ASP.NET C# Blazor Library Application

Michael Simpson (101074874)
Carleton University
COMP 4905 - Honours Project
Supervisor: Dr Lou Nel
April 25th, 2022

Abstract

This project explores the newly emerging web application framework ASP.NET C# Blazor, developed by Microsoft as a competitor to JavaScript. A library application developed with the intent of being able to track any user's book-reading habits serves as a cover to be able to the explore the different ways that a Blazor application interacts with external services such as databases and APIs, and also to explore how this framework displays and updates information to HTML pages. To be able to track different users on the application, the ASP.NET Identity library is used, alongside Google OAuth to authenticate users and provide them with an authentication token unique to them.

With learning a new web development language in mind, I thought that combining my degree in Computer Science mixed with my love for books was a great premise behind the application, and was a driving force in my determination to create a well polished product. In the end, I feel that web development with ASP.NET is much more trivial than developing through JavaScript.

Acknowledgements

I acknowledge that I did not use any copyrighted material during the development of my application. I used built-in ASP.NET libraries to create my application as well as Bootstrap for styling. Initially, I started my development using the default application that is provided by Microsoft which over time I have completely rewritten to fit my own needs. I acknowledge that the end product of my application is only code that I have written.

I would like to thank Professor Lou Nel for being my honours project supervisor, who also sparked my interest in web development.

Table of Contents

List of Figures

List of Tables

Background Information/Motivation

This application has the main goal of combining many of the ideas I have learned over my years at Carleton and applying them to learning an emerging web application framework called ASP.NET Blazor. The concept behind my project is to create a Library application to keep track of book reading habits, but this just simply serves as just a cover for the development of more intricate features and personal learning that will be happening behind the scenes. The motivation behind this project is to improve my knowledge in the following areas:

Software Development:

- Databases
- API Querying
- Authentication/Authorization
- UI Design

Personal Development:

- New Web Development Framework
- Cloud Hosting
- Git Repositories and Bash Commands

This project has been thought of to be a huge learning experience which will capture a wide range of topics that I will without a doubt be seeing in my career post-university. The impact of these topics in my project will be further described in the *Methodology* chapter, where I will be elaborating deeply upon my action-plan and how I ended up approaching each issue that arises.

What is Blazor?

Released in May 2020, Blazor is an open-source web development framework developed by Microsoft and is intended as a competitor to the JavaScript scripting language (Nilsson, 2021). It uses a blend of basic HTML and C#, and is typically used in correlation with a modified version of Model-View-Controller design pattern to achieve readability and clean functionality at the same time. With Blazor, Microsoft does not intend to block the use of JavaScript; they actually intend for users to be able to use different JavaScript APIs and libraries, providing users with a smooth experience with the execution and addition of JavaScript within their application (Guardrex, 2022).

Blazor web applications are broken down into many different components, each serving the application either on the server side or client side. Controllers, Data Types, Database Utilities and Services are all located on the server side and will contain the behind-the-scenes functionality of the application such as database queries and API calls, authentication of a user, or scripts. Blazor Pages are at the forefront of the user interface that users will interact with on the client side. Typically, when a user interacts with the Blazor Page, there will be a GET or POST request to the server with a corresponding Controller listening for these requests. The Controller will then either call services within the application to process the data, or return information to the Page as needed.

Methodology

This project has been broken down into many manageable steps with milestones that I wish to achieve, with each step taking approximately two weeks of development time. This section of my report will be broken down into many sub-chapters, giving a deeper dive into the topics, the concepts used, the challenges I faced and had to overcome, and finally any issues that may present difficulties later in development.

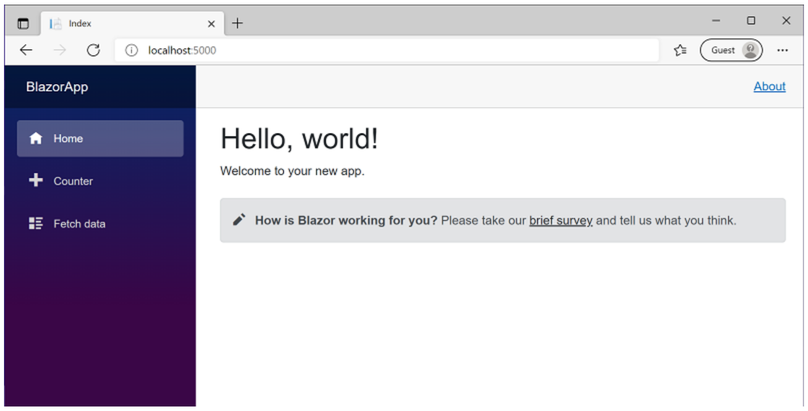
For my application, I am using the Visual Studio 2022 IDE, plus a backing postgreSQL database with the user interface pgAdmin4. To set the project up and run it for yourself, you must follow these steps:

1. Ensure visual studio is installed and updated
2. Open the **LibraryApplication.sln** file when prompted to open a project
3. Install pgAdmin4
4. Create a database entitled **Library**, with User Id **postgres** and password **postgres** on port **5433**
5. Run the program

Chapter 1: Setting up the Project

Default Application

Setting up a default application in Blazor is quite a trivial feat. Microsoft supplies a tutorial to get any user started, setting them up with a templated application which not only looks nice (using HTML bootstrap), but also functions smoothly. Within this application, there are a few pages that serve as tutorials for different methods you can use to model your own pages. The *Counter* page demonstrates the ability to write your own scripts within a Blazor Page to operate on the client's side, and the *Fetch Data* **[1]** page demonstrates a call to a service on the server side. In my application a majority of user functions will require calls to services, which I will model after the *Fetch Data* page's contents.



Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

Figure 1: The default Blazor webpage

Github Version Control

The setup of a Git repo is a very trivial process involving just a few command line commands and an initial commit to push your starter code to Github. As I am familiar with this setup, I want to learn more about the version control aspect where I will be able to review my code changes before committing them to a master branch. Before starting this project, I created a working flowchart of the process that I wish to follow throughout my development process.

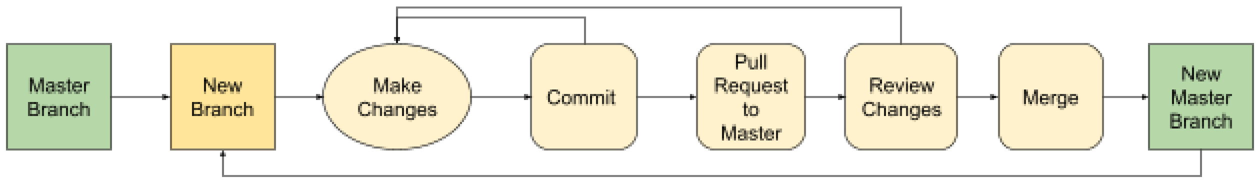


Figure 2: Flowchart of my Github software development process

When developing a new issue or topic I will first create a branch, titled after what is being worked on, off of the master branch. This is how I will mimic version control and will ensure that if any bugs arise they will be isolated to only my current branch and not the entire project. After making changes on the current branch, I will commit them with a meaningful message to be able to refer back to if needed. Once I am confident that I have completed an entire issue, a Pull Request will be created, where I will be able to look over all of my changes and review them with a precise and rigorous fashion to ensure that I will not be introducing any new bugs and that I have not left anything out of my code. If I am satisfied with my changes, I will merge them to my master branch and repeat this same process.

Chapter 2: Displaying and Caching book results

API Querying

Learning how to write in the Blazor C# framework turned out to be a quite similar process to writing standard Java in the ways of syntax and of class structure, and since I have ample experience with Java, this learning curve was much more intuitive. To start my learning process, I looked through all of the different pages the default application gives and made quick notes as to what they are attempting to accomplish and how. The different functionalities have been broken down in the *Default Application* subchapter above.

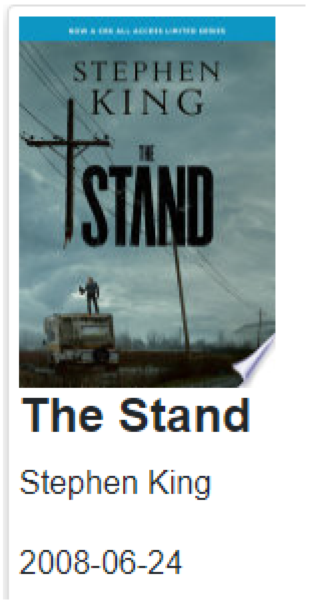
As a proof of concept, I chose my first task to be the implementation of API querying to *books.google.com*. I chose this because my project will be heavily reliant on a good implementation of this, and getting it done at the start and learning the inner workings of API calls will set myself up for success during future development. The working concept that I came up with to be able to store this data is to break the process up into a *SearchView* page, a static *BookService* class and a *BookData* object class. The *SearchView* page will allow users to search using a query bar, sending the request to the backend *BookService*, and will show the results using HTML cards. The *BookService* class will make the API call to *books.google.com*, parsing and storing the data in a list of *BookData* objects.

After the creation of a basic *SearchView* page, I remodeled the backend calling code found in the default "FetchData" page to receive a list of *BookData* objects. I created a basic GET request formed from the API user manual provided by Google (Google, 2022), and in return I received a massive string object full of JSON. Parsing this data was a relatively simple process: after deserializing the string, I was left with a dictionary with many dimensions.

Looking at the *items* entry gave me the results showing each of the different books and their corresponding data - which was slotted into its own *BookData* object and promptly added to the list to return to the *SearchView* page. This process completes my proof of concept and sets the base requirement for my web application to be able to function as I have imagined.

Displaying BookData to Users

Now that I have these objects that hold book information from *books.google.com*, I need to be able to display them to the user. After some research, I found that HTML cards were the most neat and the style is very simple to understand for the average user. I created a mock *BookData* object, and attempted to display this image in a static way.



The Stand

Stephen King

2008-06-24

Figure 3: The first HTML card for my test book

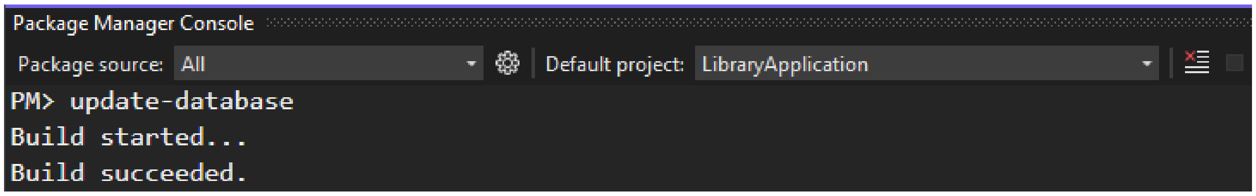
An extremely helpful function of ASP.NET Blazor is how their HTML pages are crafted. Not only can a user use the typical HTML and JavaScript tags and functions to create their pages, but you can also use C# to implement logic and functions within the pages too which heavily simplifies interactions with controller classes. With a simple *foreach* loop, I added a way for each book to have its own card created. With this underlying functionality done, I created a way for users to search for the title of a book, sent that as a GET request to the *BookService*, and upon receival of the list of books, the page was updated with each different book being displayed properly.

However, an issue arose instantly as I saw that the *books.google.com* API will only return a maximum of 10 books per API call. In real-world search applications such as online grocery stores or services, users are typically presented with massive lists of results and the ability to show the *next page* of results. Seeing as pagination is a very used concept in any service with a search feature, I wanted to find out a way to have it in my own application, and with only 10 search results at a time this was infeasible. A solution that I came up with to fix this issue was to keep my own cache of books in my database and figure out a way in the future of development to be able to use this local cache to create my own search algorithm which would return more than 10 results. This was the first major issue that I ran into during the development of my web application, and since I had the basic functionality of the page completed already, I left this as a task to be completed towards the end of my development process as I did not want this to be a big roadblock for time.

Caching Book Results

To be able to keep a cache of the results returned from the Google API, a backing database would be needed for the application. Seeing as most of my experience with databases is in terms of postgreSQL, and how ASP.NET promotes the use of a postgres database with relative ease (Chanphakeo, 2017), this seemed the obvious choice for my application. To initialize a database, it must be registered as a context within the main *Program.cs* class with a corresponding database context class that will be filled out with the entries from the tables upon startup of the application. This context class provides an easy way to access entries from database tables from anywhere within the web application, however, in the context of good practice I will be using the *Service* class-type (for example: *BookService*) to isolate all database interactions within a single class. This will ensure that if a change is needed only one class will need to be updated.

Once the database has been registered within the *Program.cs* class, the next step is to build the database. To do this, a user must use database migration commands within the Package Manager Console on the Visual Studio IDE. These commands will create new migrations classes which will run upon startup of the application and will either create, update, or drop tables within the database as required.



```
PM> update-database
Build started...
Build succeeded.
```

Figure 4: Example of the update-database command in the Package Manager Console in VS2022

Returning to how I wanted to cache the query results from Google in my database, I added the books table using the primary key of a book's International Standard Book Number (ISBN), which later had to be amended to use the primary key of an increasing integer due to some results from the API not being registered with an ISBN. Helper functions were necessary within the *BookService* class that handle the addition or deletion of book entries within the database; the addition function would be called after checking if the book is already found within the database. Since the design of database migrations within ASP.NET directly models tables after how objects are defined, adding a new object to a database can be done in as little as two lines. The first adds the object to the list of entry-objects in the database-context class, and the second will asynchronously save the changes to the database.

```
database.Add(object);
await database.SaveChangesAsync();
```

Figure 5: Example pseudocode of adding to a database

Chapter 3: Authentication

The Identity Framework

The implementation of authentication within a web application is a very intricate process, with many small parts that must be working flawlessly to function properly. Fortunately, there is a built-in Identity framework within ASP.NET Blazor that covers a significant portion of these intricate parts, requiring only the development of an additional layer of functions tuned to fit the developers own needs (Galloway, 2022). This isn't to say that it is quick to implement authentication, it is still a very lengthy process and full of challenges that I needed to overcome. The Identity framework consists of two primary classes that users will need to add to their application: *UserManager* and *SignInManager*. UserManager is the class that has functions to be able to register new users, validate user credentials and that will load user information (Foster, 2014). Manager allows users to be signed into the application, and will store a cookie on the authenticated user's browser, which will be deleted upon logout (Venkat, 2019).

I started this task with the initial intent of adding a way for users to be able to create their own account by entering a username and password, and then proceeding to be able to sign in by providing those details. Following the creation of the Blazor pages with textboxes for users to be able to register and login to their account, I then added the Identity service to *Program.cs* and called the same migration commands in the Package Manager Console to build the database that backs the Identity framework. With the same process in mind as the *BookService*, I created the *AccountServiceController* class which will contain the backing functions to register, sign a user in, and sign a user out of the application using a combination of the UserManager and SignInManager classes. The reason behind the inheritance of the *Controller* class is due to how this service is not just a way for a user to interact with a database - those interactions are done through the two Identity classes. This class is used to create the connection between these two services, thus providing the backend handling of the tasks for authentication.

At first, this process was going smoothly; registering new users was working properly and the user was correctly being added to the database. However, I ran into a roadblock where SignInManager was seemingly matching a user's provided credentials with those provided from a database profile in an accurate fashion, but it was not administering the authentication cookie on redirect. I worked on this issue for a few days, changing how cookies were configured and their policies, changing the implementation of signing in and how I redirected users but this seemingly had no effect. After these days of editing and changing how authentication worked in my program, I had wound up in a scrambled mess that was seemingly unsalvageable had I not been practicing Github version control since the beginning of development. Working in separate branches and pushing to main only once I was satisfied with my progress allowed me to relievingly revert all of my changes to the commit prior to running into this roadblock, saving my project.

Google OAuth

The cookie distribution issue through SignInManager forced the need to think of a completely different method of being able to authenticate users, and although in my project proposal I had listed this as a *if time permitted*, implementing Google OAuth seemed more like a necessity. Similar to the services that Facebook, Apple or Amazon provide, OAuth is a way for users to be able to log into a service without a username and password. Instead, the web application will redirect the user to a chosen external login provider, whether that be Google, Facebook or others, and upon logging in through that service will redirect the user back to the original application through an external callback function. If the user has been properly authenticated through the external login provider, they will be considered as an authenticated user and will be issued a valid cookie accordingly. This process effectively removes the need for any web application using OAuth to store a user's password, thus creating a safer environment (Pearson, 2020).

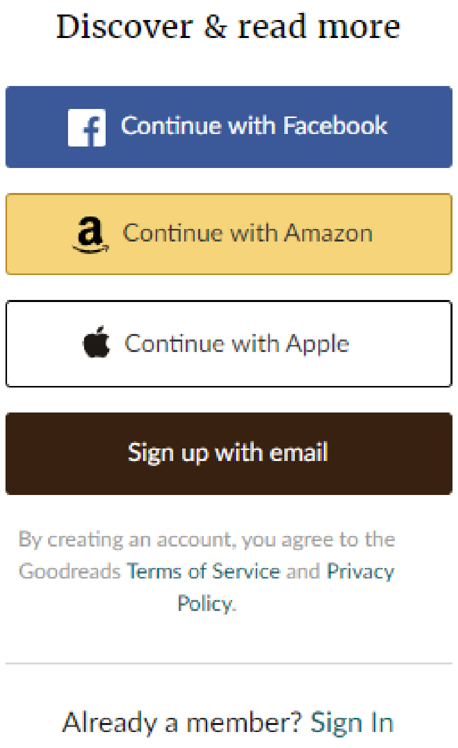
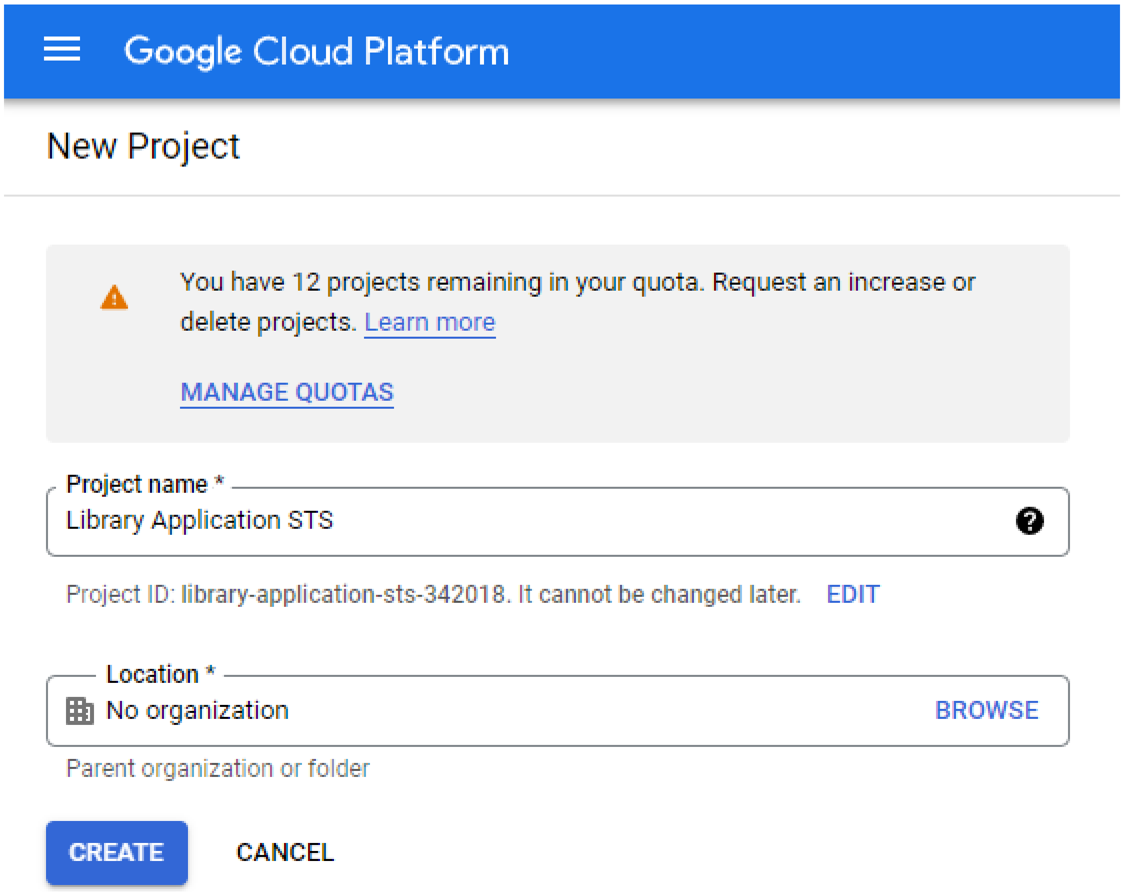

Discover & read more
Continue with Facebook
Continue with Amazon
Continue with Apple
Sign up with email
By creating an account, you agree to the Goodreads Terms of Service and Privacy Policy.
Already a member? Sign In

Figure 6: Example of OAuth login methods (Goodreads, 2022)

To begin the process of implementing Google OAuth, users must be granted a *ClientId* and a *ClientSecret*, which are used as a way to verify to Google that the application requesting a user to be authenticated is coming from a source that had been registered through them. These values are easily required by going through the Google Cloud console, navigating to the OAuth Consent page and creating a new project. On the free version, Google will grant up to 10,000 access tokens which are used in the external callback function to authenticate users, and a maximum of 100 test users when your application is listed as in the testing environment. For my application, these values are excessive. Yet for applications that will be published to the public they have the potential to max out quickly.



You have 12 projects remaining in your quota. Request an increase or delete projects. Learn more

Project name *
Library Application STS

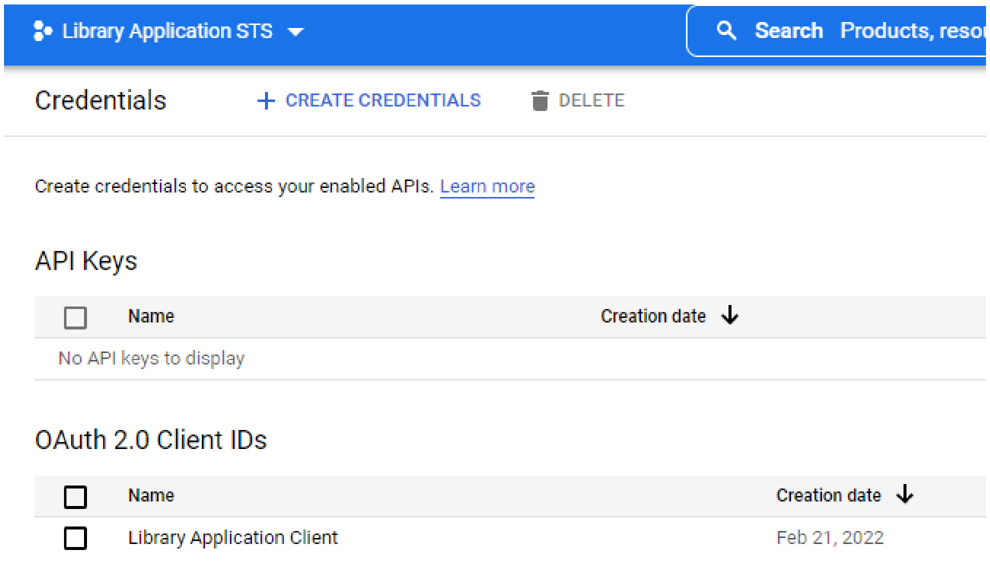
Project ID: library-application-sts-342018. It cannot be changed later. EDIT

Location *
No organization BROWSE

Parent organization or folder

Figure 7: Creating a project through Google

After the creation of a project within the Google Cloud, the next step is to navigate to the Credentials page and create your OAuth 2.0 Client Id which will grant a user their *ClientId* and *ClientSecret*. These values are used to configure the Google authentication service within the *Program.cs* class, and will be securely sent to Google in the OAuth request.



Create credentials to access your enabled APIs. Learn more

API Keys

	Name	Creation date ↓
	No API keys to display	

OAuth 2.0 Client IDs

	Name	Creation date ↓
☐	Library Application Client	Feb 21, 2022

Figure 8: Creating a OAuth Client Id to gain access to my application's Security Token System (STS)

ASP.NET Identity proves yet again that they truly have the end-user in mind when they created their authentication framework because the *ExternalLogin* functions are very simply set up, and contain objects specifically created to make the process of redirection and token management especially easy. All that is required in the first external login function is to create a redirect URL and send that as a *ChallengeRequest* to Google. In general, an authentication challenge is issued when an unauthenticated user requests access to an endpoint that requires authentication (Rousos, 2022), and in the case of OAuth the user is trying to become authenticated so a challenge is sent to the Google OAuth API.

Upon successful or unsuccessful authentication, Google will redirect the user back through the redirect URL, and the application will then handle the result of the OAuth process by means of the *ExternalLoginCallback* function. The flow of my implementation of this function is as visualized in **Fig. 9**. Unless an error has occurred, the function will attempt to sign the user in and will create an account for a new user if they are not currently in the user database already.

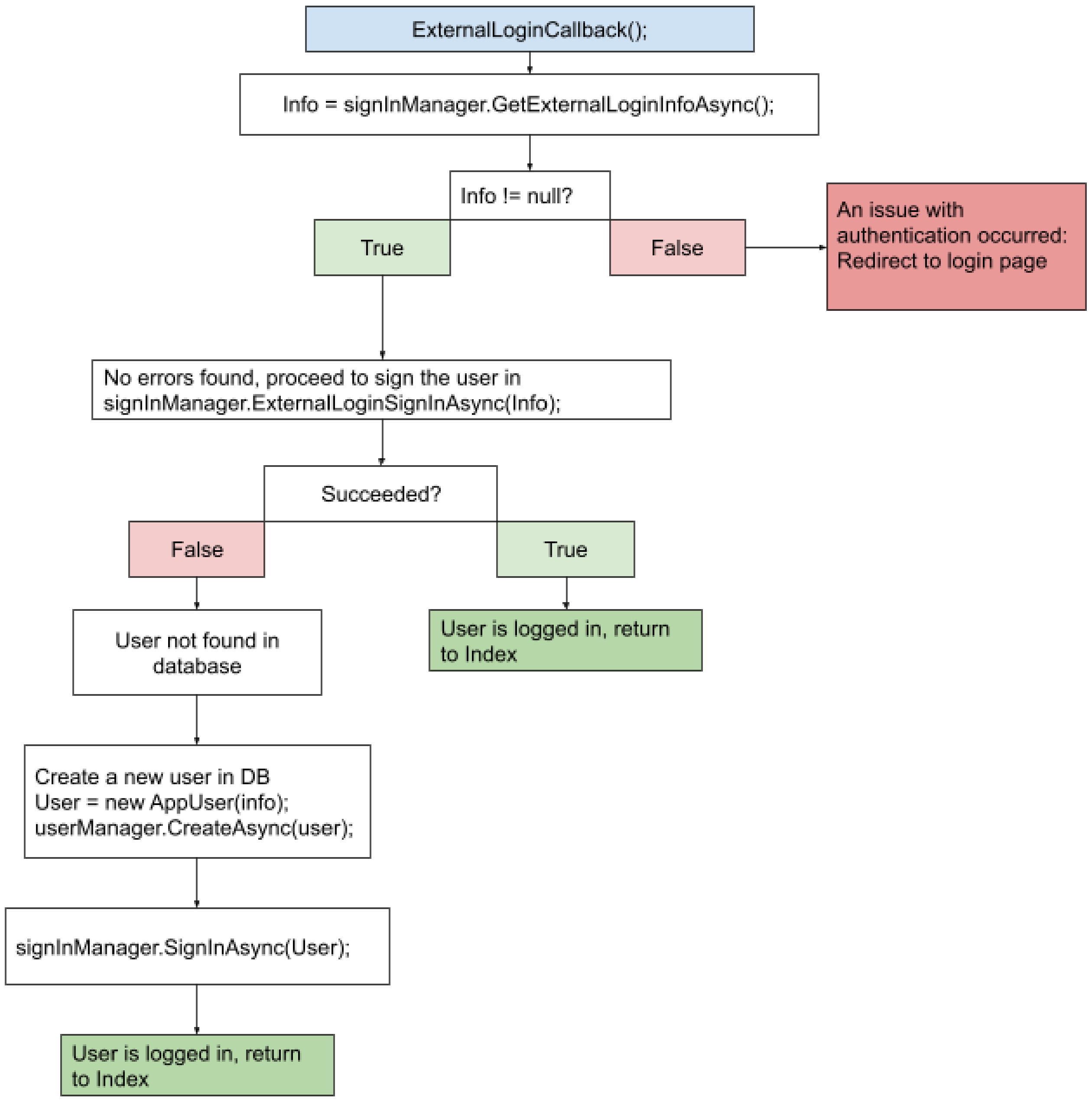

ExternalLoginCallback();
Info = signInManager.GetExternalLoginInfoAsync();
Info != null?
True
False
An issue with authentication occurred: Redirect to login page
No errors found, proceed to sign the user in signInManager.ExternalLoginSignInAsync(Info);
Succeeded?
False
True
User not found in database
User is logged in, return to Index
Create a new user in DB User = new AppUser(info); userManager.CreateAsync(user);
signInManager.SignInAsync(User);
User is logged in, return to Index

Figure 9: The flow of the ExternalLoginCallback function

The flow of this process, as compared to the flow of signing in manually with a username and password is a lot more trivial and easier to manage. It also happens to resolve the earlier issue where SignInManager was not properly distributing the authenticated cookie to a user who signed in with their credentials. With OAuth implemented, authentication for logging into my web application has been completed. Logging out is a process that is also very trivial and cookie management is managed by the SignInManager as well, through the function

SignOutAsync(). To display to the user that they have been logged in, or are currently not logged in, the bottom of the navigation bar on the left of the application will either display their username or the logout button.

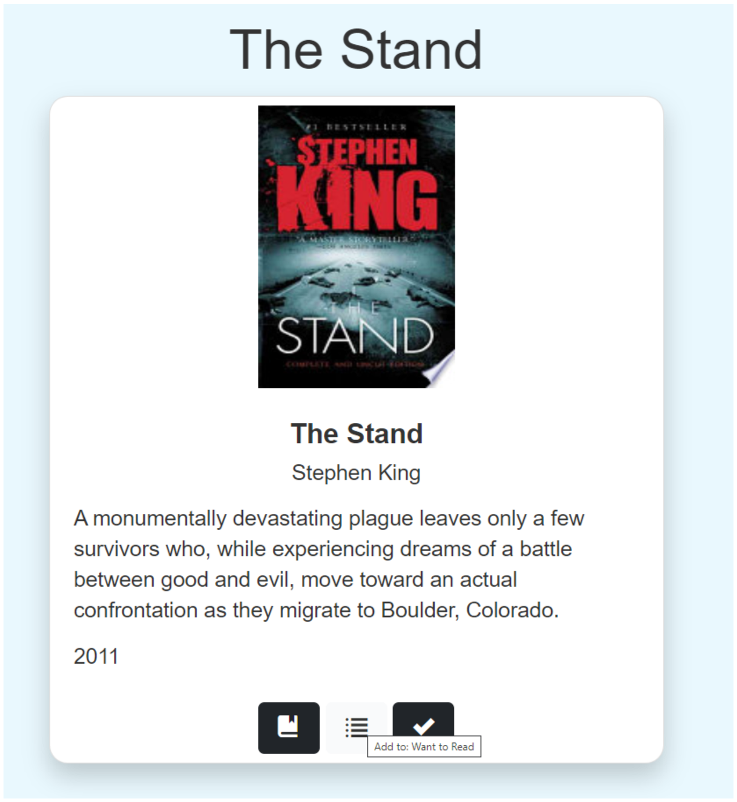
Chapter 4: User Home Pages

Storing Books for a User

To store different books for each individual user that will use the application, the database must be updated to hold new values. In my plan for the application, I defined three separate lists that I would like to address: books that the user is currently reading, books that they would like to read, and books that they have read already. The homepage will be composed of these three bookshelves which will be used as a way for users to track their book reading habits. To update the database to store these lists, I needed to create a new class that inherits from the generic ASP.NET built-in *IdentityUser* class. Doing this ensured that no previous work I had done with authentication would be affected, as *IdentityUser* class objects are important to the core of the SignInManager and UserManager classes. In my *AppUser* class that I created to inherit from *IdentityUser*, I added a string variable for each of the different lists that would be stored in the database. I chose strings to store these lists because I am the most familiar with working with strings in a database rather than lists of objects, so parsing and creating lists out of those strings was the most intuitive to do.

Once I was satisfied with the class, I created a new database migration in the Package Manager Console and updated the database to create new columns in the AspNetUser table. Next, I created some helper functions in the *BookService* class to be able to parse a string of book id's and return a list of *BookData* objects filled out from the values in the string. Earlier when I decided to store the books within the application, I chose the primary key to be an increasing integer; so I chose those integers as the way to identify which book would be stored in a users list as well. I also created helper functions in the *BookService* class to be able to append and remove individual book id's from a string, using commas as delimiters.

In regards to how any user will actually be able to add or remove books from their corresponding lists, I created a new page within my application for users to be able to see an individual book's details. Within this page, I implemented buttons for adding and removing from each of the different lists and called upon the *BookService* class to asynchronously update the database for each change.


The Stand

The Stand

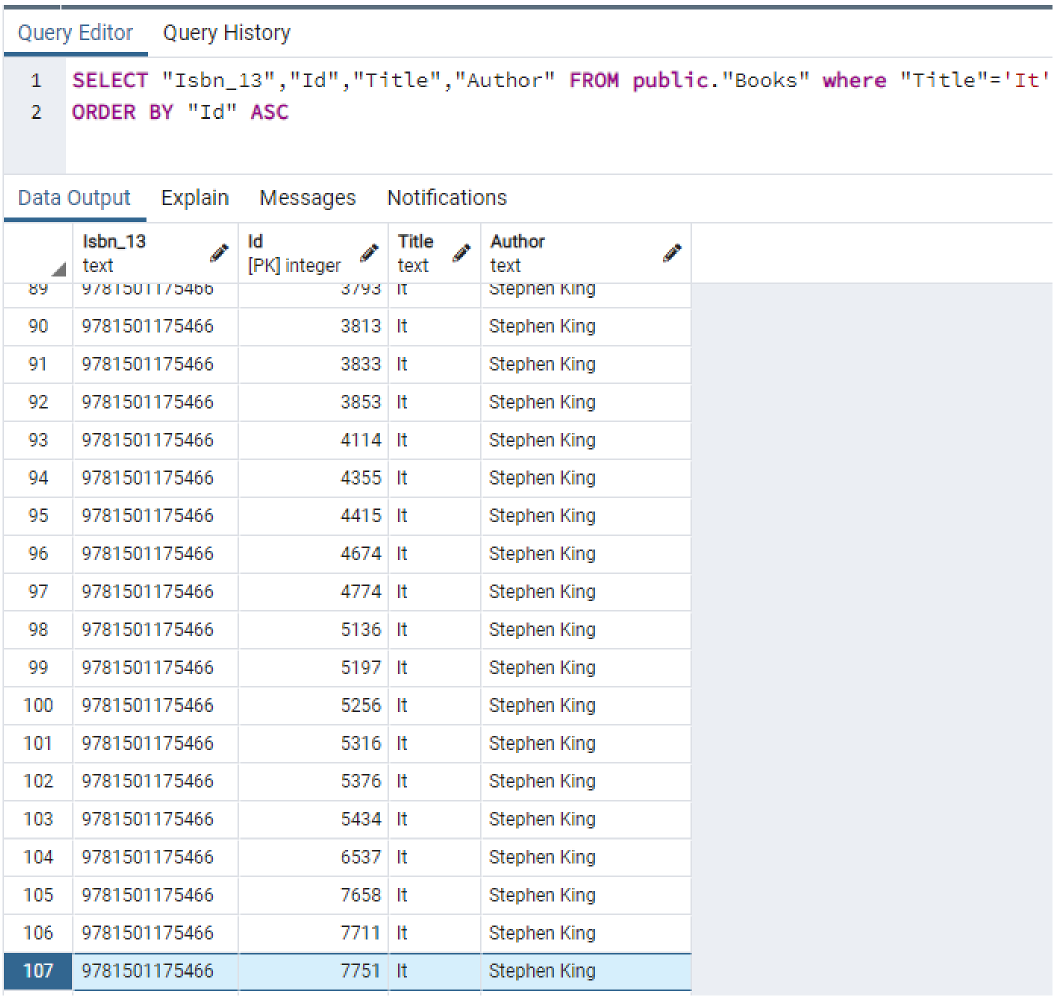
Stephen King

A monumentally devastating plague leaves only a few survivors who, while experiencing dreams of a battle between good and evil, move toward an actual confrontation as they migrate to Boulder, Colorado.

2011

Figure 10: Example of how a user will add a book to a list

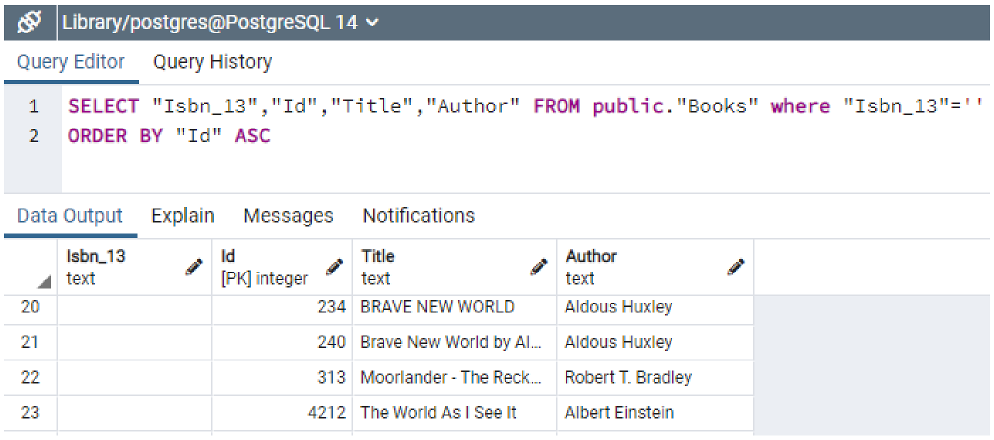
Testing these changes brought out an extremely big memory leak issue that was created at the very beginning of the application when I was adding new books to the database. The issue stems from the switch from using a book's ISBN number as the primary key for storing books in the database, to just using the incrementing integer. When checking to see if a book is already added into the book table, I was comparing the ISBN of a book from the Google API query to the *primary key* of the books table, which had been changed since the creation of that code. This means that throughout all of my development process the exact same books have been continuously re-entered into my database. My solution to this issue is not a perfect one: I decided to compare the ISBN of the query from the API to the ISBN that I store in the database. The reason this is not perfect is because in theory all books should have an ISBN for proper identification, but not all entries have an ISBN in the Google API. Some of these entries that I have come across are for extremely famous books such as *Brave New World* by Aldous Huxley, or *The World As I See It* by Albert Einstein, and why they don't appear to have ISBNs in the API is an uncertainty.



```
SELECT "Isbn_13","Id","Title","Author" FROM public."Books" where "Title"='It'
ORDER BY "Id" ASC
```

	Isbn_13 text	Id [PK] integer	Title text	Author text
89	9781501175466	3793	It	Stephen King
90	9781501175466	3813	It	Stephen King
91	9781501175466	3833	It	Stephen King
92	9781501175466	3853	It	Stephen King
93	9781501175466	4114	It	Stephen King
94	9781501175466	4355	It	Stephen King
95	9781501175466	4415	It	Stephen King
96	9781501175466	4674	It	Stephen King
97	9781501175466	4774	It	Stephen King
98	9781501175466	5136	It	Stephen King
99	9781501175466	5197	It	Stephen King
100	9781501175466	5256	It	Stephen King
101	9781501175466	5316	It	Stephen King
102	9781501175466	5376	It	Stephen King
103	9781501175466	5434	It	Stephen King
104	9781501175466	6537	It	Stephen King
105	9781501175466	7658	It	Stephen King
106	9781501175466	7711	It	Stephen King
107	9781501175466	7751	It	Stephen King

Figure 11: Querying the database which shows 107 entries for the same book


Library/postgres@PostgreSQL 14

Query Editor | Query History

```
SELECT "Isbn_13","Id","Title","Author" FROM public."Books" where "Isbn_13"=''
ORDER BY "Id" ASC
```

Data Output | Explain | Messages | Notifications

	Isbn_13 text	Id [PK] integer	Title text	Author text
20		234	BRAVE NEW WORLD	Aldous Huxley
21		240	Brave New World by Al...	Aldous Huxley
22		313	Moorlander - The Reck...	Robert T. Bradley
23		4212	The World As I See It	Albert Einstein

Figure 12: Books that don’t appear to have ISBNs according to the Google API.

Creating Individual Homepages

The homepage of the library application is a core component for a user’s experience on the application. This page will be used to showcase the different bookshelves that any user may have, and displaying the books in a proper fashion is important. To do this, I reused some of the code I wrote on the search page to be able to present a dynamic number of books in a neat way. The different bookshelves that I have used for each user consist of the books they add to their currently reading, want to read, and already read lists. This page makes use of the helper functions in *BookService* that will directly grab a list of *BookData* objects from the database based on the string lists supplied by the authenticated user’s profile. With these lists, depending on if the size of them is greater than zero, I will show the corresponding shelf. Since these shelves are unique to each individual user, no two homepages will be the same unless two users share the exact same book lists.

Chapter 5: UI Cleanup

Bootstrap

Up until this point in the project, all features on the user interface were added in the interest of getting functionality done in the back end, and providing minimal effort in the beautification of my project. However, cleanup was necessary to ensure that the application is usable and would make users want to return back in the future. The HTML library that I am most familiar with is Bootstrap, which is an open-source CSS framework that has the goal of enhancing front-end UI development (Otto, 2022). Upon first importing the bootstrap stylesheet, the look at the library application already changed drastically, but there were still many issues that needed to be addressed. The main issues that I faced were: books were not being displayed with the proper Bootstrap tags, buttons were looking very outdated, text sizes were lacking a basic structure from page to page and finally everything that was not the navigation bar was coloured white.

Taking this a step at a time, I first addressed how books were being displayed. I updated the HTML tags to incorporate Bootstrap, and during this process decided that the application as a whole will take on a light-bluish colour palette. I changed the buttons on the page to be styled with Bootstrap tags and did the same for text and captions on all pages. As a last step in the restyleing of my application, I changed the colour of the navigation bar and the background to a light blue. Adding these changes turned my application from a default, unmemorable application into an application that has its own personality.

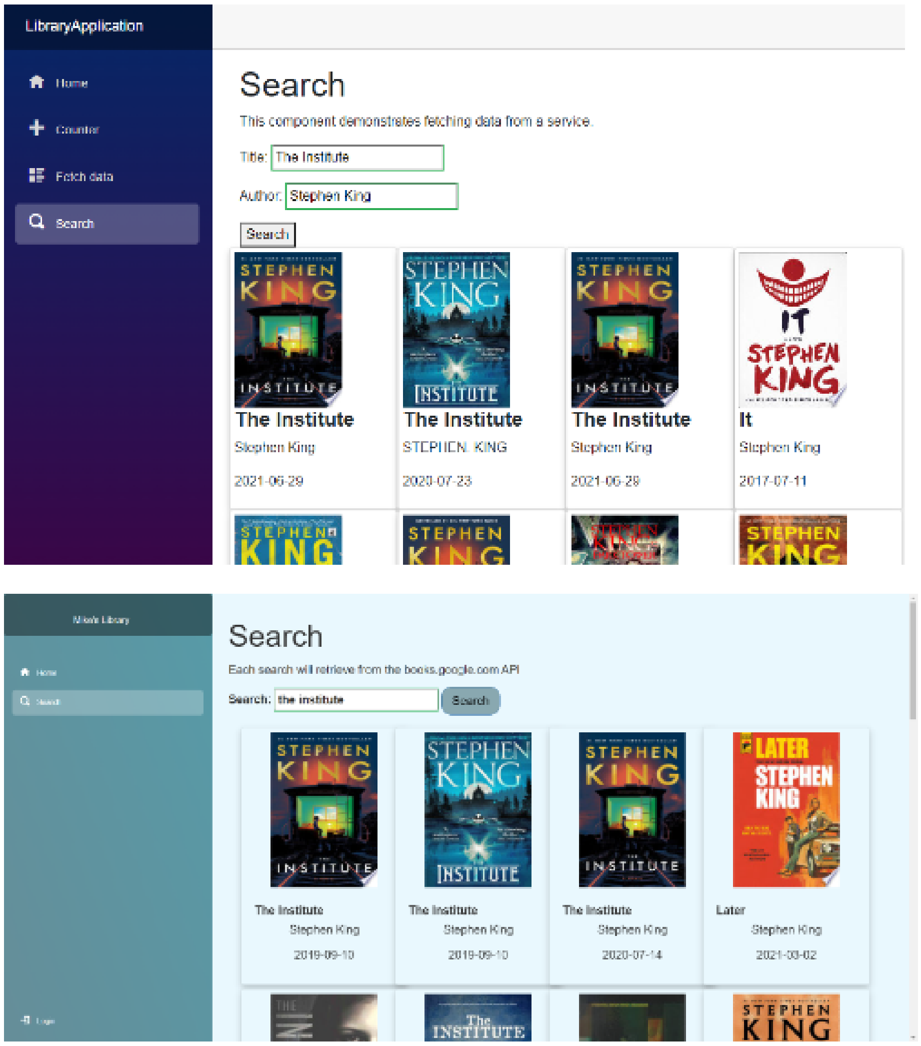


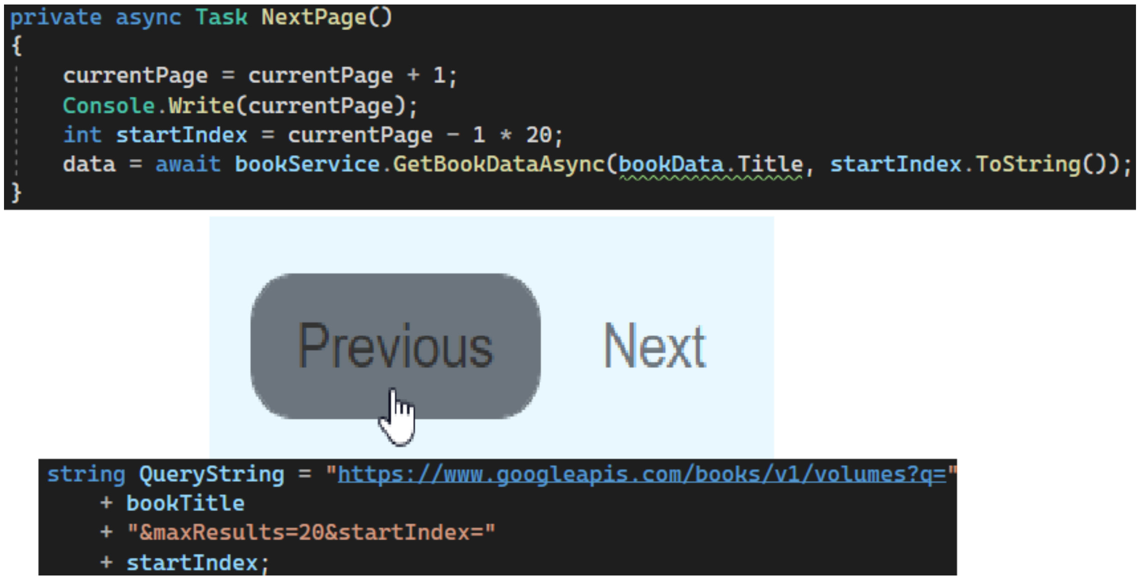
Figure 13: Before and after UI comparison

Pagination

In the *Displaying BookData to Users* section of this report there was an issue pertaining to how the maximum number of books returned from the Google Books API was maximized to 10, and since I have now been cleaning up the UI, this issue has become more prominent. After researching the different ways that I could get beyond this limitation, I found that the actual maximum books per request is 40, found by appending the keyword *maxResults=40* to the query string. Another query tag I came across was *startIndex*, where you can choose the start index of the search query. This start index key sparked the idea of adding pagination by incrementing the start index by a multiple of how many search results are being grabbed per page. Deriving a formula for this calculation, the start index can be calculated by first decrementing the current page (due to the starting index being 0) and multiplying that by the maximum results per page. Formally, the equation is:

$$startIndex = (currentPage - 1) * maxResults$$

With this in mind, I updated the get request in the search page to also send a value for the current page to the controller, and updated the query string with the calculated start index. To be able to change the current page of books on the search page, I added *Previous* and *Next* buttons, which will decrement or increment the current page integer and proceed to send the get request which updates the books that are displayed on the page. With this challenge completed, and achieving my goal of implementing pagination, the library application has been completed with all core functionality that I wished to develop during the course of my honours project.



```
private async Task NextPage()
{
    currentPage = currentPage + 1;
    Console.Write(currentPage);
    int startIndex = currentPage - 1 * 20;
    data = await bookService.GetBookDataAsync(bookData.Title, startIndex.ToString());
}
```

```
string QueryString = "https://www.googleapis.com/books/v1/volumes?q="
    + bookTitle
    + "&maxResults=20&startIndex="
    + startIndex;
```

Figure 14: The NextPage() function and pagination

Chapter 6: Additional Features

During the development of my web application, two features came to mind that I wished to give an attempt at implementing: adding a way for users to give a review for a book, and hosting the application using a cloud hosting service. These two features were not originally mentioned in my honours project proposal, but seeing as I had a bit of time toward the end of the semester I decided that I should see how far I could get for these features.

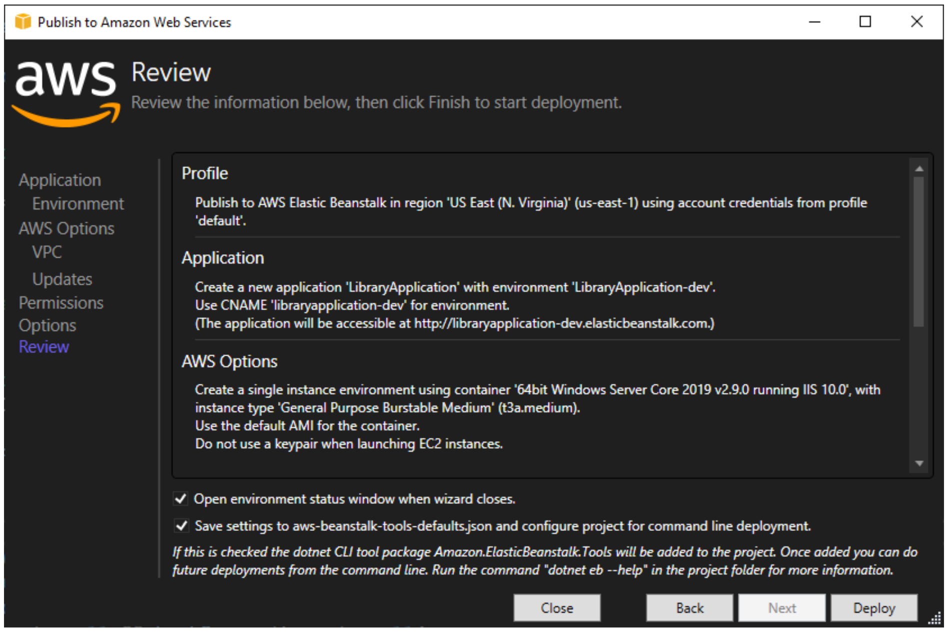
Book Reviews

Using the knowledge that I have gained through working with databases, migrations, objects and page creation all throughout this project, the addition of book reviews was completed in record time. The plan for this addition was as follows: first, create *ReviewData* and *ReviewDbContext* classes to be able to update the database to store a brand new *Reviews* table. Second, use database migrations to update the postgres database with this new table and ensure that upon startup the context class is being filled out properly. Third, create the functionality within the *BookService* class to be able to add and get different reviews from the database, and finally, add a section to the individual book pages that will grab reviews from the database, show them to the user, and add a way for authenticated users to be able to add their own review for the book. Following this plan worked flawlessly, and now any authenticated user will be able to leave their own review on a book and see other user's reviews.

Cloud Hosting

Cloud hosting was an extremely confusing and difficult challenge to include for this project. I chose to use Amazon Web Services' Elastic Beanstalk for my project because in a previous CO-OP internship I was introduced to the topic of cloud hosting using this service.

After creating an account and registering my web application under its own domain through AWS, I installed the Visual Studio plugin that helps mediate deployments and greatly simplifies changing different deployment settings within the cloud. This plugin also presents a console interface showing the status of the deployment, and if needed will pull a list of logs from the service for debugging.



Review

Review the information below, then click Finish to start deployment.

Profile

Publish to AWS Elastic Beanstalk in region 'US East (N. Virginia)' (us-east-1) using account credentials from profile 'default'.

Application

Create a new application 'LibraryApplication' with environment 'LibraryApplication-dev'.
Use CNAME 'libraryapplication-dev' for environment.
(The application will be accessible at http://libraryapplication-dev.elasticbeanstalk.com.)

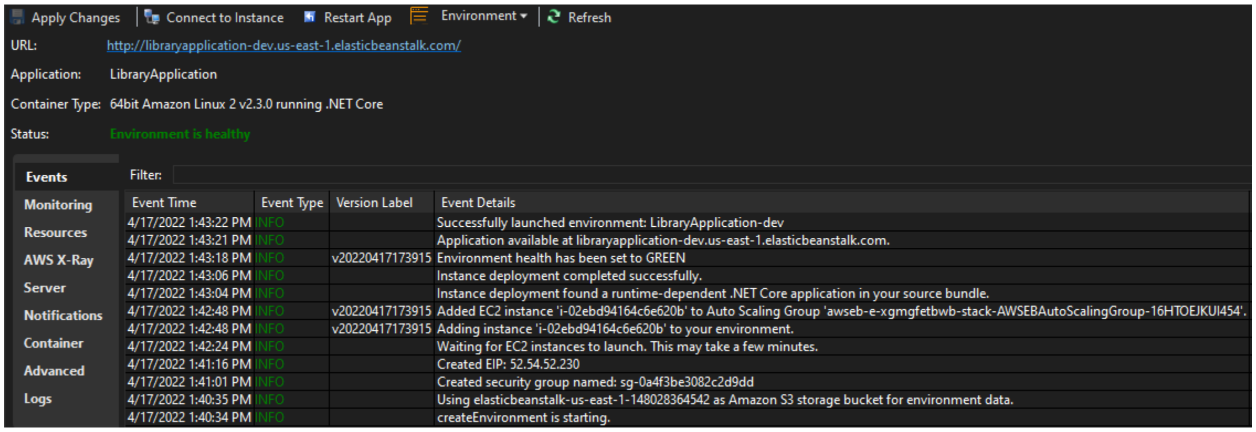
AWS Options

Create a single instance environment using container '64bit Windows Server Core 2019 v2.9.0 running IIS 10.0', with instance type 'General Purpose Burstable Medium' (t3a.medium).
Use the default AMI for the container.
Do not use a keypair when launching EC2 instances.

✓ Open environment status window when wizard closes.

✓ Save settings to aws-beanstalk-tools-defaults.json and configure project for command line deployment.

If this is checked the dotnet CLI tool package Amazon.ElasticBeanstalk.Tools will be added to the project. Once added you can do future deployments from the command line. Run the command "dotnet eb --help" in the project folder for more information.



URL: http://libraryapplication-dev.us-east-1.elasticbeanstalk.com/

Application: LibraryApplication

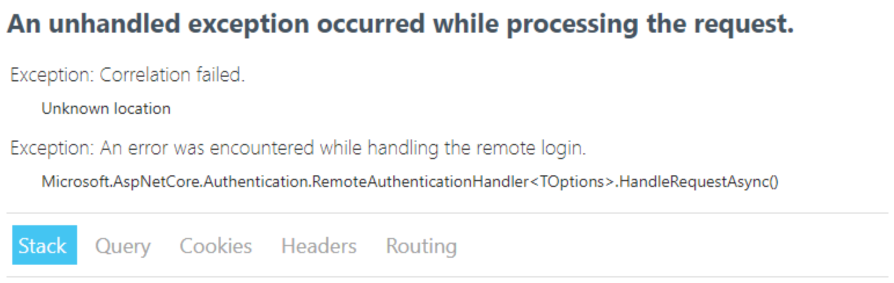
Container Type: 64bit Amazon Linux 2 v2.3.0 running .NET Core

Status: Environment is healthy

Event Time	Event Type	Version Label	Event Details
4/17/2022 1:43:22 PM	INFO		Successfully launched environment: LibraryApplication-dev
4/17/2022 1:43:21 PM	INFO		Application available at libraryapplication-dev.us-east-1.elasticbeanstalk.com.
4/17/2022 1:43:18 PM	INFO	v20220417173915	Environment health has been set to GREEN
4/17/2022 1:43:06 PM	INFO		Instance deployment completed successfully.
4/17/2022 1:43:04 PM	INFO		Instance deployment found a runtime-dependent .NET Core application in your source bundle.
4/17/2022 1:42:48 PM	INFO	v20220417173915	Added EC2 instance 'i-02ebd94164c6e620b' to Auto Scaling Group 'awseb-e-xgmgfetbwb-stack-AWSEBAutoScalingGroup-16HTOEJKUI454'.
4/17/2022 1:42:48 PM	INFO	v20220417173915	Adding instance 'i-02ebd94164c6e620b' to your environment.
4/17/2022 1:42:24 PM	INFO		Waiting for EC2 instances to launch. This may take a few minutes.
4/17/2022 1:41:16 PM	INFO		Created EIP: 52.54.52.230
4/17/2022 1:41:01 PM	INFO		Created security group named: sg-0a4f3be3082c2d9dd
4/17/2022 1:40:35 PM	INFO		Using elasticbeanstalk-us-east-1-148028364542 as Amazon S3 storage bucket for environment data.
4/17/2022 1:40:34 PM	INFO		createEnvironment is starting.

Figure 15: Showcasing the AWS plugin on Visual Studio, plus the debugging console

The deployment process of the application to the cloud went by smoothly, and upon first navigating to the website everything seemed perfect, that is until I tried to log in or search for books. When logging in, Google OAuth presented an error saying that the current web application has not been registered through the service as being a trusted URL. Fixing this was simple, all that needed to be done was add the URL that I was hosting the service on to the API. Now, after attempting to log in, I was presented with the usual log in screen, but after being redirected back to the library application I was presented with a correlation error, presumably because I had not yet set the postgres database up. The lack of database appeared to be the same issue with the search page not working, and when attempting to set the database up I was running into an issue where the tables were not being created properly, and although my connection string to the database seemed correct, nothing was changing. I ended up having to scrap this AWS branch in the interest of time, however for future work for this web application, it would be mandatory to set up cloud hosting to take my project to the next level.



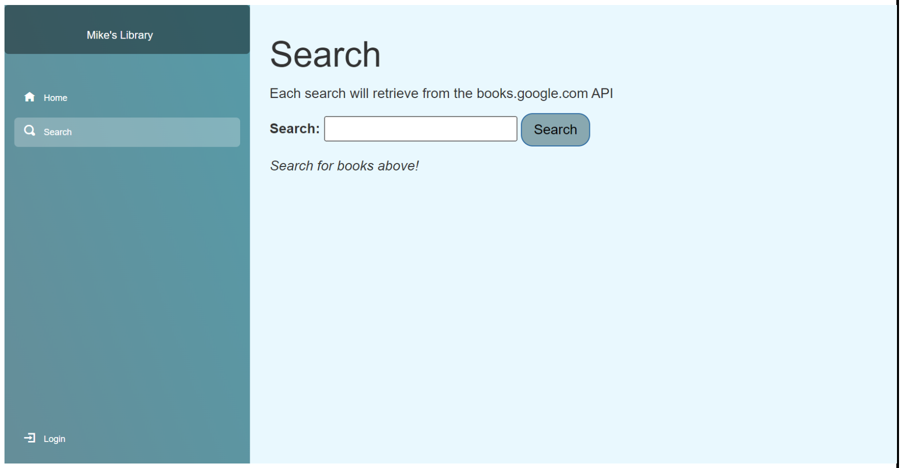
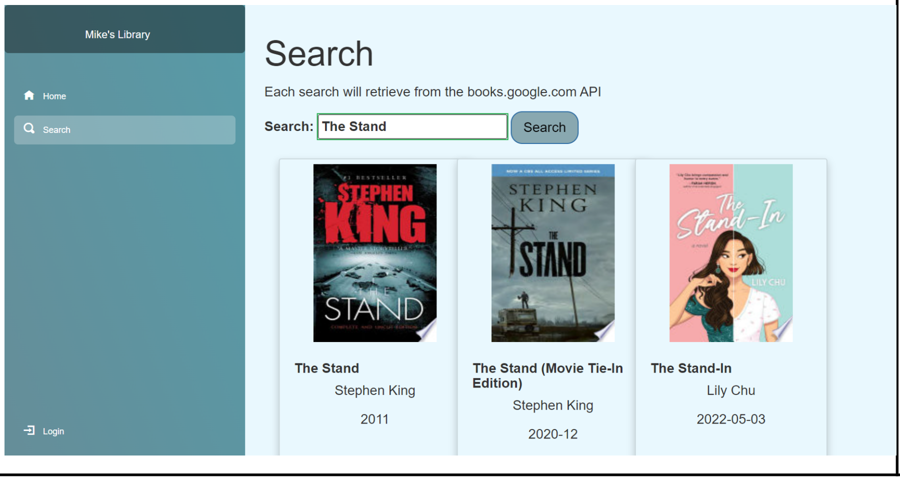
```
An unhandled exception occurred while processing the request.

Exception: Correlation failed.
    Unknown location

Exception: An error was encountered while handling the remote login.
    Microsoft.AspNetCore.Authentication.RemoteAuthenticationHandler<TOptions>.HandleRequestAsync()
```

Figure 16: The correlation error seen during deployment

Results

The final result of my project is a library application built using the ASP.NET C# Blazor framework. This application allows an unlimited amount of users to be able to sign in and create their own bookshelves and to be able to track their book reading habits. A main highlight about this project is that the book tracking user interface for this project simply serves as a mask for the development of more intricate features in the back-end. These features, as described above, are the integration of a database within a web application, making API calls and parsing the information into usable data, developing an authentication framework for users to be able to sign into the application, and finally learning the Blazor framework by following a self-created git version control flow.

The following is a table showcasing the polished user interface of the application, with all of the different pages:

 Mike's Library; Home; Search; Login; Search; Each search will retrieve from the books.google.com API; Search: [ ] Search; *Search for books above!*	The Search page (no searches)
 Mike's Library; Home; Search; Login; Search; Each search will retrieve from the books.google.com API; Search: The Stand Search; **The Stand** Stephen King 2011; **The Stand (Movie Tie-In Edition)** Stephen King 2020-12; **The Stand-In** Lily Chu 2022-05-03	The Search page (with searches)

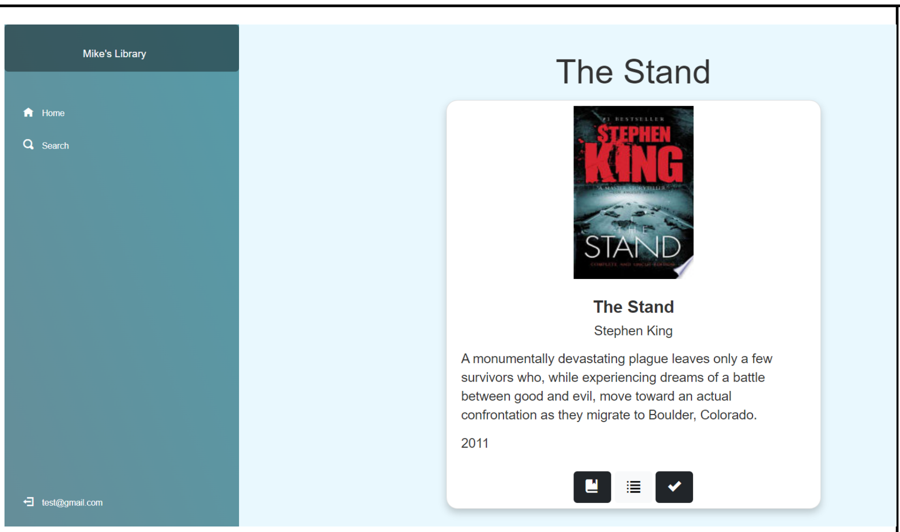
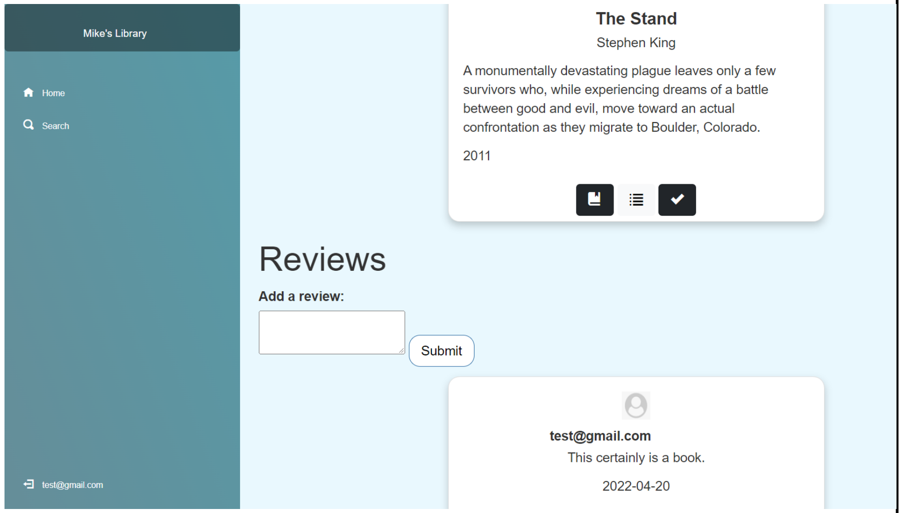
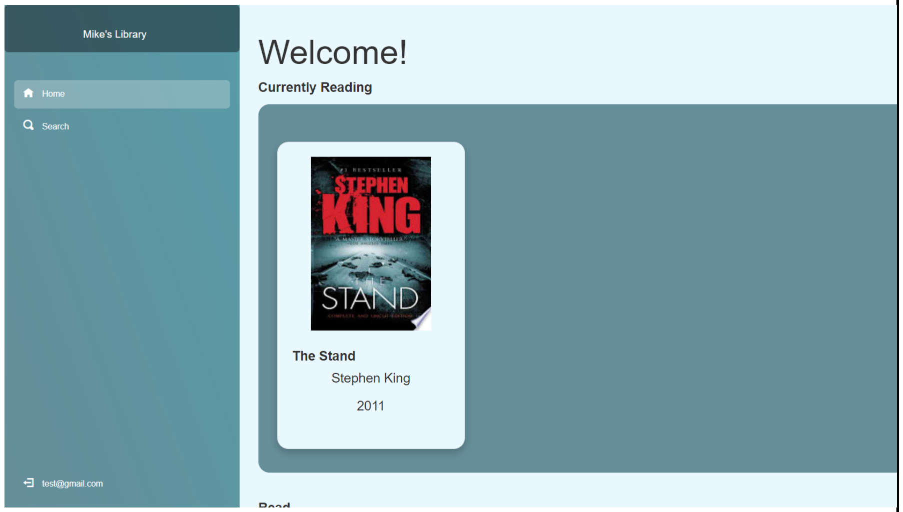
 Mike's Library Home Search test@gmail.com STEPHEN KING THE STAND The Stand **The Stand** Stephen King A monumentally devastating plague leaves only a few survivors who, while experiencing dreams of a battle between good and evil, move toward an actual confrontation as they migrate to Boulder, Colorado. 2011	BookInfo page (top)
 Mike's Library Home Search test@gmail.com Submit **The Stand** Stephen King A monumentally devastating plague leaves only a few survivors who, while experiencing dreams of a battle between good and evil, move toward an actual confrontation as they migrate to Boulder, Colorado. 2011 Reviews **Add a review:** **test@gmail.com** This certainly is a book. 2022-04-20	BookInfo page (bottom)
 Mike's Library Home Search test@gmail.com STEPHEN KING THE STAND Welcome! **Currently Reading** **The Stand** Stephen King 2011 Read	User Homepage

Table 1: The completed pages within the library

A topic that I was not able to fully complete during this project was the introduction of my application to cloud hosting. This was a topic that I wish I had a little more time to be able to fully understand and adapt my application to being able to be accessed anywhere and at any time. Another topic that I did not get to during this project is the implementation of my own search algorithm for my database. Instead, I found a work-around with the Google API to offload the work required to search for books to them, and adapted the query string to produce any amount of results I pleased.

This project is interesting because it is exploring a new and emerging web development framework created and maintained by Microsoft and other contributors, as it is open-source. ASP.NET C# Blazor is a rival to JavaScript, and since a majority of my programming experience has been in languages similar to C#, the transition to working with this framework was very straightforward. After working on this web application for the past four months, I believe that Blazor has a really strong chance of becoming a main framework for web development because it has been so intuitive to pick up and learn, as compared to JavaScript.

Conclusion

Learning the ASP.NET C# Blazor has been a fantastic time and I am incredibly happy with my results. This being the first time that I have completely built a web application from the ground up, I’ve strengthened my knowledge about the intricacies involved in the backend of web development such as using databases and the usefulness of database migrations, developing an authentication scheme, learning API interactions, and proper UI design, while also following Git version control protocols which is a common software development technique seen in the modern workforce. Seeing as I am a very book-driven person, developing this web application on a topic that I am passionate about pushed my drive even further to create a well-polished library and book-tracking system.

References

Chanphakeo, R. (2017, May 29). ASP.NET core MVC identity using postgresql database. Medium. Retrieved April 25, 2022, from https://medium.com/@RobertKhou/asp-net-core-mvc-identity-using-postgresql-database-bc52255f67c4

Foster, B. (2014, March 19). *Asp.net identity stripped bare - MVC part 2*. Ben Foster. Retrieved from https://benfoster.io/blog/aspnet-identity-stripped-bare-mvc-part-2/

Galloway, J. (2022, April 18). *Introduction to ASP.NET identity - ASP.NET 4.X*. ASP.NET 4.x | Microsoft Docs. Retrieved from https://docs.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity

Goodreads. (2022). Goodreads login page. Goodreads. Retrieved from https://www.goodreads.com/

Google. (2022, February 16). Using the API: google books apis: google developers. Google. Retrieved from https://developers.google.com/books/docs/v1/using

Guardrex. (2022, March 25). ASP.NET core blazor JavaScript Interoperability (JS Interop). Microsoft Docs. Retrieved from https://docs.microsoft.com/en-ca/aspnet/core/blazor/javascript-interoperability/?WT.mc_id=dotnet-35129-website&amp;view=aspnetcore-6.0

Nilsson, S. (2021). *Evaluation of the Blazor and Angular frameworks performance for web applications*. Retrieved April 25, 2022, from https://www.diva-portal.org/smash/get/diva2:1578257/FULLTEXT01.pdf

Otto, M. (2022). *Introduction*. Bootstrap v5.1. Retrieved from https://getbootstrap.com/docs/5.1/getting-started/introduction/

Pearson, B. L. (2020, April 23). *How to integrate users into your app with Google OAuth 2.0*. Nylas. Retrieved April 25, 2022, from https://www.nylas.com/blog/integrate-google-oauth

Rousos, M. (2022, March 25). *Overview of ASP.NET core authentication*. Overview of ASP.NET Core Authentication | Microsoft Docs. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-3.1#challenge

Venkat, K. (2019). *ASP.NET core identity UserManager and signinmanager*. ASP.NET Core Identity UserManager and SignInManager. Retrieved from https://csharp-video-tutorials.blogspot.com/2019/06/aspnet-core-identity-usermanager-and.html

Appendix

Github repo containing this project: https://github.com/MikeSimpson1/LibraryProject