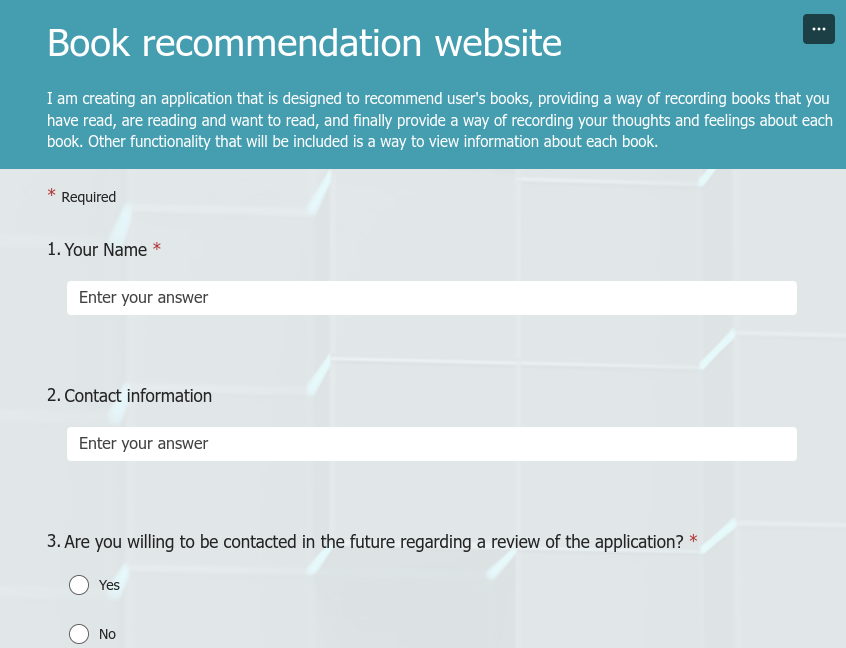
Waterstones. (2019). *Buy books, stationery and gifts, online and in store | Waterstones*. Waterstones.com. https://www.waterstones.com/

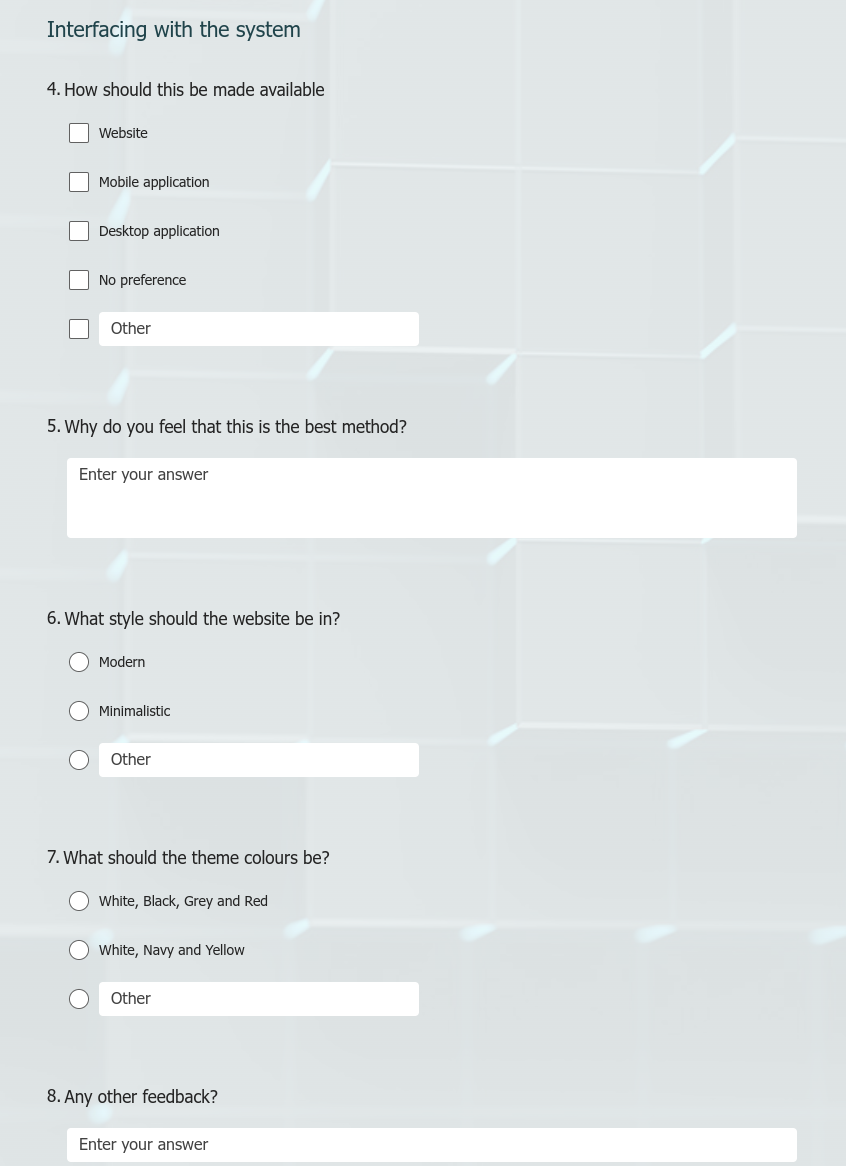
Williams, M. (2022, August 7). *Readers prefer “real” books rather than ebooks or audiobooks - writes a renewable pulp and paper manufacturer*. The New Publishing Standard. https://thenewpublishingstandard.com/2022/08/07/readers-prefer-real-books-rather-than-ebooks-or-audiobooks-writes-a-renewable-pulp-and-paper-manufacturer

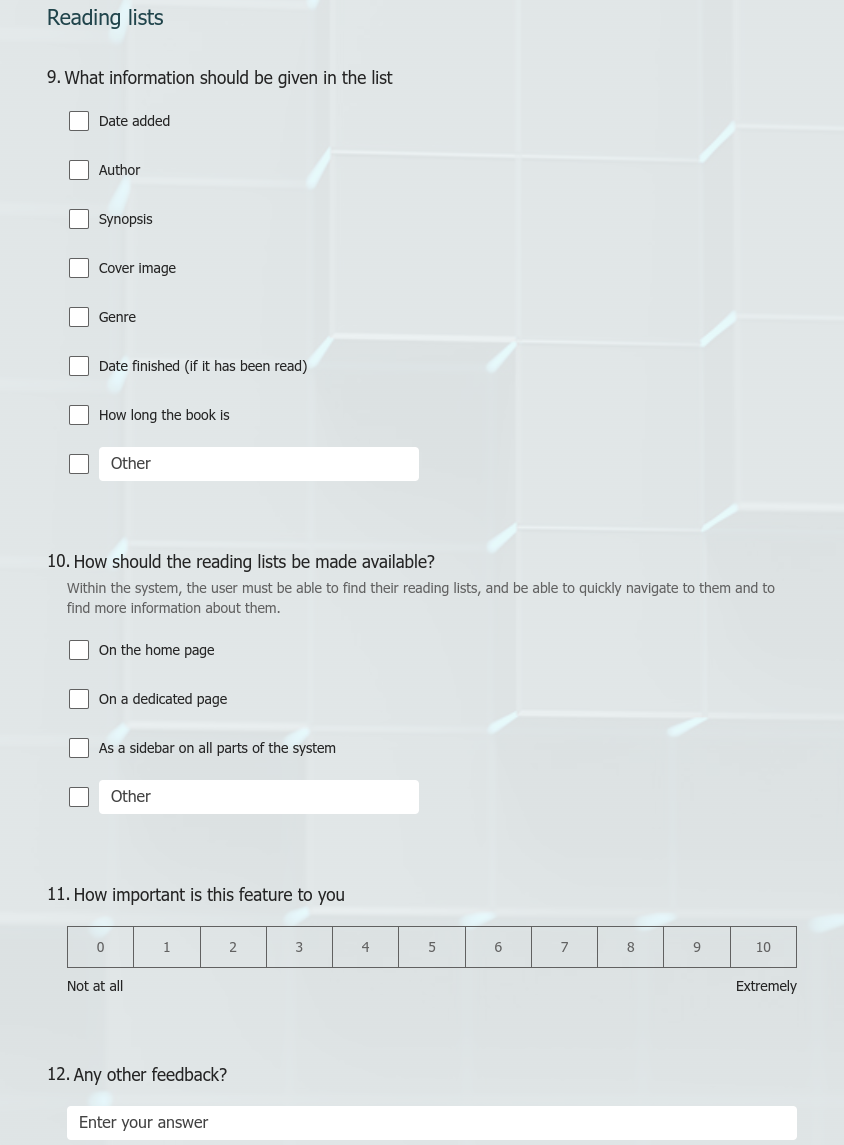
YAML. (2021, October 1). *YAML Ain’t Markup Language (YAMLTM) revision 1.2.2*. Yaml.org. https://yaml.org/spec/1.2.2/

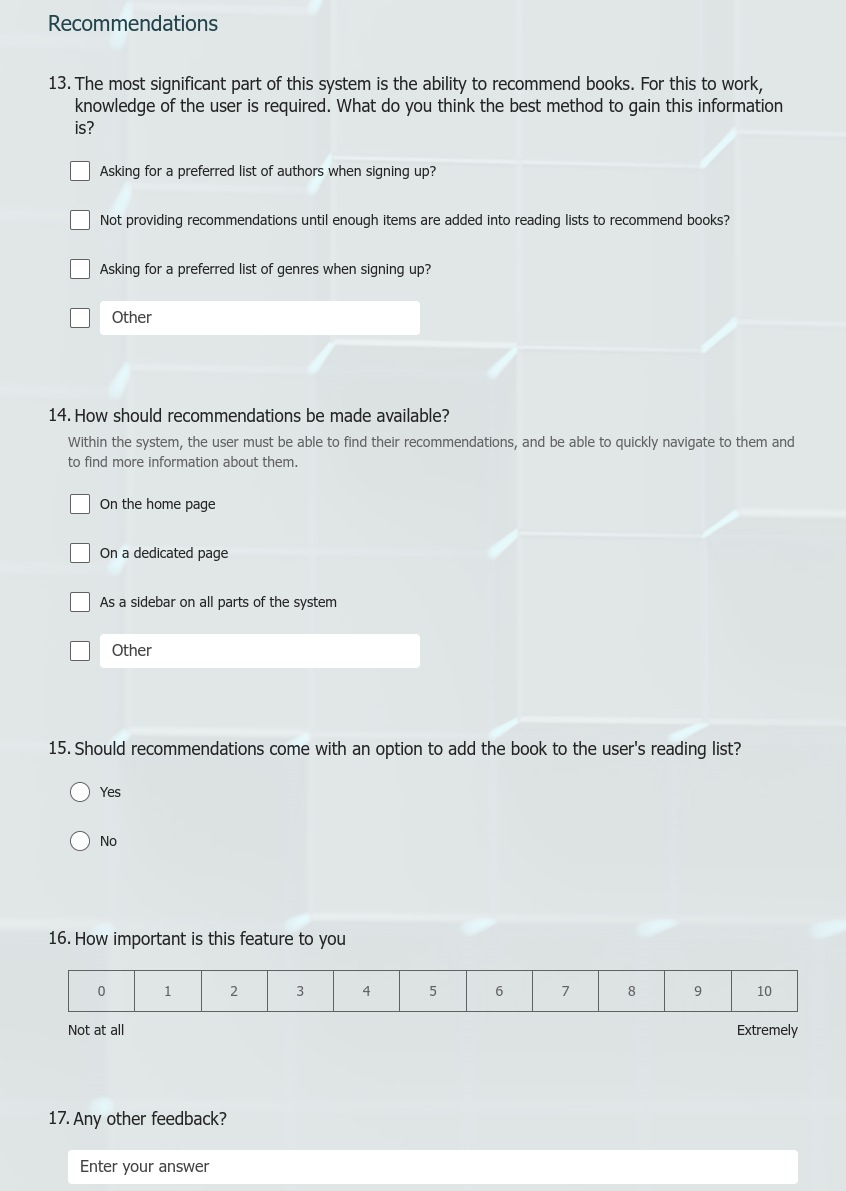
# Appendix

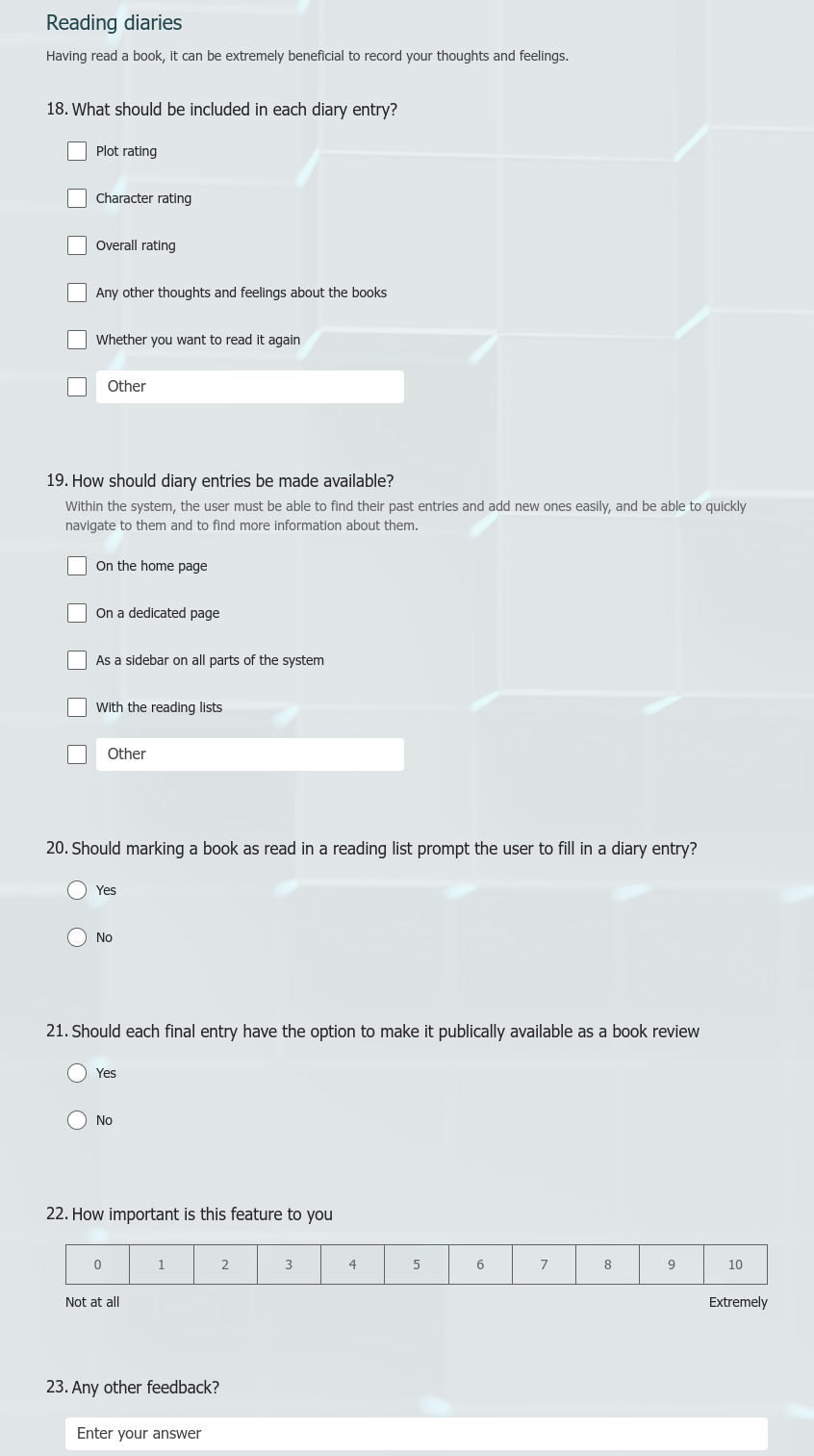
## Appendix A













## Appendix B

from math import log2, sqrt

from itertools import chain

import csv

import timeit

# Extensions only. These can be removed for simplicitly, and were included for testing

import pkg\_resources # Dependancy

from symspellpy import SymSpell # Spelling correction

from nltk.stem.snowball import SnowballStemmer # Stemming words so words such as "taking" and "takes" produce the same results

from num2words import num2words # Changing any numbers to words so searching 22, is the same as twenty two in the dataset

import contractions # Expanding contractions, such as don't to do not

stemmer = SnowballStemmer("english")

sym\_spell = SymSpell(max\_dictionary\_edit\_distance=2, prefix\_length=7)

dictionary\_path = pkg\_resources.resource\_filename(

    "symspellpy", "frequency\_dictionary\_en\_82\_765.txt"

)

bigram\_path = pkg\_resources.resource\_filename(

    "symspellpy", "frequency\_bigramdictionary\_en\_243\_342.txt"

)

sym\_spell.load\_dictionary(dictionary\_path, term\_index=0, count\_index=1)

sym\_spell.load\_bigram\_dictionary(bigram\_path, term\_index=0, count\_index=2)

class Book:

def \_\_init\_\_(self, title):

self.title = title

self.title\_word\_list = self.gen\_title\_word\_list()

self.tf = self.gen\_tf()

def gen\_title\_word\_list(self):

clean = contractions.fix(self.title)

clean = ' '.join([i if not i.isnumeric() else num2words(i).replace('-', ' ') for i in clean.split()])

clean = "".join([i if i.isalnum() else ' ' for i in clean])

clean = str(sym\_spell.lookup\_compound(clean, max\_edit\_distance=2)[0]).split(', ')[0].split()

clean = [stemmer.stem(i) for i in clean]

return clean

def title\_contains(self, word):

return word in self.title\_word\_list

def gen\_tf(self):

tf = {}

one\_over\_n = 1 / len(self.title\_word\_list)

for i in self.title\_word\_list:

if i in tf:

tf[i] += one\_over\_n

else:

tf[i] = one\_over\_n

return tf

def gen\_tf\_idf(self, idf\_data, words=None):

if words is None:

words = self.title\_word\_list

result = {i: 0 for i in words}

for i in result.keys():

if i in idf\_data and i in self.title\_word\_list:

result[i] = self.tf[i] \* idf\_data[i]

return result

class Library:

def \_\_init\_\_(self, corpus=[]):

self.books = corpus

self.update\_search\_data()

def gen\_unique\_word\_list(self):

unique\_word\_list = []

arr = list(chain(\*[i.title\_word\_list for i in self.books]))

for i in arr:

if i not in unique\_word\_list:

unique\_word\_list.append(i)

return unique\_word\_list

def num\_books\_containing(self, word):

return sum(i.title\_contains(word) for i in self.books)

def gen\_idf(self):

num\_books = len(self.books)

self.idf = {i: log2(num\_books / self.num\_books\_containing(i)) for i in self.gen\_unique\_word\_list()}

def lookup\_idf(self, words):

return {i: self.idf[i] for i in words if i in self.unique\_title\_words}

def update\_search\_data(self):

self.unique\_title\_words = self.gen\_unique\_word\_list()

self.gen\_idf()

def search(self, terms):

target = Book(terms)

num\_books = len(self.books)

search\_terms = target.title\_word\_list

target\_idf = target.gen\_tf\_idf(self.idf)

results = [0 for i in self.books]

for i in range(num\_books):

a\_total = 0

b\_total = 0

similarity = 0

book\_idf = self.books[i].gen\_tf\_idf(self.idf, search\_terms)

for k in search\_terms:

similarity += target\_idf[k] \* book\_idf[k]

a\_total += target\_idf[k] \* target\_idf[k]

b\_total += book\_idf[k] \* book\_idf[k]

if similarity > 0:

similarity /= (sqrt(a\_total) \* sqrt(b\_total))

results[i] = similarity

return sorted([(i.title, k) for i, k in zip(self.books, results) if k > 0], key=lambda x: x[1], reverse=True)

def add\_book(self, book):

self.books.append(book)

self.update\_search\_data()

## Appendix C

<https://github.com/rjames567/Book-Recommendation-Website>

## Appendix D

Test database SQL

1. This was an active decision not to fully meet this requirement. It shows all the authors in the database, but does not show all the genres. This was because the large number of genres would make this very impractical (727 in the test dataset), so only authors were shown. There were fewer authors in the test data than genres, so using authors only is more reasonable. [↑](#footnote-ref-0)
2. This was an active decision. The matrix factorisation for the recommendations and finding the book genres, gives a match to almost every genre stored in the database. Displaying all of the genres, regardless of match, could potentially give hundreds of results (with the training data, normally around 700 out of the available 727). Instead it is limited to a certain number, which is specified in the configuration. [↑](#footnote-ref-1)