Abstract

The goal of this project is to create a podcasting app that makes it easy for listeners to continue their podcasts on different devices without losing their place. The app emphasizes user-friendly design, audio synchronization, and compatibility with various devices, using up-to-date web technologies for a broad range of device support.

In addition to seamless audio continuity, the app offers personalized recommendations, custom playlists, and intuitive controls, enhancing the overall podcast listening experience. Catering to a wide audience, from avid podcast fans to casual listeners, the app ensures users can discover and enjoy content that matches their interests.

This project should investigate the intricacies of a podcasting application by doing research into the pre-existing market and what is important for a podcast application.

By providing a solution for cross-device audio continuity and an enriched listening experience, the app aims to increase podcast engagement and contribute to the industry's growth. The seamless podcasting app offers listeners an enjoyable and accessible way to experience podcasts across devices, fostering a more engaging and personalized podcasting environment.

Contents

1 Introduction

1.1 Background to the project

This project will create a solution for anyone who wants a convenient and easy way to search and listen to any podcast of their choosing.

The reason I choose to create this project is that I felt that there was a need for a simple app that would allow users to search and listen to podcasts of their choosing, whilst still being able to have access across multiple devices.

1.2 Aims and objectives

To present a synchronous podcasting app that allows users to listen to a podcast on one device and be able to continue listening to the podcast from where they left off.

Objective 1 - Build a secure and scalable database and server system for managing podcasts.

• Research suitable database management systems and server technologies.

• Design the database structure and server architecture to accommodate user accounts and podcast information.

• Implement the designed database and server system utilizing Docker images.

• Guarantee security and scalability for future expansion.

The primary objective is to develop a reliable database and server system to effectively handle user accounts and podcast information. This requires understanding the system's needs and selecting the appropriate technologies for implementation. The database and server will be constructed using Docker images, streamlining deployment, scalability, and maintenance. The success of this objective will be gauged by the system's capacity to manage user authentication and grant secure and efficient access to podcasts.

Objective 2 - Design and develop a RESTful backend server for handling user podcast data and ensuring cross-platform playback.

• Explore the application of RESTful services for effective data management and retrieval.

• Design the backend server to manage user podcast data, including the episodes they've watched and their playback progress.

• Implement the server solution using RESTful APIs, allowing seamless data access across devices.

• Test and fine-tune for optimal performance and user experience.

The second objective entails designing and creating a RESTful backend server to oversee user podcast data, such as watched episodes and playback progress. This includes researching efficient data management methods using RESTful services and designing the server to facilitate seamless data access across multiple devices. The server will be implemented with RESTful APIs, ensuring cross-platform compatibility, and improving user satisfaction. The success of this objective will depend on the server's effectiveness in managing and providing user podcast data while delivering a seamless listening experience across platforms.

Objective 3 - Develop an intuitive Windows application for podcast discovery and playback.

• Examine best practices for creating user-friendly graphical user interfaces.

• Design the Windows application's interface and features to improve user experience.

• Implement the designed application, incorporating the necessary functionalities.

• Test and adjust the application to guarantee usability and performance.

The third objective focuses on creating a Windows application that enables users to discover, access, and play podcasts. This process involves understanding user interface design principles and crafting an application that offers a seamless user experience. The Windows application will be designed with an easy-to-use interface, enabling users to efficiently search for and play podcasts. The success of this objective will be assessed by the application's user-friendliness, functionality, and performance, resulting in a positive user experience for podcast listeners.

1.3 Ethical Concerns

Data security and privacy

Data security and privacy are a big concern in a world where users are becoming increasingly aware of their data insecurity and suspicious of the companies that own them. our database will be storing all the podcasts, and episodes, the user has seen. This could be an issue because if there are any data breaches user information could be leaked. Therefore, this product will make sure to secure my database and follow GDPR.

2 Literature review

Technical Background

This project is focused on the problem of being able to access and track podcasts they want to view. “The term podcasting emerged from the use of Apple’s portable audio player, the iPod.” (Oliver McGarr, 2009). For this, it is essential to understand what podcast, synchronous media consumption and technologies will be used.

Podcasting is the practice of sharing audio files over the Internet. These resources can be delivered to subscribers automatically or manually downloaded from the Internet. Podcasts can be stored or streamed on devices such as smartphones, tablets and computers.

The distribution of podcasts relies mainly on RSS (Really Simple Syndication) feeds since they provide a common format for distributing ongoing material and metadata among several devices and apps. Podcast users subscribe to these feeds using podcast applications and aggregators to automatically get new material, and podcast authors utilise RSS feeds to post and update episodes. A podcast's and its episodes' names, descriptions, release dates, and audio file URLs are all included in an RSS feed, an XML document.

The MP3 file format, which has emerged as the current norm for audio streaming and downloading, is usually used for the distribution of podcasts, the mp3 has occupied centre stage in the world of digital audio formats (Jonathan Sterne, 2006). MP3 files are suitable for streaming and downloading across a range of devices and network setups as they provide excellent audio quality while maintaining relatively small file sizes. However, some argue that more recent audio formats, like AAC or Opus, provide better audio quality and compression than MP3 and may, in the future, be more appropriate for podcast distribution. The MP3 format is still preferred for podcasts despite the arrival of alternatives due to its broad compatibility, ease of use, and the existing frameworks built around it.

“Restful API, REST (Representational State Transfer) is an application of web-based communication architecture model. REST applications are often used for the development of web or mobile based services.” (Rudiastini E, 2018). Developers may use the RESTful API to carry out common tasks like create, read, update, and delete (CRUD) on resources, which can be any type of data item, including podcast libraries, playing status, or user preferences in the context of podcast synchronisation. Uniform Resource Identifiers (URIs), which are often URLs, are used to identify resources, and common HTTP methods like GET, POST, PUT, and DELETE are used to access and modify those resources.

Databases are crucial parts of modern software programmes because they offer an organised and effective way to store, retrieve, and manage data. They provide the basis for a wide range of systems, from straightforward personal applications to complex business solutions. There are many kinds of databases, such as relational databases, which use tables to store data and establish connections between records, and NoSQL databases, which offer flexible data models appropriate for managing unstructured or semi-structured data.

Developers may bundle apps and their dependencies into portable, self-contained entities known as containers thanks to containerization, a lightweight, effective virtualization technique. By enabling consistent operation across various settings, including development, testing, and production, these containers can solve the "it works on my machine" issue and simplify the deployment procedure. Containerization may be used to deploy and manage many system components, including web servers, APIs, databases, and cache layers, in the context of podcast synchronisation services.

Developers may assure a consistent and dependable infrastructure, streamline the development and deployment process, and optimise resource utilisation across various settings by implementing containerization.

For this project synchronous applications refer to the ability for a user to begin watching a podcast or piece of media on one device stop and be able to continue from another device exactly where they left off. A synchronous application should also track what podcasts a user has seen allowing them to switch between podcasts at will this is important as it allows for users to access their content no matter the conditions.

2.2 Pre-existing alternatives

2.2.1 2.2.1 Apple Podcasts

Apple podcasts is one of the most popular podcasting applications developed by Apple Inc in 2012, apple podcasts are available for iOS, iPadOS, macOS and tvOS, "There are now more than one million podcast shows on Apple Podcasts" (Matt Binder).

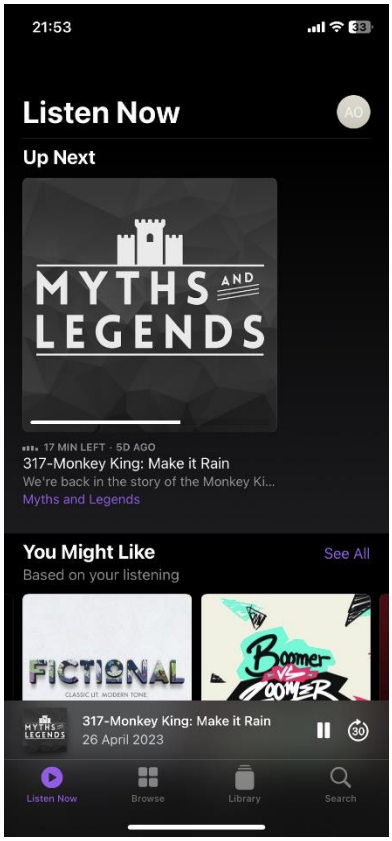

Figure 1 – Screenshot of Apple Podcasts app Listen now page

Figure 1 shows the listen now tab from the Apple podcast app on iPhone, the app features a listen now; browse; library and search tabs, On this page the app shows new episodes for podcast that the user listen to as well as podcast that are similar or recommended based on the users listening history.

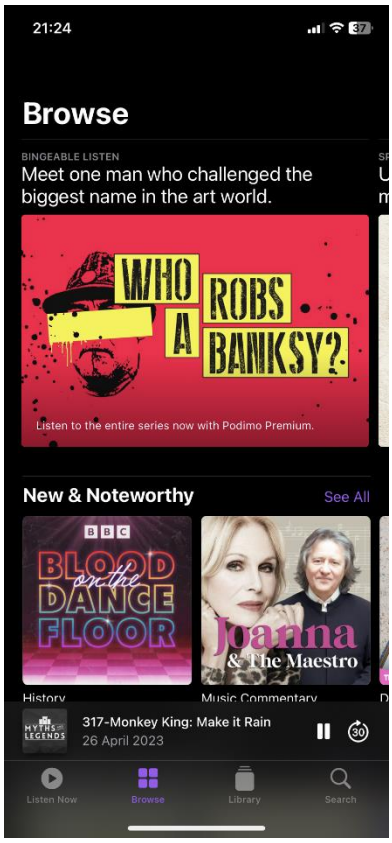

Figure 2 – Screenshot of Apple Podcasts app browse page

Figure 2 shows the browse tab of the Apple podcasts app, this tab is used for displaying podcast that apple suggest there users to listen to. This tab is useful for users who are new to the platform and have not listened to enough podcast to know what they want to listen to or enough for the apple recommender to have a good idea on what they want to listen to

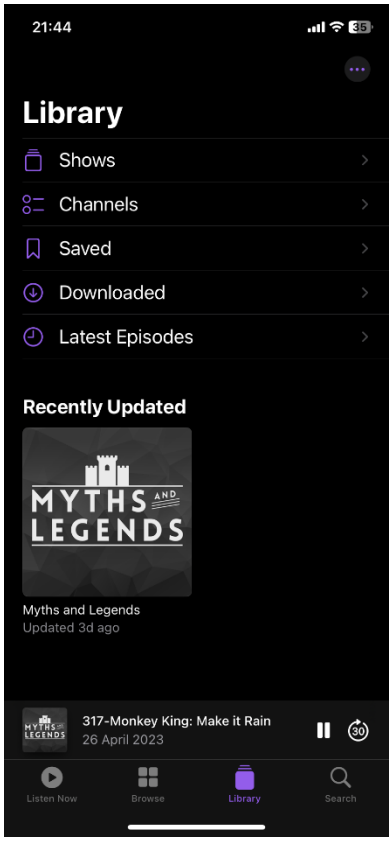

Figure 3 – Screenshot of Apple Podcasts app Listen now page

Figure 3 shows the library tab of the Apple podcast app, this tab is used for displaying to the user all of the podcasts that they have saved, this page also allows for collections based on channels that the user saves individual episodes, downloaded episodes and the latest episodes for the podcasts in their saved.

Figure 4 – Screenshot of Apple Podcasts app Listen now page

Figure 4 shows the search tab from the Apple podcast app, this tab features a search bar that allows users to search up any podcast apple has stored in iTunes based on multiple search terms such as (genre, title, author), the page also allows for users to search by category clicking on one of the panels will bring up the top shows as rated by other users on the platform.

Benefits

- The benefits of Apple Podcasts are that it has a very good and intuitive UI design making it easy for new users to pick up the app and find content
- Library splitting by multiple categories means that users can easily group podcasts they might want to watch
- Search allows users to lookup more than just a title, users can search on genre and artist

Drawbacks

- Limited to Apple devices
- Forced Ads
- Limited Customization on playlists

2.2.2 2.2.2 Spotify Podcasts

The Spotify app is a popular music streaming platform, that provides a vast selection of podcasts including content exclusive to Spotify. Spotify combines both podcasting and music into one app and synchronises the listening experience across a multitude of devices, “Spotify recommended a particular podcast based on your current listening habits, you

could visit the show's page and create a custom playlist of the episodes that looked most interesting."(Perez, S, 2020)

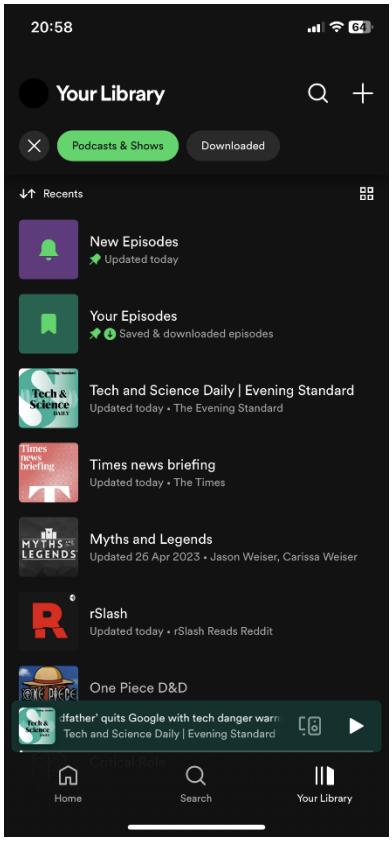

Figure 5 – Screenshot from the Spotify app "Library" page

Figure 5 shows the library tab from the Spotify app, Spotify do not split there music and podcast section of their app however they allow for users to activate tags that refine the content being displayed. in figure 5 its shows the "Podcasts and shows", Spotify also allows for users to navigate to two other pages New Episodes and Your Episodes.

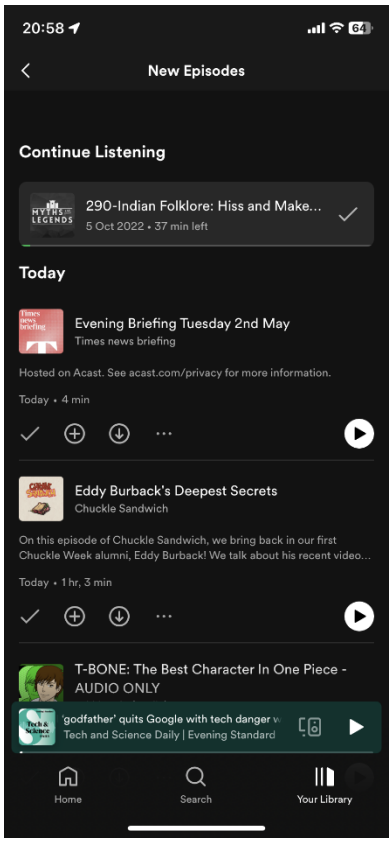

Figure 6 – Screenshot from the Spotify app "New Episodes" page

Figure 6 shows how Spotify presents the most recent episodes of the podcasts that users are subscribing to are displayed on the Spotify "New Episodes" tab, a special section within the app. With the help of this tool, listeners may instantly access new content as well as keep up with their favourite shows. Users can easily find and choose new episodes thanks to a simplified interface, ensuring a pleasant and interesting podcast experience on Spotify.

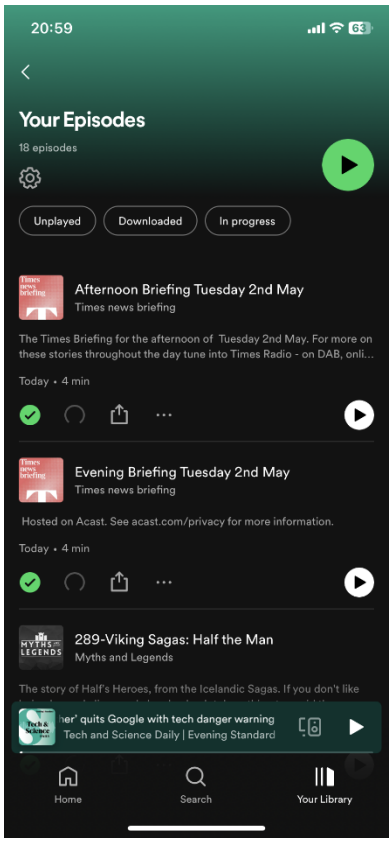

Figure 7 – Screenshot from the Spotify app “Your Episodes” page

Figure 7 shows how users can view saved podcast episodes at their convenience through the "Your Episodes" menu on the Spotify app. With the help of this tool, listeners can compile a list of episodes they want to review or catch up on in the future. Users may effectively manage their stored episodes and take advantage of a customised podcast experience on Spotify thanks to an organised UI.

Benefits

- Large content library: Spotify offers customers a wide range of shows and genres to pick from by hosting a large selection of podcasts, including exclusive content.
- Cross-platform availability: Spotify can be used on an extensive list of devices, including iOS, Android, Windows, macOS, and web browsers, ensuring a wider audience.
- Offline listening: Users can download podcast episodes for offline listening, allowing them to enjoy content without an internet connection.

Draw Backs

- Limited podcast-specific features: In comparison to apps that are entirely devoted to podcasting, Spotify offers fewer podcast-specific features like advanced playback controls, thorough show notes, and chapter support.
- Podcast discoverability: Although Spotify offers tailored recommendations, its search and discovery capabilities for podcasts often fall short of those of dedicated podcast apps, making it more difficult for users to discover new podcasts.

2.2.3 2.2.3 Pocket casts

Pocket Casts is a popular podcast management app that offers a user-friendly interface for discovering, subscribing, and listening to podcasts. It allows users to synchronize their podcast library and playback progress across multiple devices, making it easy to switch between platforms seamlessly. The app also offers customizable playlists, advanced playback controls, and various discovery features.

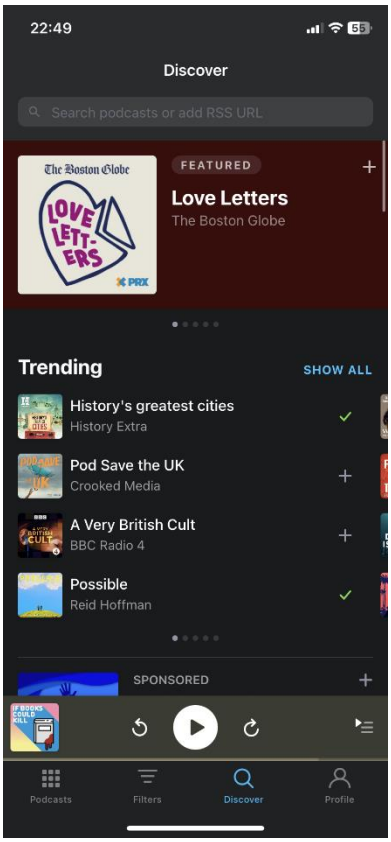

Figure 8 – Screenshot from the Pocket Casts app “Discover” page.

Figure 8 shows how users can find a wide variety of podcasts on Pocket Casts' Discover page, which provides a robust podcast discovery experience. Users can use the page's selected lists, top charts, and categories to find new podcasts that match their interests. Users can use the powerful search feature to find podcast.

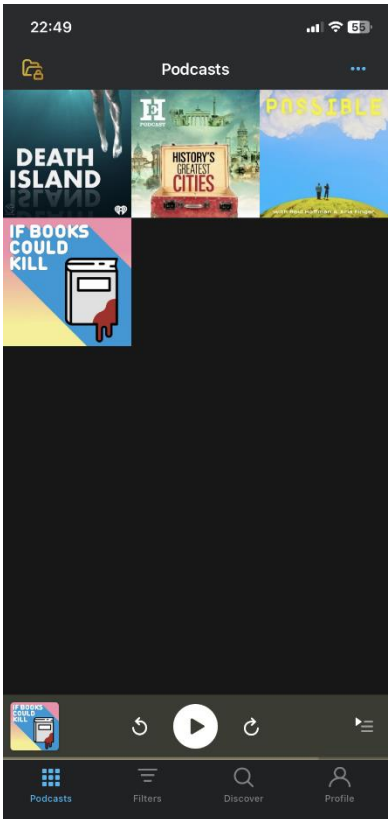

Figure 9 – Screenshot from the Pocket Casts app “Podcasts” page.

Figure 9 shows how users can control and access their subscribed podcasts from using the Podcasts tab in Pocket Casts. They can access their library effortlessly due to the clear and organised structure, which shows the podcast cover art.

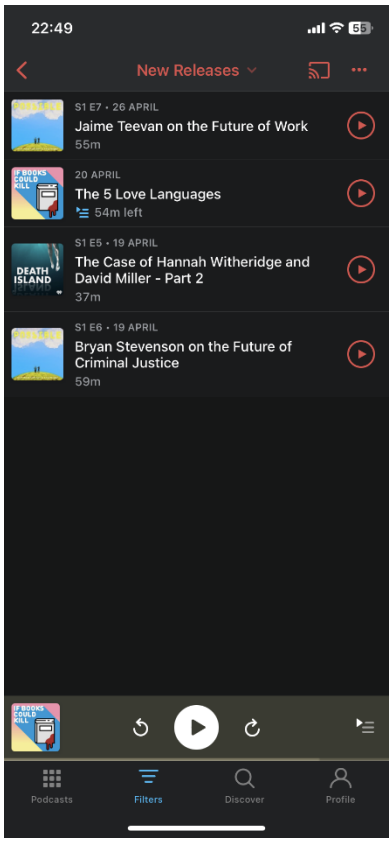

Figure 10 – Screenshot from the Pocket Casts app “New Releases” Filter.

Figure 10 shows the new releases filter, the Pocket Casts New Releases filter displays the most recent episodes of the user’s subscribed podcasts. This filter makes it easier for users to track their favourite podcasts and makes sure they never miss a new episode. The most current episodes are simple to find and play because to a simple structure, which improves listeners' podcasting experience.

Benefits

- Advanced playback controls: Pocket casts gives users the option to customise their listening experience through advanced playback settings like adjustable speed and volume boost.
- Stronger search features and curated lists in Pocket Casts make it easier for users to find new podcasts that match their tastes and interests.
- Podcast collection management and organisation are made simple by the app's ability to allow users to create personalised filters and playlists.

Drawbacks

- Users who are new to podcast apps or those switching from simpler alternatives may experience a longer learning curve due to the app's rich functionality and customization choices.
- Occasional device sync issues have been reported by some users. These issues may lead to inconsistent listening experiences.

2.3 2.3 Conclusion

In conclusion, this literature review has successfully delved into the key technical aspects of the podcast application domain. By discussing vital technologies such as RSS feeds, RESTful APIs, databases, and containerization, we've laid a solid foundation for developing a new, synchronous podcast app. To ensure an effective design, we'll also take a close look at popular existing alternatives like Apple Podcasts, Spotify Podcasts, Pocket Casts, and Overcast, examining their unique features, advantages, and limitations. By understanding the user experience and features provided by these well-known podcast applications, we can create a new synchronous podcast app that overcomes any shortcomings and builds on their strengths. This will result in a more comprehensive and enhanced solution for podcast enthusiasts, ultimately providing a seamless and enjoyable podcast listening experience across multiple devices.

3 Requirements

3.1 Product requirements

Based on the research conducted for the literature review the final product will be a Windows application that allows users to keep a library of podcasts and download them, users should be able to have an account that stores their podcasts in a library and episodes should continue from where they left off.

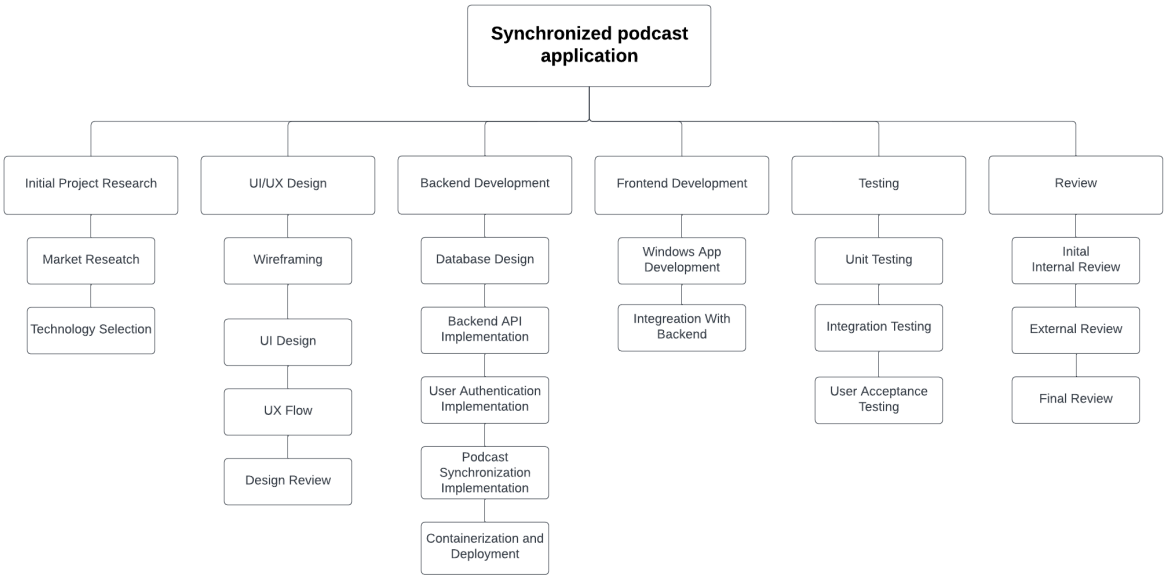

Figure 11 Project Breakdown Chart: A Roadmap for Developing a Synchronous Podcast Application

The breakdown chart illustrates the roadmap for project development and highlights key development milestones that are needed for the project's success. Initially, my project proposal included the development of an Android app, however, due to time constraints and the significant time investment required to learn and develop an effective Android app, this aspect was removed from the project scope.

3.2 Functional requirements

The Windows app must allow a user to log in with an account that will store information on their podcasts and how far through an episode they are, they should have access to basic media playback that will allow them to skip through media, pause, play, and control media volume. There should be an option that allows them to download the newest episode of a podcast that they are subscribed to automatically. The user interface should be somewhat intuitive and easy to use.

3.2.1 Interfaces

For this project the user interface will be a Windows app, the app will have to be responsive and user-friendly with a design that is clear and intuitive. The layout should be clear and simple to navigate with the most important user information presented upfront in a space that will be easy for the users.

This project will be using the iTunes Search API. The iTunes Search API is a free, open-source RESTful API provided by Apple that allows developers to search and obtain data on a range of media types, including podcasts, that are accessible through the iTunes Store and Apple Podcasts, the responses are typically in the JSON format and return a list of objects that contain information such as (podcast

name, genre information, podcast artist, and the RSS feed URL) these responses can then be parsed and sent to the podcast builder.

This project will also utilize RSS feeds for podcast management. RSS feeds, or Really Simple Syndication feeds, are files based on XML that allows for the effortless distribution and access to frequently updated web content, including podcasts. By subscribing to an RSS feed of a podcast, users can get new episodes and related metadata, such as episode title, description, and release date, automatically. Within the scope of this project, the application will parse the RSS feeds to obtain important information and fill the podcast library, guaranteeing that users have the latest episodes and updates from their preferred podcasts.

The plan for this project involves creating a RESTful server as the backbone for managing essential information such as podcast details and user login data. This backend system will be set up to facilitate smooth communication between the client application and the server, enabling efficient access and organization of the required data. By employing a RESTful server, the system will adeptly handle requests related to user authentication, podcast subscriptions, and playback updates, among others. This design choice ensures a seamless and personalized user experience while also considering aspects such as scalability and maintainability for potential future developments and adjustments.

3.2.2 Functional Capabilities

The solution should contain each of the following:

3.2.2.1 Windows Application

The main goal of this project is to develop an easy-to-use podcast management and playback app for Windows users. Users should be able to sign in with an account to save their podcast subscriptions and track their playback progress. The app should have a user-friendly interface, allowing users to effortlessly manage podcast subscriptions, access podcast information, and control media playback. Basic media control options should be available, such as play, pause, skip, and volume adjustment. Users should also be able to enable automatic downloads of the latest episodes from their subscribed podcasts.

3.2.2.2 Back End Server

The backend services are responsible for handling user login details and podcast data. A RESTful server will be implemented for the client app to access the necessary information. The server should be configured to enable effective communication between the client app and the database, providing a smooth user experience.

3.2.2.3 Database

A well-designed database is essential for storing user account information, podcast subscription data, and playback progress. The database should work closely with the backend services to supply the client app with the required data, ensuring seamless podcast management for users.

3.2.2.4 Integration of External API

In addition to the primary components of the project, the app will utilize external APIs like the iTunes Search API and RSS feeds to access and display podcast details. These APIs should be integrated into the app to offer users a diverse range of podcasts, helping them easily discover and subscribe to new content.

3.2.3 Programming Language & Frameworks

C#:

C# is an object-oriented programming language created by Microsoft in 2002 as an alternative to Java and is often used in applications such as desktop applications, web applications, web services, games, mobile applications, and database applications. C# is one of the most popular programming languages to use, is easy to learn and has a huge community that has created modules for the language. C# is a requirement for this project as the libraries that have been created are used so often that they are properly developed an example of this is the Newtonsoft JSON library that can both serialize and deserialize JSON files and text. This is beneficial as a custom-created JSON handler would be an extreme time investment if you account for initial development as well as time that would be needed to fix any edge case bugs that may arise, this helps the project be developed within the time constraints set.

ASP.NET:

ASP.NET is a web framework for C# used for building websites, web applications and web API using real-time technologies like web sockets. Web APIs in ASP.NET are easy to build HTTP services and serve as an ideal platform for building RESTful (Representational State Transfer) applications such as what is needed in this project.

WinForms:

WinForms(Windows Forms) is a Graphical User Interface(GUI) class library which is bundled in .Net Framework, it provides an easy interface to develop desktop applications on Windows, and it offers a wide range of tools that allow for, WinForms is great for rapid development as it uses a drag and drop style UI development make it quick and easy for developers to create functional interfaces additionally WinForms is compatible with the .NET Framework leveraging all C#'s many libraries and allowing for a desktop application to run on a wide array of Windows computers.

3.2.4 Tools

An IDE (Integrated software environment) is software that is used for building applications, IDEs combine multiple common developer tools into a single piece of software typically an IDE consists of a source code editor, a compiler and a debugger. Choosing the correct IDE for a project can drastically reduce the time it takes to develop a project.

For this project, the frontend and backend will both be developed using Visual Studio 2022 the following are some of the reasons:

- Debugging: In software engineering debugging is a multi-step problem that involves, identifying, isolating, and correcting a problem Visual Studio helps by detecting an error showing you the line that it happened on and showing you an exception that was raised solving the first 2 steps for a developer.
- IntelliSense: IntelliSense is a code-compilation aid that will recommend the rest of a line based on what you have currently inputted, this is a feature that can drastically increase the speed at which a programmer can develop a feature.
- Integrated unit testing

For this project the backend server and database will need to be containerized to provide a dependable, portable, and secure environment to run applications that can easily be set up on a wide array of devices for testing and scaled quickly in deployment for this docker has been chosen as docker offers a further layer of abstraction compared to other containers making it easier to deploy.

Version control is an important requirement for this project. This project will be using the version control Git as it is used to control and manage different versions of applications allowing developers to easily revert to an earlier version if there are any detrimental issues with the current instance of a project, Git also allows for developers to branch there code, a branch is a copy of the codebase split off from main development allowing for parallel development of features.

3.2.5 Security/Privacy

One security requirement for this project is to hash user passwords, hashing passwords is done as if the database storing all the user passwords was to be compromised it would result in all of the user passwords being leaked hashing is a suitable remedy for this as even if the database is breached the attacker would only be able to obtain hashed versions of the passwords.

Another security requirement for this project is for all the data stored in the databases should be encrypted. For similar reasons as the password if the database is breached user data will be leaked to prevent this from occurring all the user data stored in the database should be encrypted.

3.2.6 Project Management

Waterfall: The waterfall method is a project management and development that focuses on a linear progression from the start to end of a project, it relies heavily on careful planning, documentation, and good execution each phase must be completed before moving on to the next.

The phases of waterfall consist of:

- Analysis: for waterfall to be effective you must be able to gather and evaluate all the necessary information on project requirements upfront. The project manager should gain a detailed understanding of the project stakeholders' requirements, risks, costs, and timelines.
- Design: The development team should take all the requirements and project information gathered during analysis and begin to design technical solutions to the problems set out, initially a high-level design is created that illustrates the scope of the project and the general flow of the product. Once this is complete a more in detail design that covers specific technologies can be created.
- Development: After the design has been completed technical implementation can begin, this is typically the shortest phase of the waterfall process as all the research and design have already taken place, if there are crucial roadblocks there may be a need to go back to the design stage with the new information learnt.
- Testing: Before the product can release testing occurs to ensure that the product has no fatal errors and meets all the requirements expected by the stakeholders, during this time the testing team will typically reference the design documents supplied from the design stage.
- Deployment: Once the software has been deployed user reactions will be collated and any necessary updates and improvements that were not discovered during testing will be assigned to the relevant team members.

Advantages of using waterfall:

- Waterfall has a straightforward methodology that outlays requirements from the beginning of the project allowing effective time management.
- Project costs can be accurately estimated as early in the project as requirements and features have been defined.
- Easier to define milestones during the early stages of the project.

Disadvantages of waterfall:

- Waterfall can be rigid if new requirements arrive late in the project the whole process must be restarted with new information in mind.
- Clients are not involved in design and Development.
- A delay in one phase can have a knock-on effect on the timeline of the entire project.

3.3 Design constraints

One design constraint that this project can face is that Apple has limited their iTunes search API to 20 requests per minute. Initially, it was intended that the client would send the server a request that would include a podcast name and the server would contact iTunes search API parse the response and send the client a list of acceptable fully built including podcast author, thumbnail, and episodes list that could then be displayed to the user. However since Apple limits requests to 20 per minute, the server would run out of requests extremely quickly and would not be scalable due to this the client will instead query iTunes and build the podcast itself, this could cause an increase in delay and negatively affect user experience as now app performance relies heavily on the device it is being run on it is a suitable trade-off for the needs of this project.

Another design constraint that was faced during this project was due to time, initially, the project was going to include an Android app however due to time constraints this had to be dropped.

4 Design

4.1 Software design

4.1.1 Windows Application Design

4.1.1.1 User Flow

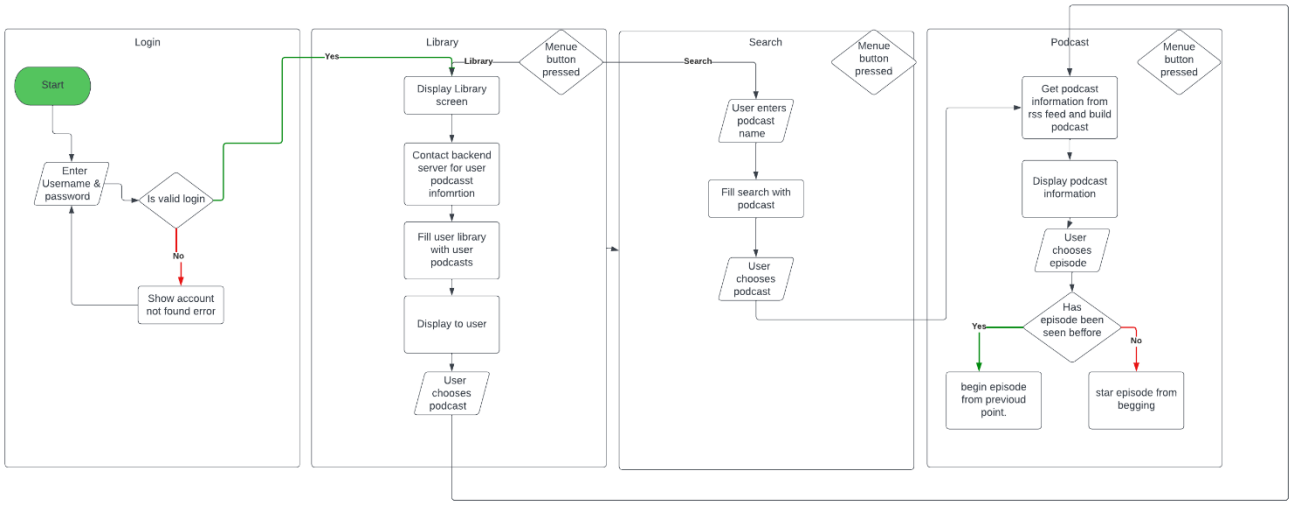

Figure 12 User Flow Diagram

A user flow diagram, also known as UI flow, is a visual diagram that illustrates the general flow that a user will experience when using an application, designers should decide on user wants and needs when navigating an application.

As seen in Figure 12 the user will begin in the login screen where users will enter username and password, if the account does not exist there will be an error message, if the account does exist it will move the user onto the library page.

From the library page, the app will contact the backend server and get the user's podcast library data and fill out the library page with the user's podcasts if a user clicks on a podcast, it will send them to that podcasts podcast page.

The podcast page will use the RSS Feed of the podcast to fill out information such as podcast name, podcast author, podcast episodes, podcast thumbnail, and podcast description and display that information back to the users. Once the user chooses an episode it will check if the episode has been seen and downloaded if it has it will continue from where it was if not it will download and begin the episode from the start.

The search page will take a podcast name from the user and fill the page with search queries from Apple's iTunes search results when the user chooses a podcast they will be sent to that podcast page.

Depending on the menu button pressed it will take the user to either the search or library page.

4.1.1.2 Wireframing

Wireframing is the act of designing a user interface at the structural level, wireframes are frequently used to plan out application content and functionality as this allows designers to get a feel for what

the application will flow like and if there are any problems it is much easier to edit a wireframe that an application mid development.

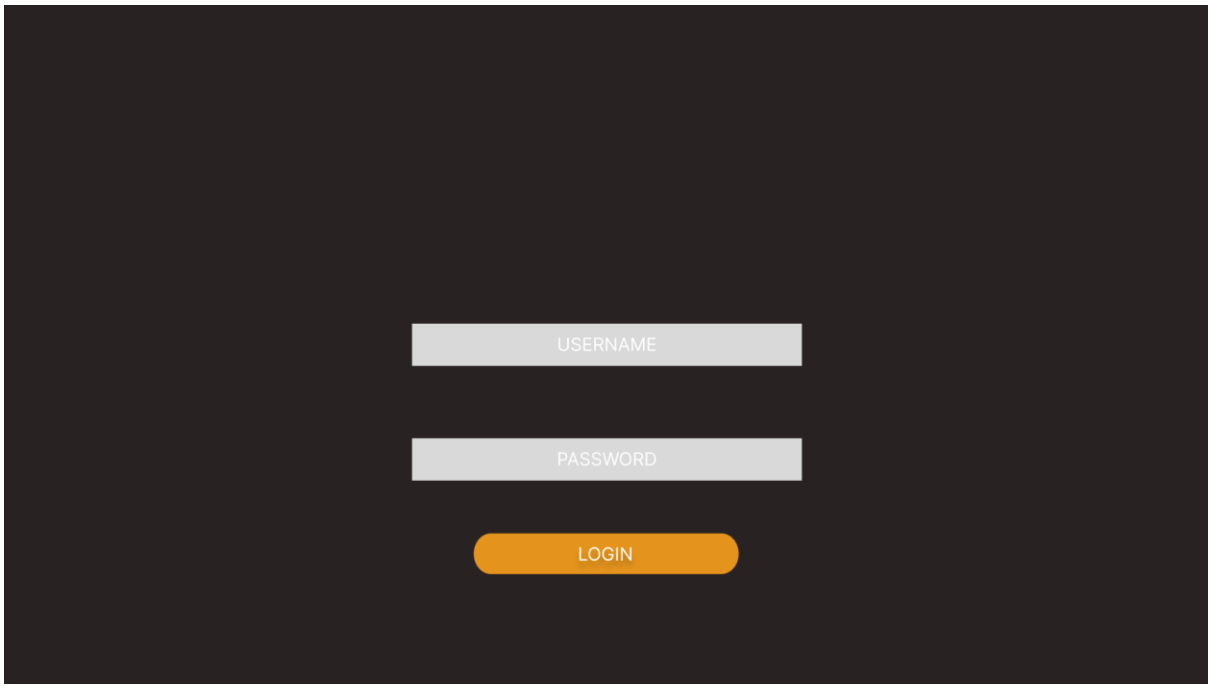

Figure 13 Login Screen Wireframe

As seen in Figure 13 for the login screen wireframe there are two text boxes and a button this was chosen to keep the design clean and minimal.

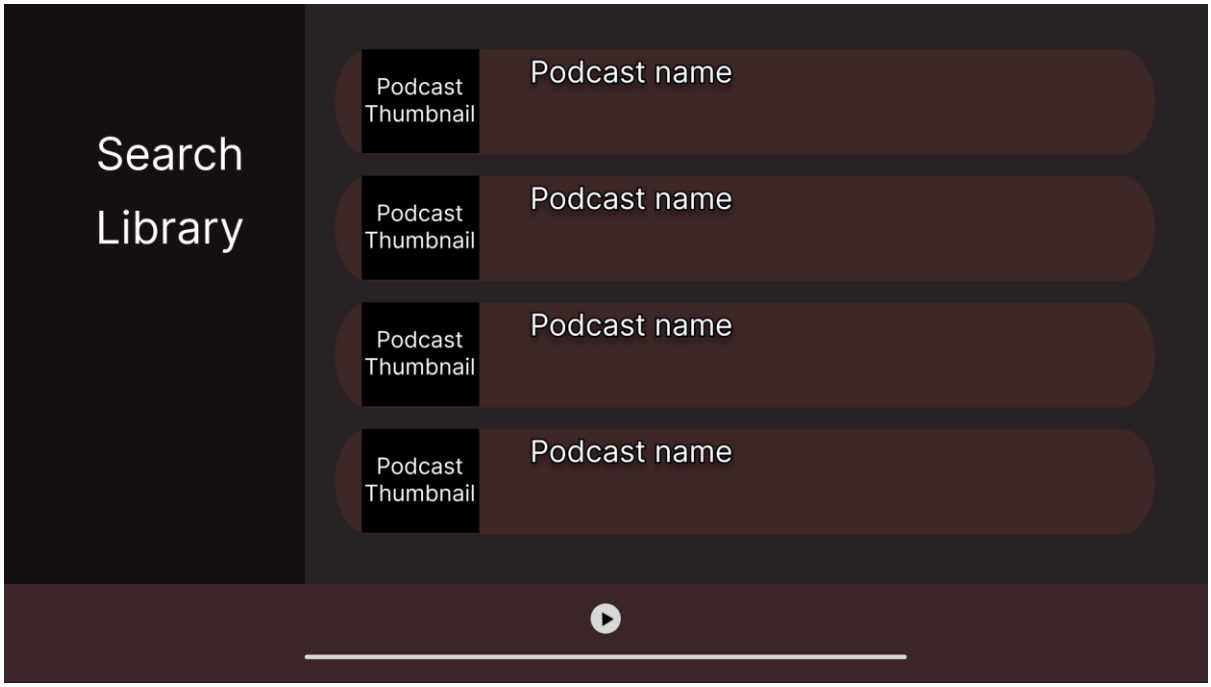

Figure 14 Library Screen

For this project there will be a need to display a user's podcast library the Figure 14 library screen will prototype this, on the left there is a menu bar that contains two buttons "Search" & "Library" These will allow the user to switch between the two screens. On the bottom of the screen is the media controller this will allow the user to play, pause and skip through the media that is currently queued.

This screen will consist of a list that contains all the podcasts in a user's podcast library podcasts should essentially be custom buttons that will display their name and thumbnail and users can click on them to bring up the podcast screen with the podcast.

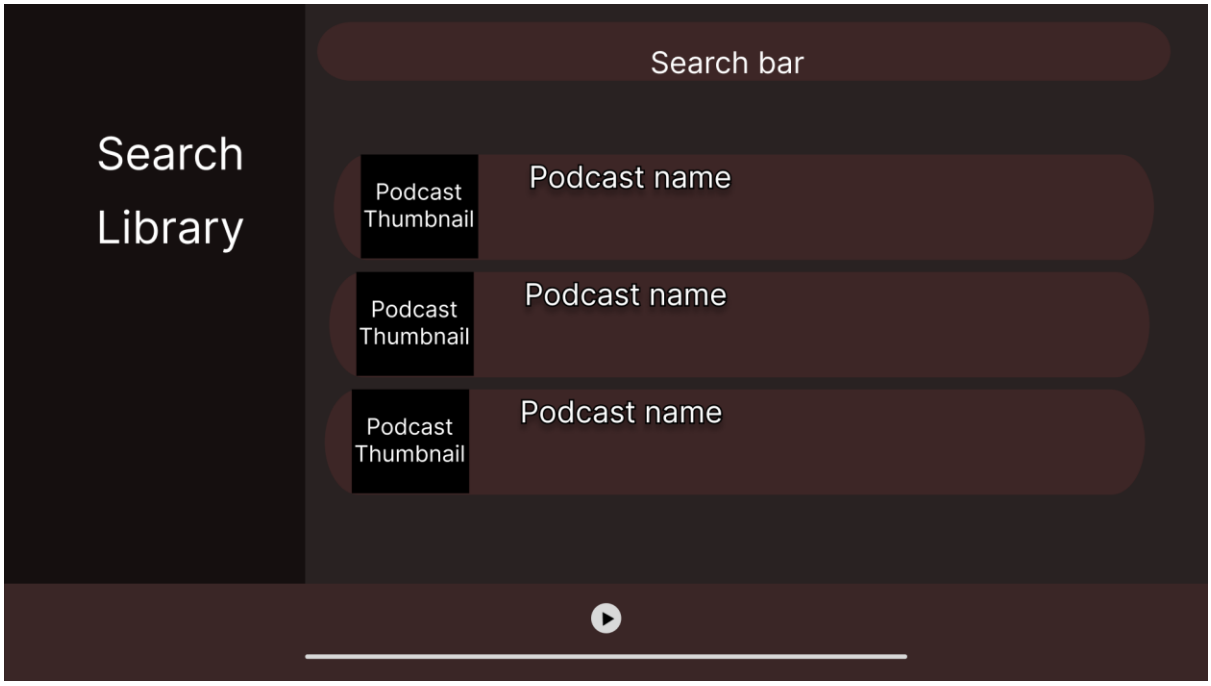

Figure 15 Search Screen

For the search screen, there will be a text box that allows users to enter a podcast name, once a user submits a list of podcast results will be filled allowing users to click on a podcast and add it to their library.

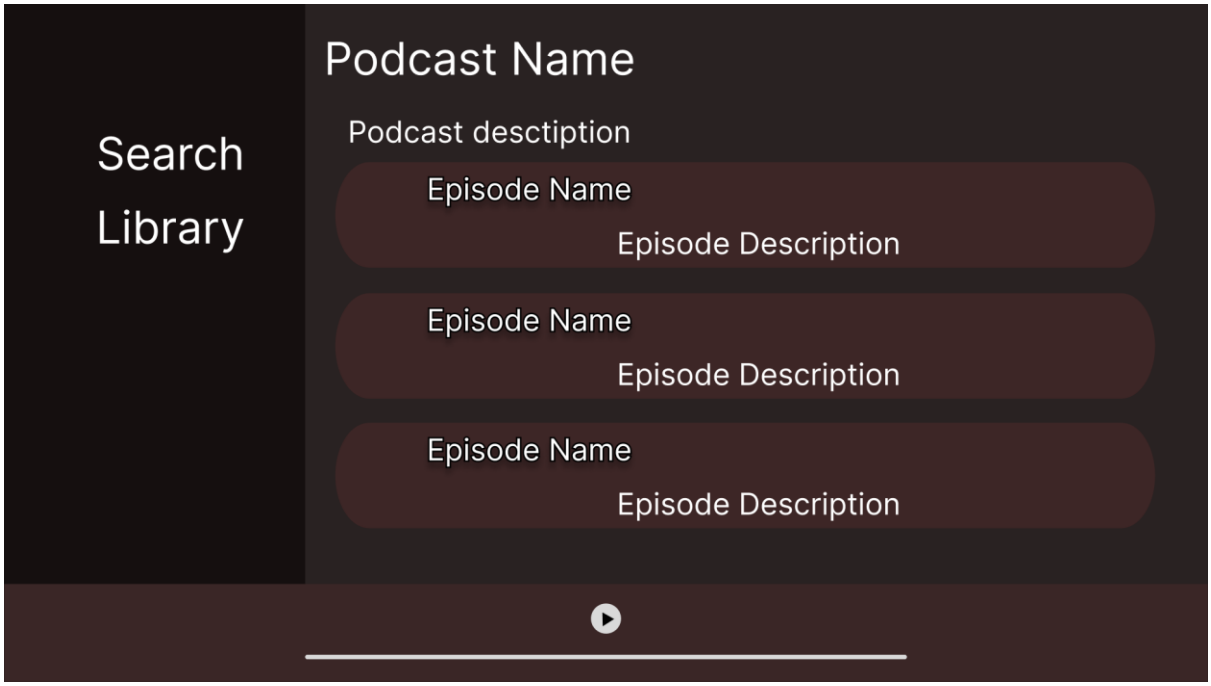

Figure 16 Podcast Screen

For the podcast screen, there will be the podcast name, descriptions, and list of episodes, similarly, the podcast's episodes should be a custom button that displays the episode name and description and when clicked on will be downloaded and played.

4.1.1.3 iTunes search API communication

For this project, there will be a need for a way to search and index podcasts the Apple “iTunes Search API” was chosen for this. The Search API allows for searching of content in the iTunes Store and podcasts.

Example request

```
GET https://itunes.apple.com/search?term=[SearchTerm]&entity=podcast HTTP/1.1

Host: itunes.apple.com

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.82 Safari/537.36

Accept: application/json
```

Example Response

```
{
 "resultCount": 1,
 "results": [
  {
   "kind": "podcast",
   "collectionId": [Podcast ID],
   "artistName": [Artist/Creator],
   "collectionName": [Podcast Title],
   "feedUrl": [Podcast Feed URL],
   "artworkUrl100": [Artwork URL],
   "releaseDate": [Latest Release Date],
   "primaryGenreName": [Genre]
  }
 ]
}
```

4.1.1.4 *Important code*

The following are important methods and classes that need to be designed before implementation represented in pseudocode that will be represented in a Python-like pseudocode for readability and it has access to the functions and libraries that would be needed to properly illustrate what would be needed for these methods:

- GET Request to iTunes Search API

```
def lookup_podcast(podcast_name):
    encoded_name = parseName(podcast_name)
    url =f"https://itunes.apple.com/search?term={encoded_name}&entity=podcast"
    response_string = requests.get(url).text
    response = json.loads
      (
      response_string,
      object_hook=lambda d: {k.lower(): v for k, v in d.items()}
      )

    if len(response['results']) > 0:
        return json.dumps(response)
    else:
        print(f"No results found for '{podcast_name}'")
        return ""
```

The function begins taking an input that is a string podcast_name and encodes it, so that It can be put in the get request, and does a get request to the iTunes search API with the encoded podcast name, serializes a new JSON object, if the response has more than one result the response is returned if not it returns an empty string.

- Podcast RSS feed data collector:

```
def parse_rss_feed_data(url):
    response = requests.get(url)
    xml_data = response.text

    podcast_info = {
        'title': extract_podcast_title(xml_data),
        'description': extract_podcast_description(xml_data),
        'image': extract_podcast_image(xml_data),
        'episodes': extract_episode_list(xml_data)
    }

    return podcast_info
```

This function begins by taking an input parameter which is the RSS feed URL it then creates a get request and stores all the collected data in “xml_data” From there a new dictionary is created that will represent the object that the code will create, and it calls extract methods that parse the XML code and pull the data needed.

- Contact Backend Server to get podcast information.

```
def send_get_request(url, api_key):
    try:
        request_url = f"{url}/{api_key}"
        response = requests.get(request_url)
        response_string = response.text

        if response_string.strip() == "":
            return None
        else:
            return response_string
    except Exception as ex:
        print(str(ex))
        return None
```

This method takes two strings as parameters the URL of the site and the user's API key and requests the server check that the response isn’t an empty string after removing trailing or leading whitespaces and return the response.

- Update podcast information on the server

```
def send_post_request_json(url, api_key, json_file_path):
    try:
        with open(json_file_path, 'r') as file:
            json_data = file.read()

        post_data = {
            "APIKey": api_key,
            "LibraryData": json_data
        }
        headers = {"Content-Type": "application/json"}

        response = requests.post(url, json=post_data, headers=headers)
        response_string = response.text
        return response_string

    except Exception as ex:
        print(str(ex))
```

This method updates the podcast library information held in the backend it takes 3 strings as parameters the server URL the user’s API key and the file path that the library is being cached to locally, the file is opened and reads into the variable json_data and a dictionary

that will represent the object that will hold the key-value pair that identifies a user's data through their API key the request is posted to the server and the response is returned

- Fill the search panel with new custom podcast buttons

```
def search_and_add_new_user_control(podcastName):
    search_response_panel.clear()
    print(f"Searching for podcast {podcastName}")

    results = itunes_lookup.lookup_podcast(podcastName)

    for i in range(len(results)):
        result = results[i]
        podcast = await Podcast.build_podcast_from_rss(result)


        podcastButton = SearchResponseUC(podcast, target_panel)
        search_response_panel.controls.add(podcastButton)
```

4.1.2 Backend Server Design

4.1.2.1 The connection between the client and the backend server

Example requests the client would make to the server:

Login Request

POST https://localhost:44390/api/user/
BODY:
{
Username: [username]
Password: [password]
}

Update Podcast Library:

```
POST https://localhost:44390/api/data/
BODY:
{
ApiKey: [User API key]
library: [user library json information]
}
```

Get podcast library:

```
POST https://localhost:44390/api/data/[user api key]
```

4.1.2.2 Rest API

For this project, the server's architecture will be rooted in RESTful principles to guarantee compatibility and fluid communication with the client application. The server's main function will be to oversee user podcast data and the tracking of users' watched episodes and their playback progression. The design of the backend server is set to incorporate various design patterns and best practices to ensure its robustness, ease of maintenance, and scalability. Among these, the Model-View-Controller (MVC) design pattern is a significant choice. This pattern will split business logic from request definitions, improve the readability of the code, and enhance maintainability.

5 Implementation and testing

5.1 Implementation

This section will discuss the creation and implementation of technically difficult parts of this project. It will take into consideration how modules adjusted over time to find the best choice for a solution.

5.1.1 Creating the main screen

For the creation of the main screen, it needed to meet the expectations set by the design set out in the design stage, for this, it needed to be one main screen that had its inner model changed the best way to do this in WinForms was to use user controls. A user control is a component that allows users to create a custom control comprised of pre-existing controls.

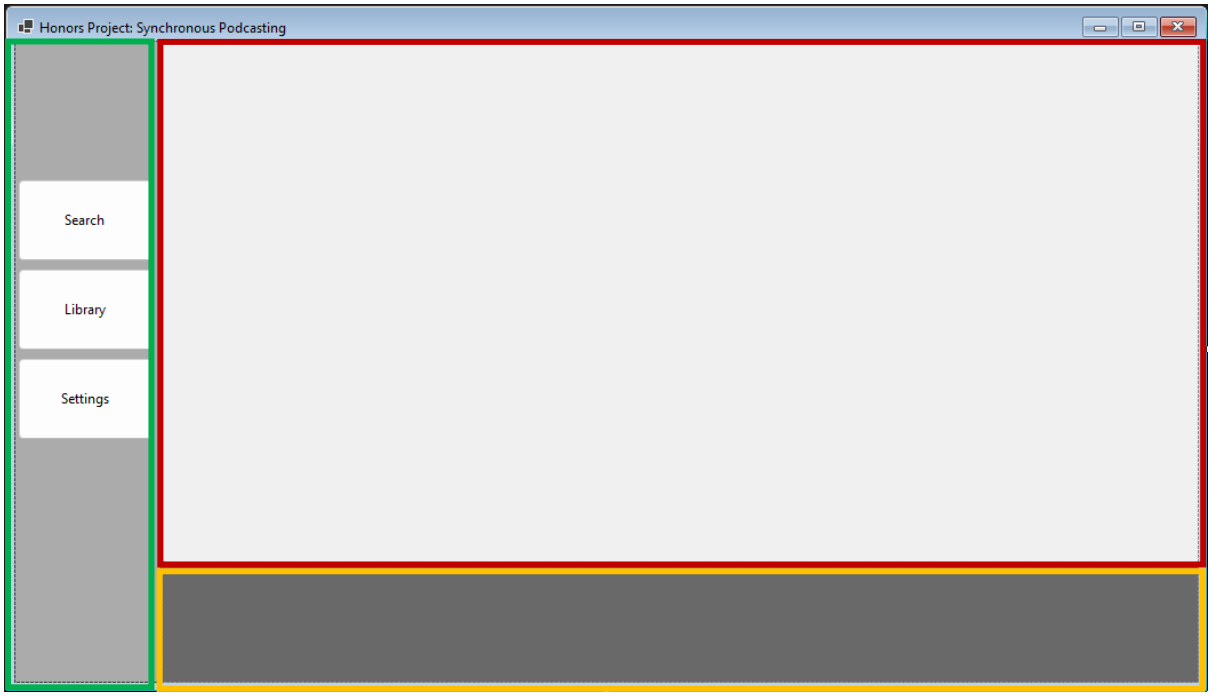

Figure 22 Main Form

Here is the final implementation for the main form, it is comprised of 3 major sections, first the menu bar on the left of the screen, highlighted in green, allows users to navigate between the 3 main screens Search, Library, and settings this is a panel containing 3 buttons. The media controller panel, highlighted in yellow, will be filled when media is queued it allows users to skip through media pause and play. The main screen, highlighted in red, is the main screen that changes to one of the 3 main screens.

Figure 23 Library User Control Design

The library screen contains a panel that is used to dynamically display podcasts stored in a user's library. When creating this user control, it was important to maintain simplicity as this would be the user control that the user is greeted with when launched into the main application.

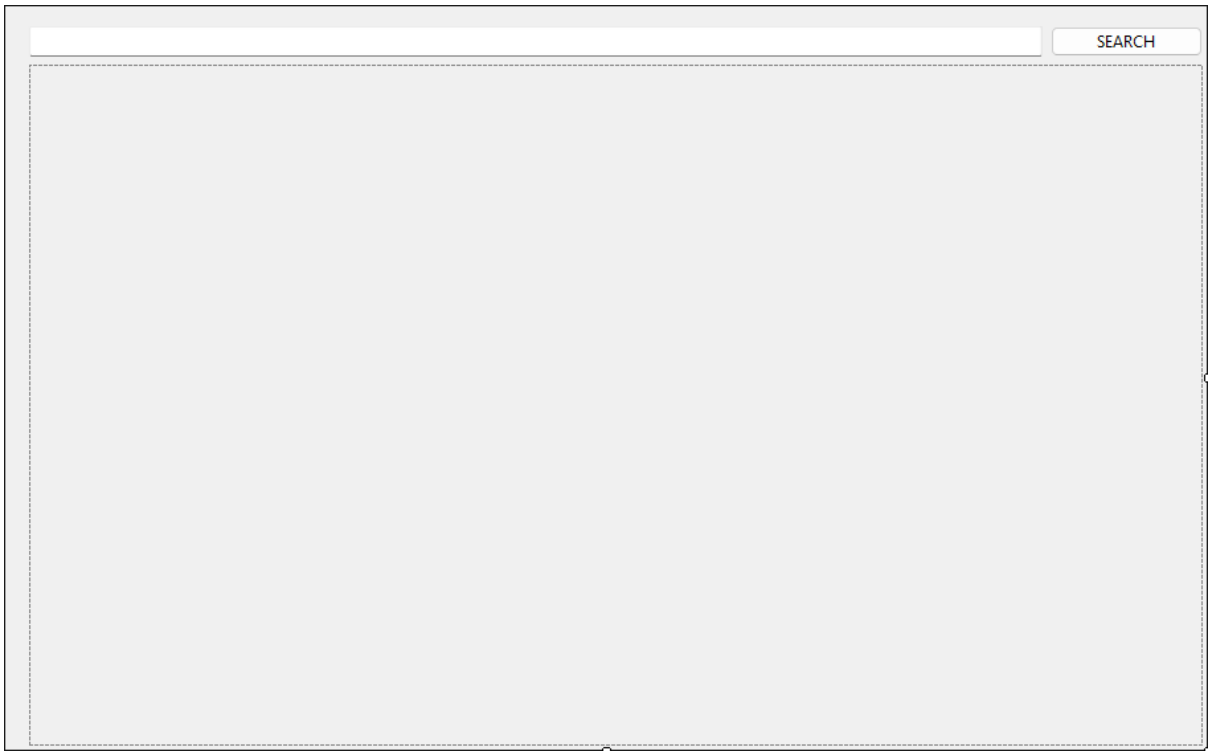

Figure 24 Search User Control Design

The search user control is comprised of a text box button and panel, like the library user control the panel allows for the panel to be filled with podcasts related to the search term entered by the user. When creating this user control a good user experience was key for example there is a search button however a user can press enter to search, this was done to make the page intuitive.

```
public partial class SearchUC : UserControl
    {
        private static SearchUC _instance;
        private static Panel _targetPanel;


        public static SearchUC Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new SearchUC(_targetPanel);
                return _instance;
            }
        }

        public static void InitializeWithPanel(Panel targetPanel)
        {
            if (_targetPanel == null)
                _targetPanel = targetPanel;
        }

        public SearchUC(Panel targetPanel)
        {
            InitializeComponent();
            SearchTextBox.KeyPress += KeyPressReg;
        }

    }
```

Figure 25 Code used in user control to get an instance and create a user control

```
private void openSearch()
        {
            if (!windowPanel.Controls.Contains(SearchUC.Instance))
            {
                SearchUC.InitializeWithPanel(windowPanel);
                windowPanel.Controls.Add(SearchUC.Instance);
                SearchUC.Instance.Dock = DockStyle.Fill;
                SearchUC.Instance.BringToFront();
            }
            else
            {
                SearchUC.Instance.BringToFront();
            }
        }
```

Figure 26 Code to switch user control is displayed in the main form.

Figures 25 & 26 display the code that is used when switching between main screens, each screen is a singleton that contains one instance of itself as well as the panel it should have control over if an instance of the user control already exists it will return it if not It will create a new one, Initialize with panel tells the user control what panel it will be controlling, for this example, it will be the panel highlighted in red in figure 22. When the open command corresponding to the window is called it checks if the panel already contains an instance of that user control if it does it will bring it to the front if not it will initialize the correct user control panel and add it to the main window panel.

5.1.2 Curating and creating podcasts.

Podcast lookup begins with the user inputting a search term.

```
public static string lookupPodcast(string podcastName)
        {
            // Encode the podcast name to handle spaces and special characters
            string encodedName = HttpUtility.UrlEncode(podcastName);

            // Search for the podcast on iTunes and get the first result
            string url =
$"https://itunes.apple.com/search?term={encodedName}&entity=podcast";
            string responseString = new
HttpClient().GetStringAsync(url).Result;
            // Deserialize the JSON response into strongly-typed objects
            var options = new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            };
            var response =
JsonSerializer.Deserialize<Models.SearchResponse>(responseString, options);

            // Get the URL of the first search result
            if (response.Results.Length > 0)
            {
                return JsonSerializer.Serialize(response);
            }
            else
            {
                Console.WriteLine($"No results found for '{podcastName}'");
                return "";
            }
        }
```

Figure 27 iTunes lookup Method

From there the Figure 27 lookup podcast method is called, it takes the podcast name as a parameter and encodes it so that it is suitable for an HTTP request, this is the practice of removing special characters, for example, the '&' symbol would be replaced with '%26', from there the request is sent to the iTunes Search API and the response is stored in a new variable response if there is a valid response it will be serialized and returned if not an empty string will be returned.

The returned result is parsed and stored in a list of strings, to make it easier to iterate over, and is iterated over in a loop calling the podcast builder function which takes an RSS feed as an input and

creates a podcast object.

```
public class Podcast
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public string Author { get; set; }
        public string ImageURL { get; set; }

        public List<Episode> Episodes { get; set; }

        public Podcast(
            string name,
            string description,
            string author,
            string imageURL)
        {
            Name = name;
            Description = Regex.Replace(
                description, "<.*?>", string.Empty);
            Author = author;
            ImageURL = imageURL;
            Episodes = new List<Episode>();
        }



        }
    }
```

Figure 28 Podcast class

Figure 28 is the podcast class it is one of the most relevant object classes in the project it is used to store all the information.

```
public static async Task<Podcast> BuildPodcastFromRSS(string rssFeedUrl)
        {
            try
            {
                using (var client = new HttpClient())
                {
                    var response = await client.GetAsync(rssFeedUrl);
                    response.EnsureSuccessStatusCode();

                    var xml = await response.Content.ReadAsStringAsync();
                    var doc = XDocument.Parse(xml);

                    var itunesNS = XNamespace.Get(
                        "http://www.itunes.com/dtds/podcast-1.0.dtd");

                    var channel = doc.Descendants(
                        "channel").FirstOrDefault();
                    var podcastName = channel.Element("title").Value;
                    var podcastDescription = channel.Element("
                        description").Value;
                    var podcastAuthor = channel.Element(
                        itunesNS + "author").Value;
                    var podcastImageURL = channel.Element(
                        itunesNS + "image")?.Attribute("href")?.Value;

                    var podcast = new Podcast(
                        podcastName,
                        podcastDescription,
                        podcastAuthor,
                        podcastImageURL);

                    var items = doc.Descendants("item");
                    foreach (var item in items)
                    {
                        var episodeTitle = item.Element("title").Value;
                        var episodeLink = item.Element("link")?.Value;
                        var episodeDescription = item.Element(
                            "description")?.Value;
                        var pubDate = item.Element("pubDate").Value;
                        var episodeDuration = item.Element(
                            itunesNS + "duration")?.Value;
                        var enclosure = item.Element("enclosure");
                        var episodeURL = enclosure.Attribute("url").Value;

                        var episode = new Episode(
                            episodeTitle, episodeLink,
                            episodeDescription,
                            episodeDuration,
```

```
                            episodeURL,
                            pubDate,
                            podcastImageURL);
                        podcast.Episodes.Add(episode);
                    }
                    return podcast;
                }

            }catch (Exception ex)
            {
                return null;
            }
        }
```

Figure 29 Method that builds podcast from an RSS feed.

As shown in Figure 29 the BuildPodcastFromRSS method receives and parses a podcast's RSS feed from the URL supplied, the method will return a Podcast object that contains details about a podcast and its episodes. It begins by sending an HTTP GET request to fetch the RSS feed, it then parses the XML data extracting information such as a podcast's title, description, thumbnail link and, episodes.

5.1.3 Dynamically filling the search panel with podcasts

In many places in this project, it is necessary to fill a panel with podcasts at runtime for this some features have been developed to assist with this. For this example, it will display how to search window will fill with podcasts based on a user search term.

Figure 30 Search Response User Control Design

For the creation of the search response user control, it was important to create a simple intuitive interface because of this the user control is only comprised of a picture box for the thumbnail two labels for the author and title and a button that surrounds the user control that will take users to the PodcastInfoUC

```
 private async Task AddNewUserControl(Podcast p)
        {
            SearchResponseUC newUserControl = new SearchResponseUC(p,
_targetPanel);

            int top = SearchResponsePanel.Controls.Count *
newUserControl.Height;
            newUserControl.Location = new Point(0, top);

            SearchResponsePanel.Controls.Add(newUserControl);
        }

        private async void Search() {
            SearchResponsePanel.Controls.Clear();
            Debug.WriteLine($"Searching for podcasat {SearchTextBox.Text}");
            string resultString =
ItunesLookup.lookupPodcast(SearchTextBox.Text);

            List<FeedResult> results = FeedResultParser.Parse(resultString);

            int numSearches = Math.Min(results.Count, 10);
            for (int i = 0; i < numSearches; i++)
            {
                var result = results[i];

                Podcast p = await Podcast.BuildPodcastFromRSS(result.FeedUrl);


                if (p != null && p.Name != null && p.ImageURL != null &&
p.Author != null)
                {

                    await AddNewUserControl(p);
                }
            }
        }
```

Figure 31 Code for creating and displaying user controls.

These two methods control all the dynamic creating and displaying of SearchResponseUC's to the search window panel, the AddNewUserControl method is an asynchronous method that takes a Podcast as an argument. A new SearchResponseUC is created with the podcast information and a target panel as input. It then calculates the position top for the new user control based on the number of existing controls in SearchResponsePanel and the height of the new control. The new user control is placed at this position within the panel and is then added to the SearchResponsePanel.

The search method clears any existing SearchResponseUC's from the user panel and uses to iTunes lookup to search for a user's podcast search term. The method then iterates over up to 10 search results. For each result, it builds a Podcast object from the feed URL associated with the search result. It then calls the AddNewUserControl method to add a SearchResponseUC user control to the search window panel, displaying the podcast's information. The AddNewUserControl method is called asynchronously, and the Search method waits for it to complete before processing the next search result. Initially, it was filling all the results that a search returned however this had to be capped at 10 as too many search results would use a lot of memory and be detrimental towards performance.

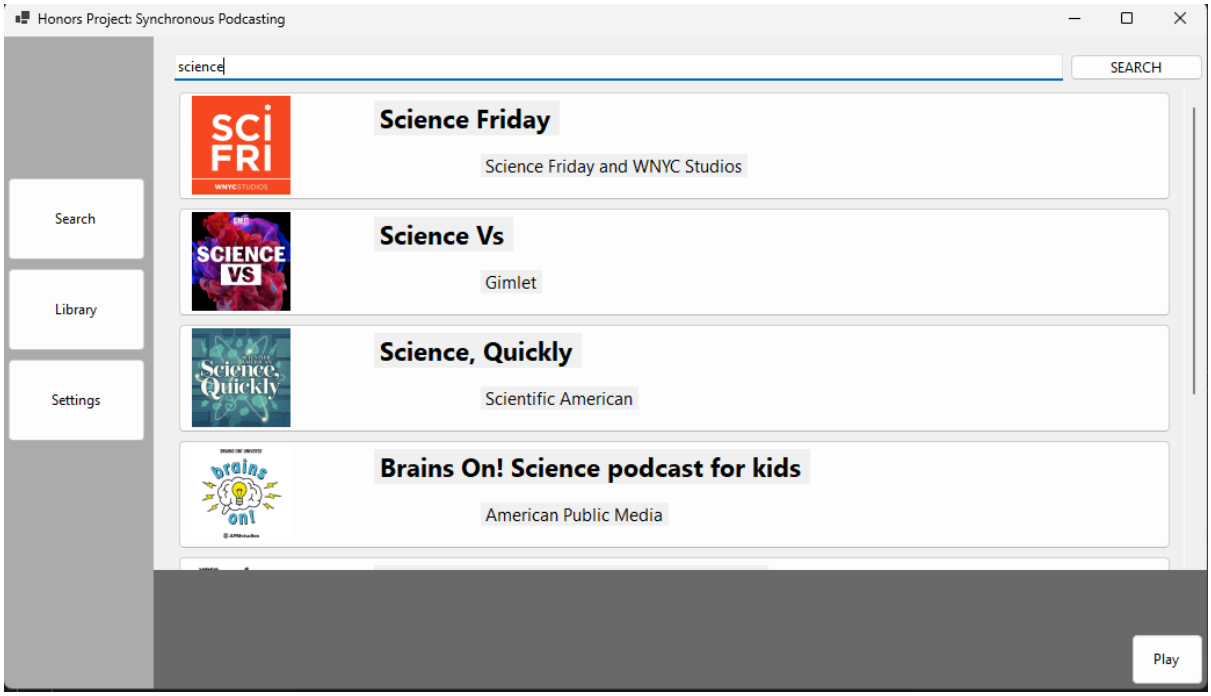

Figure 32 Working version of a filled search window

5.1.4 Synchronizing podcasts

Synchronisation serves as one of this project's key components. To start it was necessary to have a place to store all the podcasts that a user had in their library.

```
private static List<Podcast> podcastLibrary = new List<Podcast>();

        public static void addPodcastToLibrary(Podcast p) {

            if (!podcastLibrary.Contains(p)) {
                podcastLibrary.Add(p);
                saveLibrary();
            }
        }
```

Figure 33 Podcast library code

For this, a List of podcast objects was used, as it allows for podcasts to easily be added as needed and would not use too much memory-affecting preference. The List is private as all editing to the list will be done through the addPodcastToLibrary method; the List is static as there should only be one instance of it per client.

Next, there was a need to develop a way to cache and access the podcast data locally.

```
public static void loadLibrary()
    {

        string jsonString;
        if ((jsonString = Database.SendGetRequest(
            "https://localhost:44390/api/data", User.APIKey) )!= null)
        {
            podcastLibrary = System.Text.Json.JsonSerializer.Deserialize<
            List<Podcast>>(jsonString);
        }
        else {
            if (File.Exists("myLibrary.json"))
            {
                jsonString = File.ReadAllText("myLibrary.json");

                podcastLibrary = System.Text.Json.JsonSerializer.Deserialize<
                List<Podcast>>(jsonString);
            }
        }

    }


public static void saveLibrary()
    {
        var library = PodcastLibrary.getPodcastLibrary();

        string jsonString = System.Text.Json.JsonSerializer.Serialize(library,
            new JsonSerializerOptions
            {
                WriteIndented = true
            });

        if (File.Exists("myLibrary.json"))
        {
            File.Delete("myLibrary.json");
        }

        using (StreamWriter sw = new StreamWriter("myLibrary.json"))
        {
            sw.Write(jsonString);
        }

        Database.SendPostRequestJSON(
            "https://localhost:44390/api/data", User.APIKey, "myLi-
brary.json");
    }
```

Figure 34 Library loading and saving JSON library data.

For load library first, the method attempts to request the backend with the user API key if it cannot I will return null, and the method will get the information stored in myLibrary.json and de-serializer it and put it in the podcast library, if it can make a connection, it will set podcast library to a deserialized version of what the backend returns.

For the saved library the library is serialized into a JSON string written to a file called myLibrary.json this decision was made as it was seen as important to store a local copy to the drive that could be accessed. Once the local copy is cached there is a post request to the backend server.

```
public static void SendPostRequestJSON(string url, string apiKey, string
jsonFilePath)
        {
            try
            {
                var request = (HttpWebRequest)WebRequest.Create(url);
                request.Method = "POST";
                request.ContentType = "application/json";

                var jsonData = File.ReadAllText(jsonFilePath);
                var postData = new
                {
                    APIKey = apiKey,
                    LibraryData = jsonData
                };
                var json =
Newtonsoft.Json.JsonConvert.SerializeObject(postData);
                var data = Encoding.UTF8.GetBytes(json);
                request.ContentLength = data.Length;

                using (var stream = request.GetRequestStream())
                {
                    stream.Write(data, 0, data.Length);
                }

                var response = (HttpWebResponse)request.GetResponse();
                var responseString = new
StreamReader(response.GetResponseStream()).ReadToEnd();
                Console.WriteLine(responseString);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
        }
```

Figure 35 Method for updating podcast library in the backend

The SendPostRequestJSON method takes the URL, user API key and, the file path for the JSON as parameters and creates an HTTP request of type get with all the data from the JSON file and sends it to the backend server.

```
public static string SendGetRequest(string url, string apiKey)
        {
            try
            {
                var request = (HttpWebRequest)WebRequest.Create(
                    $"{url}/{apiKey}");
                request.Method = "GET";

                var response = (HttpWebResponse)request
                .GetResponse();
                var responseString = new StreamReader(
                    response.GetResponseStream()).ReadToEnd();

                if (string.IsNullOrWhiteSpace(responseString))
                {
                    return null;
                }
                else
                {
                    return responseString;
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
                return null;
            }
        }
```

Figure 36 Method for getting podcast library from the backend server.

The SendGetRequest method takes a URL and API key as parameters and sets an HTTP request of type to get to the backend server with the API key this server will return the podcast library stored in the backend server.

```
public static void updateEpisodeProgress(string episodeFileName, int progress)
{
            foreach (Podcast podcast in podcastLibrary)
            {
                foreach (Episode episode in podcast.Episodes)
                {
                    if (episode.FileName == episodeFileName)
                    {
                        episode.duartionProgres = progress;
                        return;
                    }
                }
            }
        }

public static int getEpisodeProgress(string episodeFileName)
{

            foreach (Podcast podcast in podcastLibrary)
            {
                foreach (Episode episode in podcast.Episodes)
                {
                    if (episode.FileName == episodeFileName)
                    {
                        return episode.duartionProgres;
                    }
                }
            }

            return 0;
}
```

Figure 37 Methods for updating episode progress.

The updateEpisodeProgress method takes two parameters, the episode's file name and the progress in seconds through the episode it will then loop through the episodes of each podcast until it finds the correct episode then it changes the episode duration.

The getEpisodeProgress method takes the episode file name as a parameter and loops through the episodes of each podcast until it finds the correct episode and returns its duration progress.

For these methods, the episodes file name was chosen as an identifier as due to restraints created by services that host podcasts the episodes need to have no duplicated names.

When all these methods are put together It creates a system that allows users to synch their podcast library and episode progress across multiple devices while allowing for offline usage.

5.1.5 Episode downloading and playback

```
private static async Task DownloadMP3Async(string url, string filePath)
        {
            using (WebClient client = new WebClient())
            {
                await client.DownloadFileTaskAsync(url, filePath);
            }
        }
```

Figure 38 Episode mp3 downloading method.

The DownloadMP3Asynch is an asynchronous method that is designed to download an episode's mp3 file for playback, for parameters it takes the URL address of the mp3 and the file path that the file is to be saved to. The method makes use of the .NET WebClient class to handle the downloading of episode files.

5.1.6 Backend server implementation

For this project's backend implementation, an ASP.NET Web API was used.

```
// GET api/<DataController>
[HttpGet("{APIKey}")]
public IActionResult Get(string APIKey)
{
    return Ok(db.GetDataByUUID(APIKey));
}

// POST api/<DataController>
[HttpPost]
public IActionResult Post([FromBody] DataInput dataInput)
{
    db.AddData(dataInput.APIKey,dataInput.LibraryData);
    return Ok();
}
```

Figure 39 Backend server data controller

```
public class DataDB
    {
        private static DataDB _instance;
        private readonly IMongoDatabase _database;

        private DataDB(string connectionString, string databaseName)
        {
            var client = new MongoClient(connectionString);
            _database = client.GetDatabase(databaseName);
        }

        public static DataDB GetInstance()
        {
            if (_instance == null)
            {
                _instance = new DataDB("mongodb://localhost:2717/", "user");
            }

            return _instance;
        }

        public IMongoCollection<DataModel> GetDataCollection()
        {
            return _database.GetCollection<DataModel>("data");
        }

        public IEnumerable<DataModel> GetAllData()
        {
            var collection = GetDataCollection();

            var result =   collection.Find(_ => true).ToEnumerable();
            return result;
        }

        public void AddData(string uuid, string json)
        {
            var collection = GetDataCollection();
            var existingData = GetDataByUUID(uuid);

            var encryptedData = EncryptionHelper.EncryptString(json);

            if (existingData != null)
            {
                var filter = Builders<DataModel>.Filter.Eq(x => x.APIKey,
uuid);
                var update = Builders<DataModel>.Update.Set(x => x.JSON,
json);
                collection.UpdateOne(filter, update);
```

```
            }
            else
            {
                var data = new DataModel { APIKey = uuid, JSON = encryptedData
};
                collection.InsertOne(data);
            }
        }

        public string GetDataByUUID(string uuid)
        {
            var collection = GetDataCollection();
            var datamodel = collection.Find(x => x.APIKey ==
uuid).FirstOrDefault();
            var result = EncryptionHelper.DecryptString(datamodel?.JSON);
            return result;
        }
    }
```

Figure 40 Class for accessing the database on the backend server.

On the backend server, the data controller is kept simple to increase readability, the DataDB database access class is a singleton as it ensures efficient resource management, providing a thread-safe interface for database management.

5.2 Testing

5.2.1 Unit testing

```
[TestMethod]
0 references
public void WriteToBackendTest()
{
    Database.SendPostRequestJSON("https://localhost:44390/api/data", "testApiKey", "D:\\Source\\Repos\\HonorsFinal\\UnitTests\\bin\\Debug\\net7.0\\test.json");

    string response = Database.SendGetRequest("https://localhost:44390/api/data", "testApiKey");

    Assert.AreEqual(response, "OK");
}
```

Figure 41 Unit Test 1

```
[TestMethod]
0 references
public void GetFromBackend()
{
    string response = Database.SendGetRequest("https://localhost:44390/api/data", "testApiKey");

    Assert.AreEqual(response, "OK");
}
```

Figure 42 Unit Test 2

```
[TestMethod]
0 references
public void iTunesLookup()
{
    string response = ItunesLookup.lookupPodcast("science");

    Assert.IsNotNull(response);
}
```

Figure 43 Unit Test 3

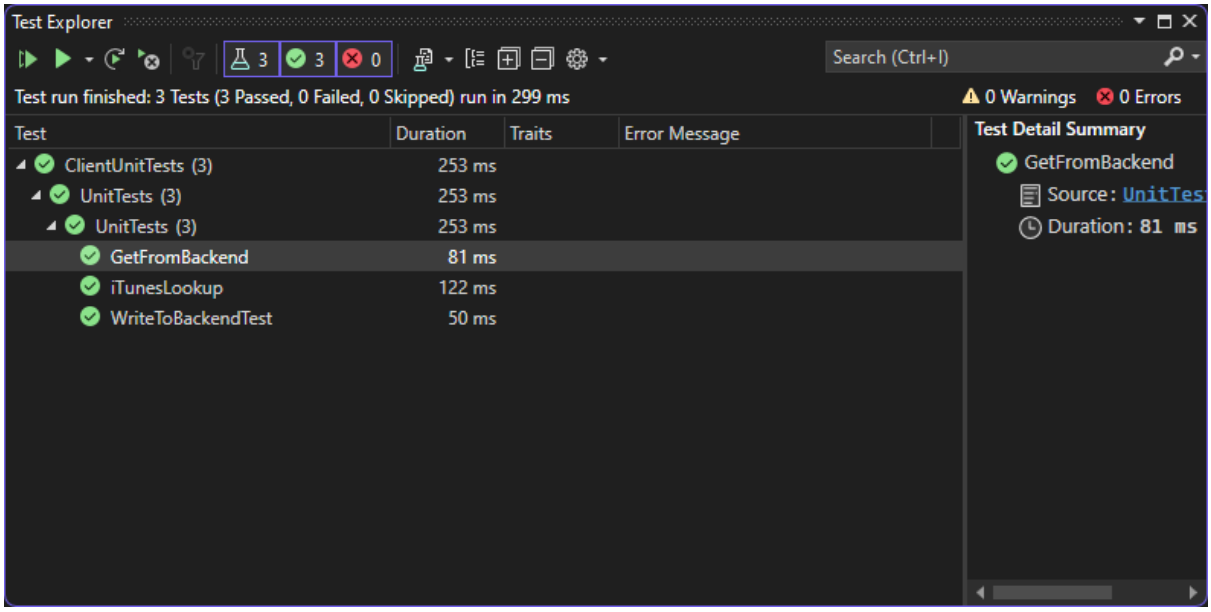

Figure 44 Proof for tests

5.2.2 Integration testing

These are tests that evaluate the flow of the major system procedures.

Podcast search test

This test is designed to ensure that the product has the functionality to create a podcast object based on multiple methods and classes working together. For this test the name of a chosen podcast will be entered the user should then be able to find the podcast in the list of returned podcasts choose it and add it to their library users should also be able to see relevant information about a podcast such as description author and all podcast episodes.

For this test, the podcast “Myths and Legends” was selected.

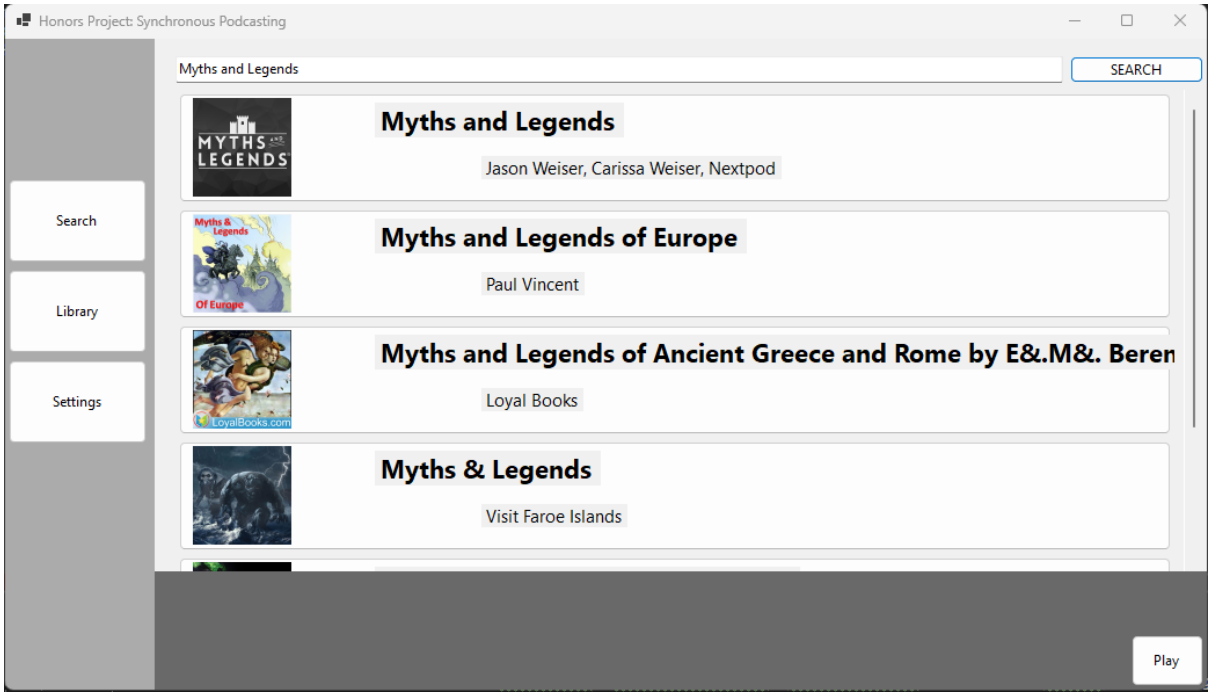

Figure 45 Support for test case

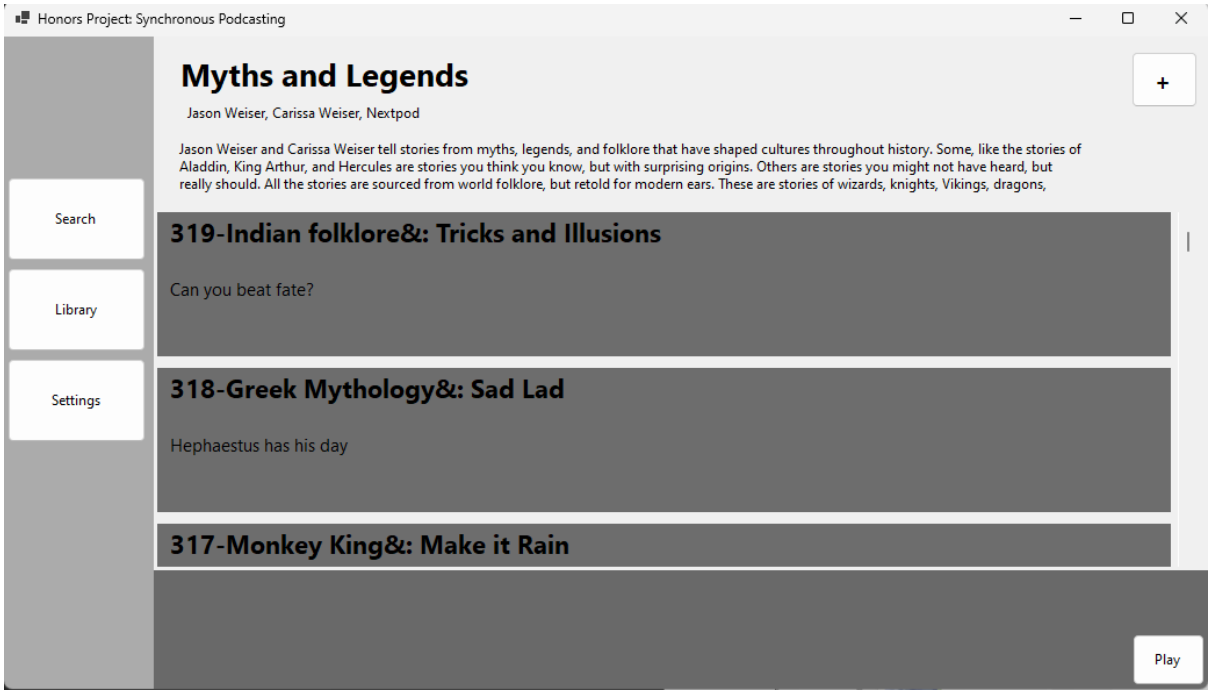

Figure 46 Support for test case

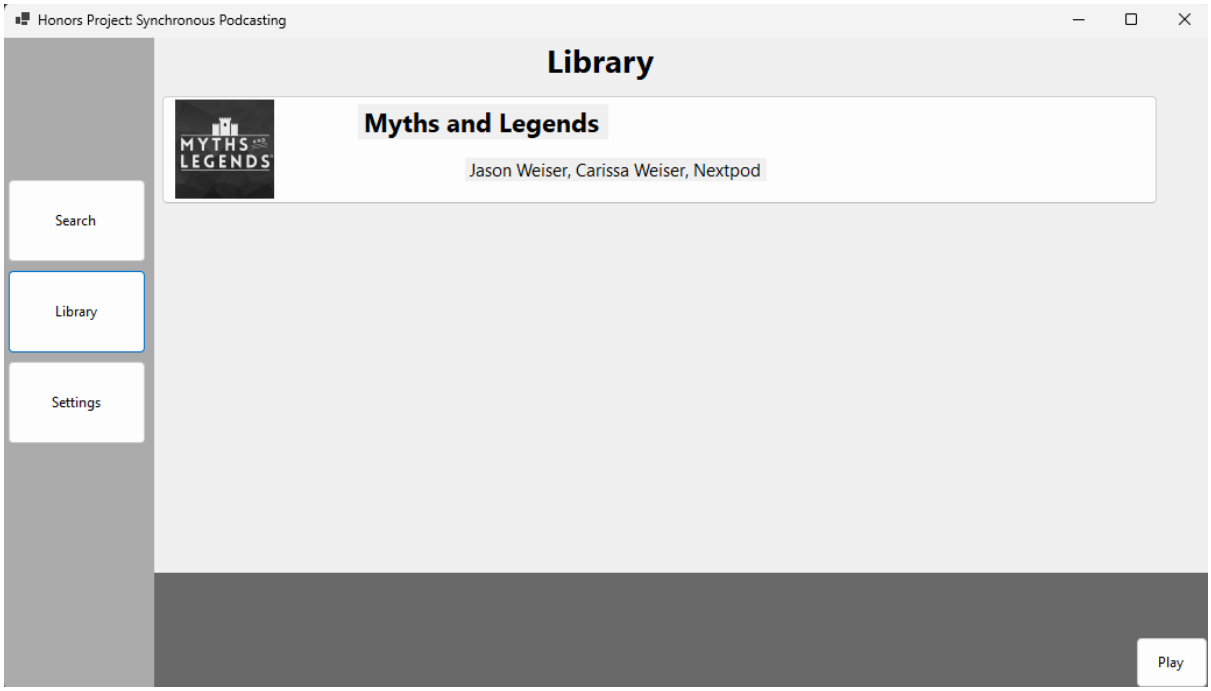

Figure 47 Support for test case

As seen from Figures 41, 42 & 43 this test was completed, the user was able to successfully find the podcast, see all of its relevant information and add it to their library. This test was a success.

Episode retrieval & playback test

This test is created to ensure that a user can download an episode begin playback to a certain point and continue from where they left off. For this test, a user will download an episode watch to a certain point close the application, reopen it and then continue from where they left off.

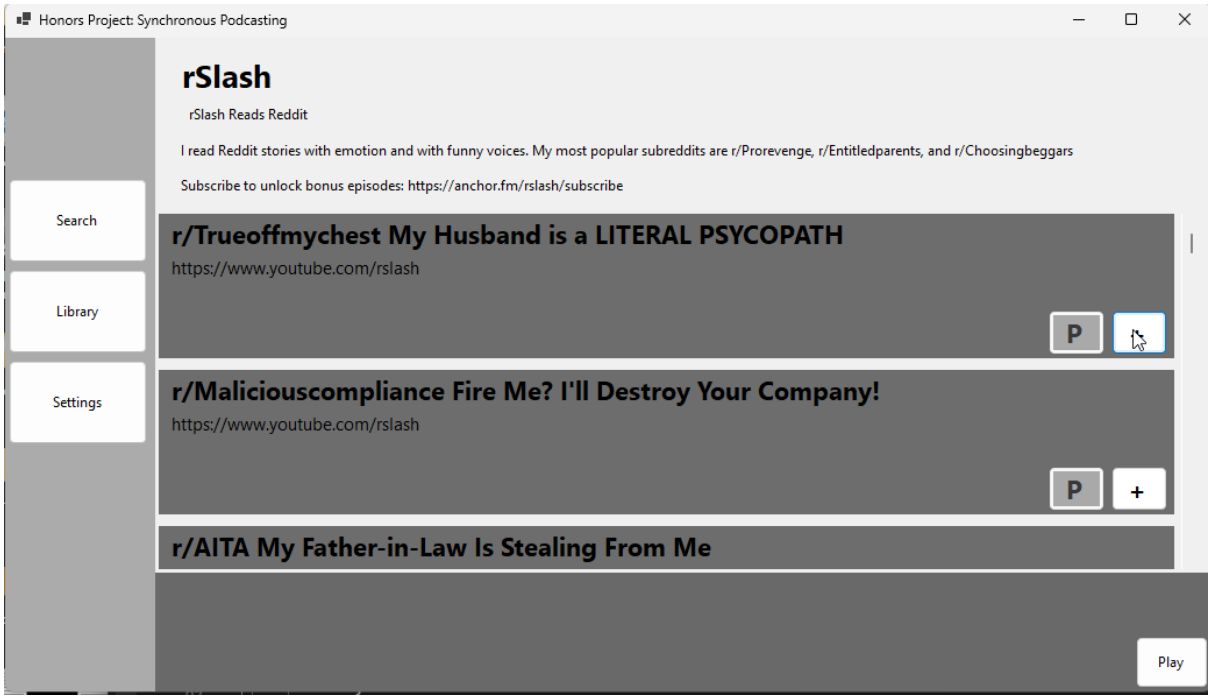

Figure 48 Support for test

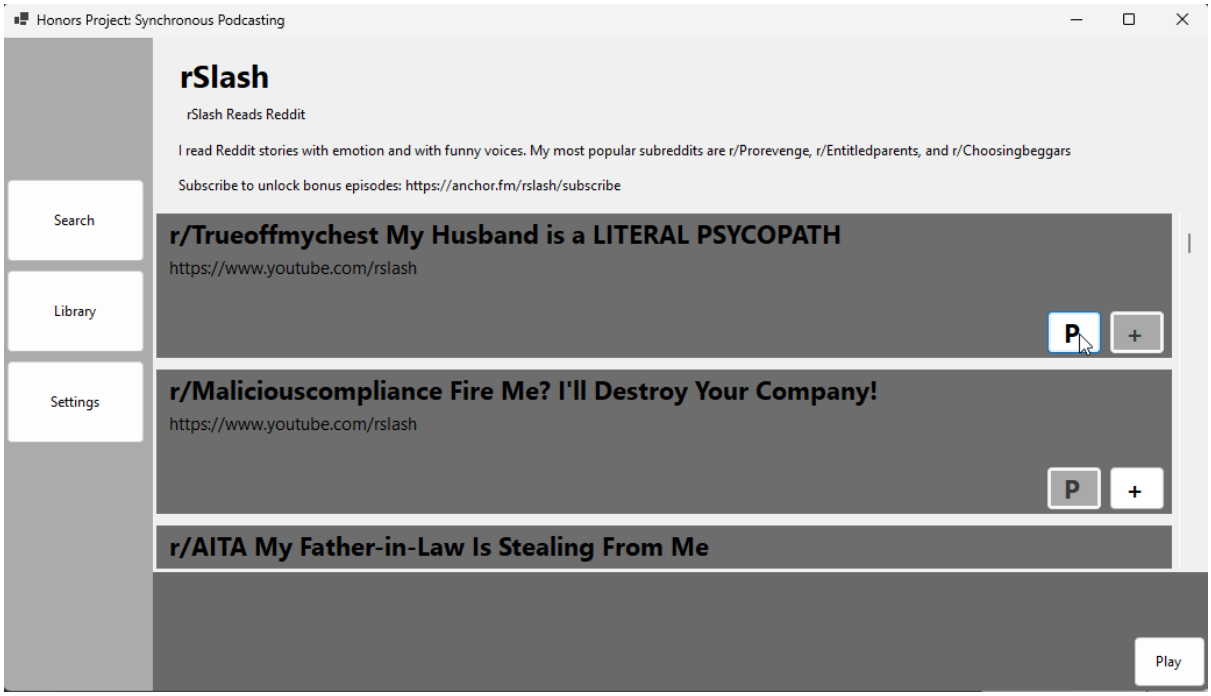

Figure 49 Support for test

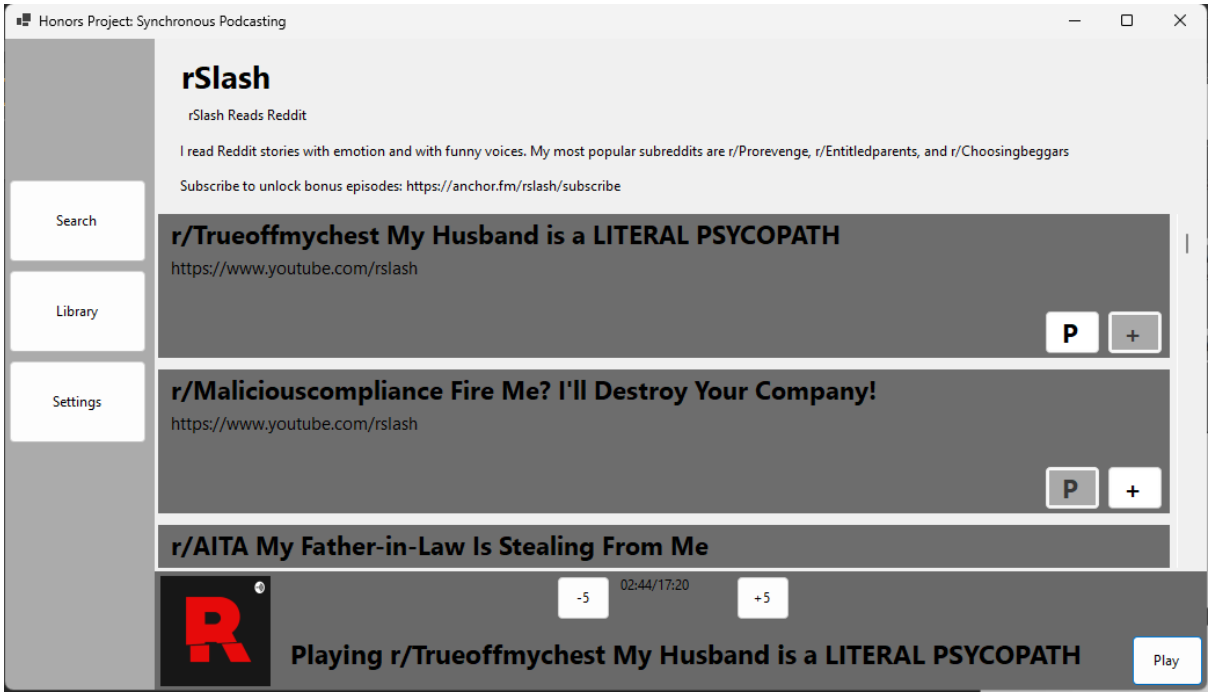

Figure 50 Support for test

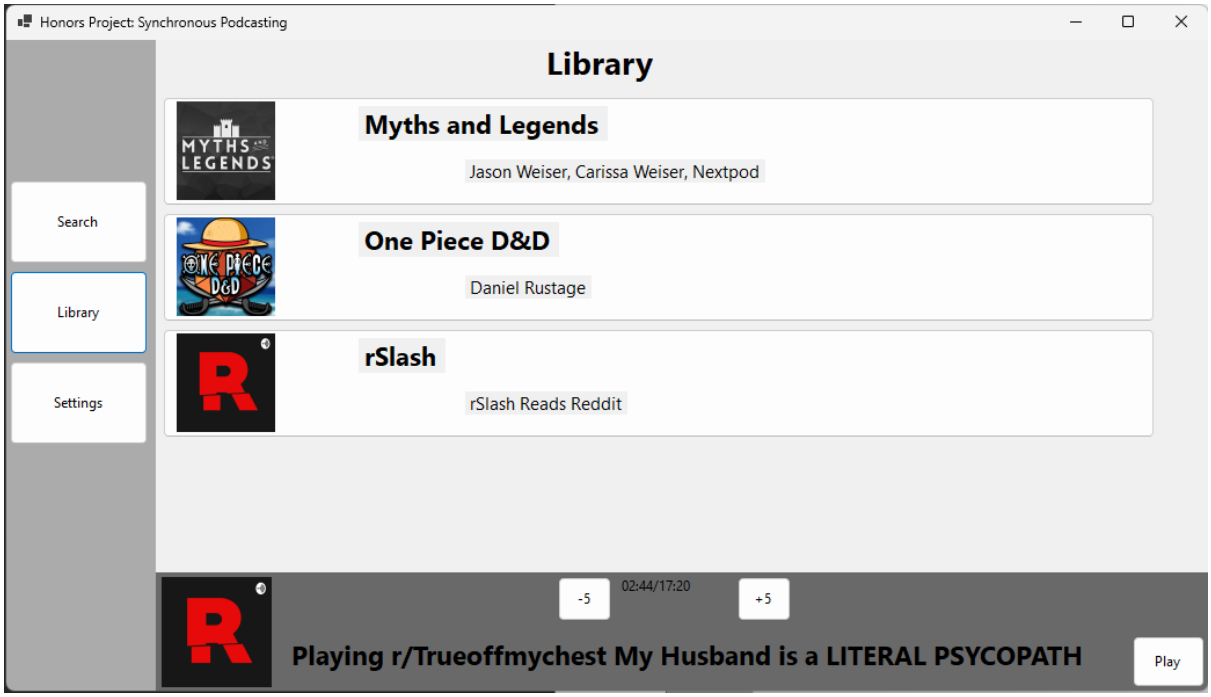

Figure 51 Support for test

As seen in the figures above the application was able to download and play a podcast episode close and resume from where it was this test was a success.

6 Evaluation and discussion of results

Delete the red paragraphs and replace this one with your content (use the “Normal” paragraph style).

6.1 Objectives Met

This section will go over the aims and objectives outlined in section 1.2:

Objective 1 - The successful creation of a secure and scalable database and server system for managing podcasts has been realized.

• We identified and thoroughly evaluated suitable database management systems and server technologies, leading to a selection that aptly meets the system's requirements.

• A thoughtful design of the database structure and server architecture now effectively supports user accounts and podcast data.

• The implementation of the database and server system leveraged Docker images, enhancing deployment, scalability, and maintenance.

• The system is not only secure but also scalable, providing a strong foundation for potential future expansion.

This primary objective was all about establishing a reliable database and server system to handle user accounts and podcast data. Its successful completion is reflected in the system's enhanced ability to manage user authentication and provide secure, efficient access to podcasts.

Objective 2 - Successful design and development of a RESTful backend server for managing user podcast data and enabling cross-platform playback have been achieved.

• Thorough exploration and understanding of RESTful services led to their effective application for data management and retrieval.

• The backend server was designed to efficiently handle user podcast data, including tracking the episodes they've watched and their playback progress.

• RESTful APIs were used in the implementation of the server solution, ensuring seamless data access across multiple devices.

• Rigorous testing and fine-tuning were conducted to optimize performance and enhance user experience.

The second objective focused on designing and developing a RESTful backend server to effectively manage user podcast data and ensure cross-platform playback. Research into efficient data management using RESTful services and the careful design of the server have resulted in seamless data access across various devices. The server solution, implemented with RESTful APIs, has ensured cross-platform compatibility, significantly enhancing user satisfaction. The effectiveness of this objective is reflected in the server's capability to manage and provide user podcast data while delivering a seamless listening experience across platforms.

Objective 3 - A user-friendly Windows application for podcast discovery and playback has been successfully developed.

• We thoroughly explored best practices for creating user-friendly graphical user interfaces and applied them to our design process.

• The interface and features of the Windows application were designed with a strong emphasis on improving user experience.

• We brought the design to life, implementing the application with all the necessary features and functionalities.

• A cycle of rigorous testing and adjustments was completed, ensuring top-notch usability and performance of the application.

The third objective was all about crafting a Windows application that facilitates easy discovery, access, and playback of podcasts. By gaining a deep understanding of user interface design principles and focusing on a seamless user experience, we designed and implemented an application that hits the mark. The successful completion of this objective is evident in the application's usability, functionality, and performance, leading to a more enjoyable experience for podcast listeners.

6.2 Requirements Met

This section is meant to compare the final project to the outline set out in the requirements section.

6.2.1.1 Windows Application

The application was thoughtfully engineered to allow users to sign in with their accounts, ensuring seamless tracking of podcast subscriptions and playback progress. The user interface is intuitive and user-friendly, making podcast subscription management, access to podcast information, and media playback control a breeze. The application boasts all the necessary media control options such as play, pause, skip, and volume adjustment. To keep users up-to-speed with their preferred podcasts, an automatic download feature has been installed, downloading the latest episodes from subscribed podcasts.

6.2.1.2 Back End Server

The backend services manage user login details and podcast data. A RESTful server has been set up, allowing the client app to retrieve the essential information it needs. The server's configuration has been fine-tuned to foster effective communication between the client app and the database, ensuring a smooth user experience.

6.2.1.3 Database

The database, carefully constructed and now fully operational, securely stores user account details, podcast subscription data, and playback progress. Working in harmony with the backend services, it feeds the client app with the essential data. Its successful implementation is a cornerstone in delivering a seamless podcast management experience for the users. This accomplishment serves as a testament to the overall system's effectiveness.

6.2.1.4 Integration of External API

The application now expertly uses external APIs, including the iTunes Search API and RSS feeds, to access and showcase podcast details. The integration of these APIs into the app broadens the

spectrum of podcasts available to users, making it simple for them to discover and subscribe to fresh content.

6.3 User product evaluation

Quotes are taken from a user test group:

User 1: "Pros: Streamlined layout, good error verification, search option for podcasts with keywords is innovative, volume slider is needed, can finish podcasts from where you left off is useful, podcasts can be saved to the library

Cons: No header label for username and password,"

This shows user 1 overall had a positive experience with the product and believed that it was effective in its objective innovatively and seamlessly. However, their criticism show that there is a lack of intuitiveness in the design as it lacked some crucial labels.

User 2: "Pros: good layout, the ability to save a user's library to an account is good if you want to use the app on another device, appreciate that podcast episodes will continue from where you left off.

Cons: design lacks colour and can be bland to look at"

User 2 found the layout of the app to be done well however the colour pallet should have been less bland.

User 3:" Like the ability to search on a wide variety of search terms such as title, genre and author, the design is mostly intuitive and I could tell what to do almost instantly, good layout for the podcasts in the library and search screen.

Cons: when going from search to a podcast there is no back button to go back to search"

User 3 appreciated the implementation of the search and design however they felt that some aspects of the UI left more to be desired.

The app overall had positive feedback from the evaluation group with a general theme that the implementation of the synchronous aspect of the project was effective and the layout of the app was mostly good.

7 Conclusion

The waterfall method used through development was effective as the clear structure of waterfall allowed for development to be smooth however when new objectives were created during development the rigidity of waterfall made it difficult to implement these features, compared to a methodology such as agile is designed to be flexible and adaptable and would allow for changes to be made mid development without disrupting the project.

In conclusion, although some aspects of the project had to be cut the overall goal of the creation of a synchronous podcasting application that would allow users to conveniently access podcast media across multiple devices whilst their library and progress are saved, was met and the product that was delivered allows for other systems to later be built on.

7.1 Personal Reflection

The project had good initial planning however during development there were times when tunnel visioning on a feature caused massive delays to development, the use of weekly evaluations would have helped lessen the chance that development would see major delays.

The use of project management tools such as Trello mixed with distribution tools such as Git was wise in the development of this project as it allowed for application development progress to be easily managed and tracked and allowed features to be developed in a way that would not negatively impact the project in the case of a catastrophic failure.

7.2 Further work

Further work/maintenance that could be done on this project are. The addition of a feature that would allow for the application to download the newest episode in a podcast that a user has not seen, would increase convenience for users. Additionally, a feature that would delete a podcast episode if it has not been viewed in a month could help alleviate storage requirements imposed by the application. Finally, a mobile app would be beneficial as it would allow a wider audience to access the project.

Based on user feedback it would also be beneficial to make some tweaks to the UI as it is currently seen as bland, changes that could be made are the inclusion of icons and more colours, this could make the product feel more human and nicer to use.

References

C# introduction (no date) Introduction to C#. Available at: https://www.w3schools.com/cs/cs_intro.php (Accessed: April 20, 2023).

Daka, E. and Fraser, G. (2014) “A survey on unit testing practices and Problems,” 2014 IEEE 25th International Symposium on Software Reliability Engineering [Preprint]. Available at: https://doi.org/10.1109/issre.2014.11.

Datadog (2022) Containerized Applications Overview, What Are Containerized Applications? Available at: https://www.datadoghq.com/knowledge-center/containerized-applications/#:~:text=The%20purpose%20of%20containerization%20is,consistent%20from%20host%20to%20host.

GitLab (no date) What is version control?, GitLab. GitLab. Available at: https://about.gitlab.com/topics/version-control/#the-basics-of-version-control.

Introduction to C# windows forms applications (no date) GeeksforGeeks. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-to-c-sharp-windows-forms-applications/ (Accessed: April 20, 2023).

ITunes search API (no date) iTunes Search API - Apple Services Performance Partners. Available at: https://performance-partners.apple.com/search-api (Accessed: April 20, 2023).

ITunes search API (no date) iTunes Search API - Apple Services Performance Partners. Available at: https://performance-partners.apple.com/search-api.

Rick-Anderson (no date) ASP.NET Overview, Microsoft Learn. Available at: https://learn.microsoft.com/en-us/aspnet/overview (Accessed: April 21, 2023).

TerryGLee (2023) Parameter info, list members, and quick info in Intellisense - Visual Studio (windows), Parameter info, list members, and quick info in IntelliSense - Visual Studio (Windows) | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022.

Waterfall methodology: Project management | Adobe Workfront (no date). Available at: https://business.adobe.com/blog/basics/waterfall (Accessed: April 21, 2023).

What is an API? (2022) Red Hat - We make open source technologies for the enterprise. Available at: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

What is an IDE? (2019) Red Hat - We make open source technologies for the enterprise. Available at: https://www.redhat.com/en/topics/middleware/what-is-ide (Accessed: April 20, 2023).

Engineer, D.A.C. and Dan AriasR&D Content EngineerHowdy! ? I do technology research at Auth0 with a focus on security and identity and develop apps to showcase the advantages

or pitfalls of such technology. I also contribute to the development of our SDKs (2019) How to hash passwords: One-way road to enhanced security, Auth0. Available at: https://auth0.com/blog/hashing-passwords-one-way-road-to-security/.

McGarr, O. (2009) “A review of podcasting in higher education: Its influence on the traditional lecture”, Australasian Journal of Educational Technology. Melbourne, Australia, 25(3). doi: 10.14742/ajet.1136.

Binder, M. (2020). There are now more than a million podcasts. [online] Mashable. Available at: https://mashable.com/article/apple-one-million-podcasts-milestone.

Perez, S. (2020) Spotify adds a built-in podcast playlist creation tool, 'your episodes', TechCrunch. Available at: https://techcrunch.com/2020/11/16/spotify-adds-a-built-in-podcast-playlist-creation-tool-your-episodes/ .

Sterne, J. (2006) ‘The mp3 as cultural artifact’, New Media & Society, 8(5), pp. 825–842. doi: 10.1177/1461444806067737.

Manuaba, I.B.P. & Rudiastini, E. 2018, "API REST Web service and backend system Of Lecturer’s Assessment Information System on Politeknik Negeri Bali", Journal of Physics: Conference Series, vol. 953, no. 1.

1.3 getting started - what is Git? (no date) Git. Available at: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F.

Engineer, D.A.C. and Dan AriasR&D Content EngineerHowdy! ? I do technology research at Auth0 with a focus on security and identity and develop apps to showcase the advantages or pitfalls of such technology. I also contribute to the development of our SDKs (2019) How to hash passwords: One-way road to enhanced security, Auth0. Available at: https://auth0.com/blog/hashing-passwords-one-way-road-to-security/.

Appendix A – Interesting but not vital material

iTunes search API tabel

<table>
<tr><th>Parameter Key</th><th>Description</th><th>Required</th><th>Values</th></tr>
<tr><td>term</td><td>The URL-encoded text string you want to search for. For example: jack+johnson.</td><td>Y</td><td>Any URL-encoded text string. Note: URL encoding replaces spaces with the plus (+) character and all characters except the following are encoded: letters, numbers, periods (.), dashes (-), underscores (_), and asterisks (*).</td></tr>
<tr><td>country</td><td>The two-letter country code for the store you want to search. The search uses the default store front for the specified country. For example: US. The default is US.</td><td>Y</td><td>See http://en.wikipedia.org/wiki/ ISO_3166-1_alpha-2 for a list of ISO Country Codes.</td></tr>
<tr><td>media</td><td>The media type you want to search for. For example: movie. The default is all.</td><td>N</td><td>movie, podcast, music, musicVideo, audiobook, shortFilm, tvShow, software, ebook, all</td></tr>
<tr><td>entity</td><td>The type of results you want returned, relative to the specified media type. For example: movieArtist for a movie media type search. The default is the track entity associated with the specified media type.</td><td>N</td><td>The following entities are available for each media type:
<table>
<tr><td>movie</td><td>movieArtist, movie</td></tr>
<tr><td>podcast</td><td>podcastAuthor, podcast</td></tr>
<tr><td>music</td><td>musicArtist, musicTrack, album, musicVideo, mix, song Please note that “musicTrack” can include both songs and music videos in the results</td></tr>
<tr><td>musicVideo</td><td>musicArtist, musicVideo</td></tr>
<tr><td>audiobook</td><td>audiobookAuthor, audiobook</td></tr>
<tr><td>shortFilm</td><td>shortFilmArtist, shortFilm</td></tr>
<tr><td>tvShow</td><td>tvEpisode, tvSeason</td></tr>
<tr><td>software</td><td>software, iPadSoftware, macSoftware</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td></td><td><table>
<tr><td>ebook</td><td>ebook</td></tr>
<tr><td>all</td><td>movie, album, allArtist, podcast, musicVideo, mix, audiobook, tvSeason, allTrack</td></tr>
</table></td></tr>
<tr><td>attribute</td><td>The attribute you want to search for in the stores, relative to the specified media type. For example, if you want to search for an artist by name specify entity=allArtist&attribute=allArtistTerm. In this example, if you search for term=maroon, iTunes returns “Maroon 5” in the search results, instead of all artists who have ever recorded a song with the word “maroon” in the title.

The default is all attributes associated with the specified media type.</td><td>N</td><td>The following attributes are available for each media type:<table>
<tr><td>movie</td><td>actorTerm, genreIndex, artistTerm, shortFilmTerm, producerTerm, ratingTerm, directorTerm, releaseYearTerm, featureFilmTerm, movieArtistTerm, movieTerm, ratingIndex, descriptionTerm</td></tr>
<tr><td>podcast</td><td>titleTerm, languageTerm, authorTerm, genreIndex, artistTerm, ratingIndex, keywordsTerm, descriptionTerm</td></tr>
<tr><td>music</td><td>mixTerm, genreIndex, artistTerm, composerTerm, albumTerm, ratingIndex, songTerm</td></tr>
<tr><td>musicVideo</td><td>genreIndex, artistTerm, albumTerm, ratingIndex, songTerm</td></tr>
<tr><td>audiobook</td><td>titleTerm, authorTerm, genreIndex, ratingIndex</td></tr>
<tr><td>shortFilm</td><td>genreIndex, artistTerm, shortFilmTerm, ratingIndex, descriptionTerm</td></tr>
<tr><td>software</td><td>softwareDeveloper</td></tr>
<tr><td>tvShow</td><td>genreIndex, tvEpisodeTerm,</td></tr>
</table></td></tr>
</table>

			showTerm, tvSeasonTerm, ratingIndex, descriptionTerm
			all: actorTerm, languageTerm, allArtistTerm, tvEpisodeTerm, shortFilmTerm, directorTerm, releaseYearTerm, titleTerm, featureFilmTerm, ratingIndex, keywordsTerm, descriptionTerm, authorTerm, genreIndex, mixTerm, allTrackTerm, artistTerm, composerTerm, tvSeasonTerm, producerTerm, ratingTerm, songTerm, movieArtistTerm, showTerm, movieTerm, albumTerm
callback	The name of the Javascript callback function you want to use when returning search results to your website. For example: wsSearchCB.	Y, for cross-site searches	wsSearchCB
limit	The number of search results you want the iTunes Store to return. For example: 25. The default is 50.	N	1 to 200
lang	The language, English or Japanese, you want to use when returning search results. Specify the language using the five-letter codename. For example: en_us. The default is en_us (English).	N	en_us, ja_jp
version	The search result key version you want to receive back from your search. The default is 2.	N	1, 2
explicit	A flag indicating whether or not you want to include explicit content in your search results.The default is Yes.	N	Yes, No