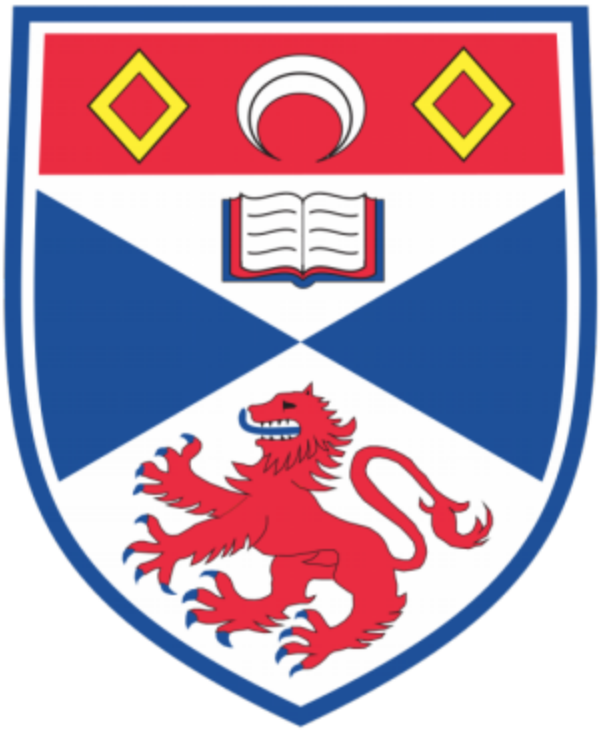


Librilisten Podcast Service

Cedonia Peterson
University of St Andrews
12 April 2021

Abstract

Librivox is a site that provides free, unlimited access to user-created audiobook versions of public domain books. While they provide several different listening options for enjoying their catalogue, each is just a variant on accessing the entire book at once. This can be a daunting and unwieldy prospect for many readers. The goal of Librilisten is to provide them with a more personalized experience by turning any Librivox book into a podcast that publishes new chapters on a schedule chosen specifically by the user to match their reading pace. Librilisten consists of a React.js web application, a Node.js API, and a MariaDB database.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 8,690 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

1 Introduction

Librivox.org [1] is an audiobook site that provides access to Public Domain recordings of books. While this is a great way for people to discover classic books, the sheer length of many books can be daunting for less confident readers. Furthermore, many of these older books were not originally intended to be read in a few days like our modern books. During the Victorian era, many novels - including all of Charles Dickens's works - were originally published as serials. Librivox's model of storing each chapter as a separate audio file provides a great opportunity to recreate that experience, but there is currently no functionality to set up a scheduled publication of chapters rather than receiving them all at one time.

This is where Librilisten comes in. It is an easy-to-use tool that lets users turn any Librivox audiobook into a podcast subscription that updates on a schedule chosen by the user. This allows users to start with just one chapter and receive more only when they need them so that they are never tempted to read ahead when they shouldn't or overwhelmed with the sight of so many unread chapters still to go. I also included a secret edit code, so that users can distribute their podcast to as many people as they choose while maintaining the sole power to edit it.

Beyond solving this initial problem Librilisten may have additional uses for different kinds of users. It provides a simple way for members of a book club to stay on a reading schedule together. It could also be used by English teachers to give all of their students on track with the reading.

Here is a list of the overarching aims/objectives of the project. They are roughly sorted by priority. Each of them is discussed in more detail in the rest of the report.

- Writing and hosting a web application that provided a smooth user experience and interfaced appropriately with the API
- Writing and hosting an API with meaningful endpoints which allowed for full functionality while minimizing risk to the security of the project.
- Designing a database to hold the podcast data in the most accessible, efficient way possible
- Parsing Librivox RSS files and transforming them into the files required for Librilisten (i.e. chopping off excess chapters and adding chapter publication dates)
- Hosting Librilisten RSS files in a way that made it simple for users to subscribe using the podcast app of their choice
- Automatically performing a daily update on all Librilisten RSS files

2 Context Survey

2.1 Librivox

The Librivox service has catalogued over 15,000 public domain books by turning them into public domain audiobooks. They provide users with a wide variety of options to enjoy their books, including downloading the files in a zip and following an RSS link to subscribe on the podcast app of their choice. For example, Figure 1 shows the listening options for Standish's *Freshman Against Freshman:*

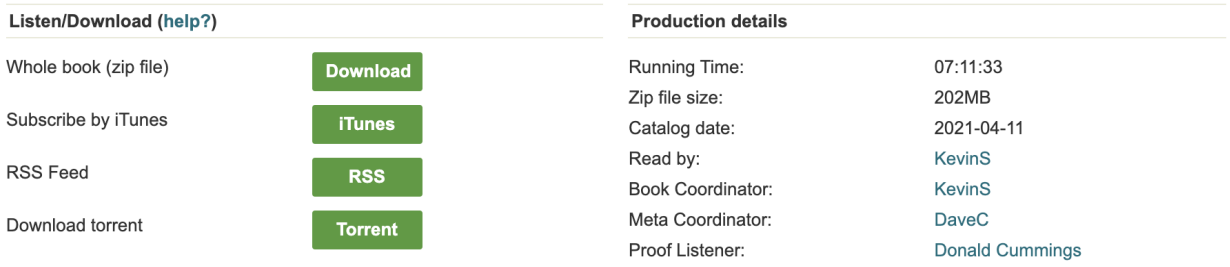


Figure 1: listening options for Freshman Against Freshman [1]

When I attempted to use the "iTunes" button, on several different works, it never worked. The RSS Feed is the option that is most useful for my project, and most similar to it. It is the link to an RSS file with every single chapter of the book as an episode.

The difference between Librilisten and this Librivox RSS file is that Librilisten will provide the user with an RSS file with just the first chapter-episode. On a schedule of the user's choosing, new chapter-episodes will be published.

2.2 Tools and Technologies

The Librivox API was crucial for the development of Librilisten. I used it to retrieve the original Librivox RSS files, as well as details such as the author and title of the book, so I could parse them and generate the service.

As discussed above, this entire project hinges on the manipulation of RSS files. These are the files that podcast services parse to serve the episodes of podcasts to their users. They are xml files, which include metadata about the podcast as a whole (name, description, content warnings, etc.) and includes a separate object to describe each episode, including the link to its actual audio file as well as its publication date and title, etc. As an example, here is the start of the Librivox RSS file for *The Public Orations of Demosthenes*:

```
<?xml version="1.0" ?><rss xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"
xmlns:media="http://search.yahoo.com/mrss/"
xmlns:creativeCommons="http://backend.userland.com/creativeCommonsRssModule" xmlns:atom="http://www.w3.org/2005/Atom"
version="2.0">
<channel>
  <title><![CDATA[Public Orations of Demosthenes, The by Demosthenes (384 BCE - 322 BCE)]]></title>
  <link><![CDATA[https://librivox.org/the-public-orations-of-demosthenes/]]></link>
  <atom:link rel="self" href="https://librivox.org/rss/15773" />
  <description><![CDATA[This book, originally published in two volumes, collects the most important complete public
orations by Demosthenes, arguably the most famous Athenian orator, that we still have access to. Demosthenes was a
Greek statesman and orator of ancient Athens. His orations are nowadays important sources of contemporary Athenian
oratorical expression and provide an insight into the politics and legal affairs of Athens during the 4th century BC.
- Summary by Leni]]></description>
  <!--<genre>project element=Genre</genre>-->
  <!--<language>project element=lang.code</language>-->
  <itunes:type>serial</itunes:type>
  <itunes:author>LibriVox</itunes:author>
  <itunes:summary><![CDATA[This book, originally published in two volumes, collects the most important complete public
orations by Demosthenes, arguably the most famous Athenian orator, that we still have access to. Demosthenes was a
Greek statesman and orator of ancient Athens. His orations are nowadays important sources of contemporary Athenian
oratorical expression and provide an insight into the politics and legal affairs of Athens during the 4th century BC.
- Summary by Leni]]></itunes:summary>
  <itunes:owner>
    <itunes:name>LibriVox</itunes:name>
    <itunes:email>info@librivox.org</itunes:email>
  </itunes:owner>
  <itunes:category text="Arts">
    <itunes:category text="Literature" />
  </itunes:category>
  <!-- file loop -->
    <item>
    <title><![CDATA[Introduction]]></title>
    <itunes:episode><![CDATA[0]]></itunes:episode>
    <!--<reader>file element=reader</reader> -->
    <link><![CDATA[https://librivox.org/the-public-orations-of-demosthenes/]]></link>
          <enclosure
url="http://www.archive.org/download/the_public_orations_of_demosthenes_2104_librivox/publicorations_00_demosthenes_64
kb.mp3" length="0" type="audio/mpeg" />
  <itunes:explicit>No</itunes:explicit>
  <itunes:block>No</itunes:block>
  <itunes:duration><![CDATA[]]></itunes:duration>
  <!--<pubDate>file element=rss.pubDate</pubDate>-->
  <media:content
url="http://www.archive.org/download/the_public_orations_of_demosthenes_2104_librivox/publicorations_00_demosthenes_64
kb.mp3" type="audio/mpeg" />
 </item>
    <item>
    <title><![CDATA[On the Naval Boards]]></title>
    <itunes:episode><![CDATA[1]]></itunes:episode>
    <!--<reader>file element=reader</reader> -->
```

Figure 2: the Librivox RSS file for The Public Orations of Demosthenes *[1]*

2.3 Similar Projects

The idea of turning a book into a podcast has not been explored very much. The results when I Google for anything related to "turning a book into a podcast" are targeted at modern creators. There are several major categories of results.

First, there are many sites like the NY Book Editors article for authors interested in having their books recorded and turned into podcasts (see Figure 3).

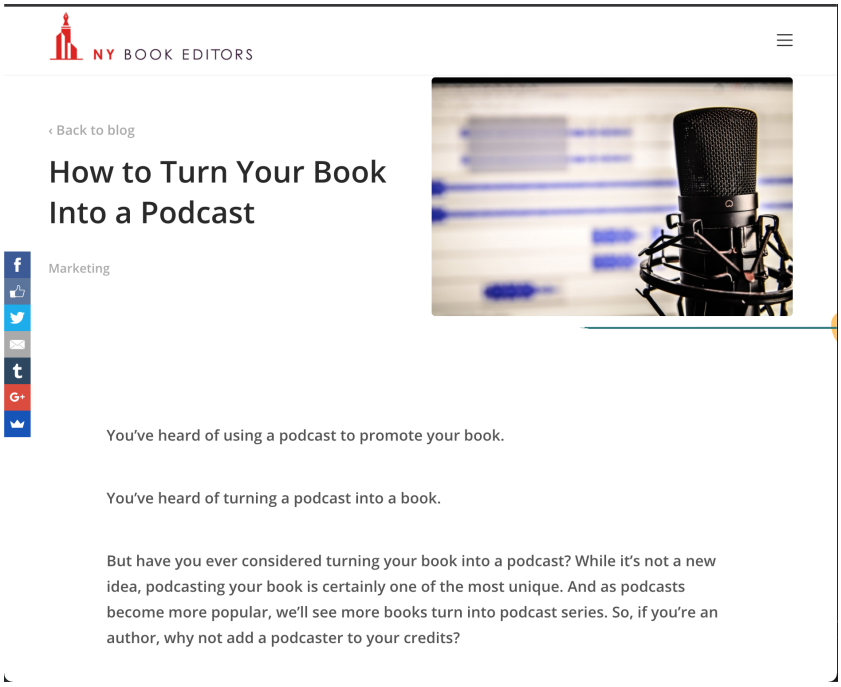


Figure 3: "How to Turn Your Book Into a Podcast" [2]

Second, there are articles about turning a podcast into a book, such as the following article by Publishing Parrot (Figure 4).

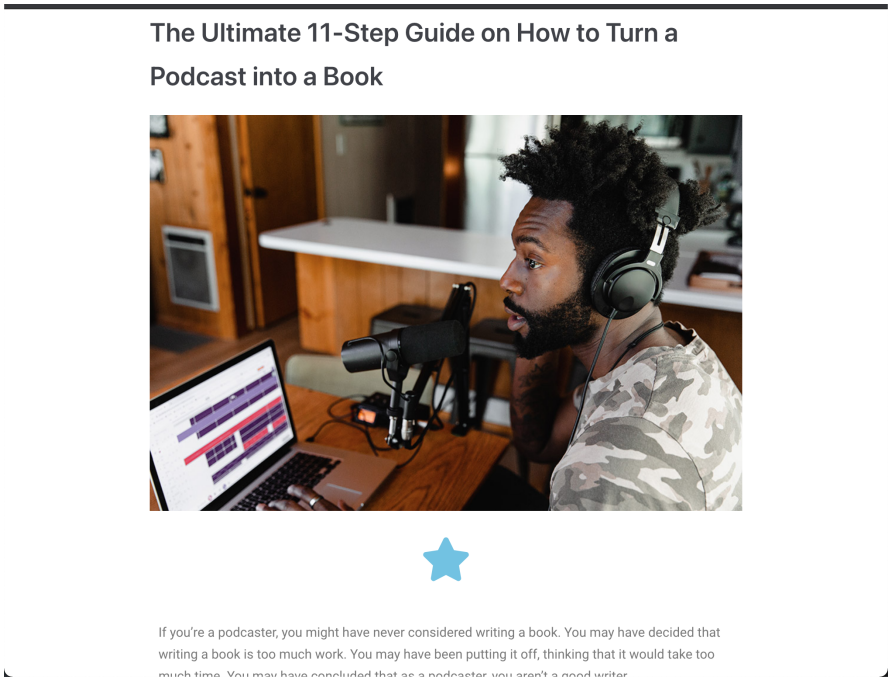


Figure 4: "The Ultimate 11-Step Guide on How to Turn a Podcast Into a Book" [3]

Finally, several services turn public domain books into podcasts. The lists of public domain audiobook podcasts mostly consist of YouTube and Librivox links [4], [5]. An example of

the podcasts on these lists is the Spotify "Audiobooks Daily" podcast, which publishes new chapters of public domain podcasts regularly (Figure 5).

Figure 5: Audiobooks Daily on Spotify [6]

While this approach of publishing chapters periodically for listeners to enjoy as they go is similar to Librilisten's functionality, users get no choice in what title is published or how often its chapters are released. It is a much less personal experience.

Users can also get audiobooks through their libraries, either from checking out physical CDs or by entering their library card information into apps like Libby or Overdrive if their library participates in those programs. This is a great way to get access to audiobooks beyond just public domain titles, but it still doesn't have that personalized feel of a customized podcast. It also raises the barrier to entry for these books, since users must have a library account, download the app, and not be put off by the long list of chapters.

3 Requirement Specification

The overarching goal of this project was to have a fully functioning hosted service that would take a user-input Librivox book, and a chapter publication schedule, and would generate a link to a (periodically updating, as specified by the inputs) RSS file which the user could use with any podcast app of their choice.

This goal was broken into smaller requirements which were summarized in the Description, Objectives, Ethics and Resources document submitted at the beginning of the

school year. Each requirement is revisited and discussed in the Evaluation section to identify whether or not it was successfully completed.

3.1 Primary

- An attractive, minimalist, user-friendly web application that allows users to generate a podcast feed by indicating a Librivox book and how often they wish to receive chapters (may include an About page if usage instructions are necessary)
- Users can easily find their customized podcast in the podcast app of their choice; multiple users (e.g. a book group) can all access it from different apps
- An API server with an endpoint to generate a podcast with a specified Librivox book and publication frequency
- A database that stores whatever data is necessary for maintaining the podcast publication schedule
- A cron job, or something similar, which runs periodically to launch whatever activities are required to continue publishing podcast chapters

3.2 Secondary

- Users may generate a podcast that begins on any chapter they choose (all previous chapters will be available on the RSS feed)
- After a podcast has been created, its creator may edit the frequency its chapters are published (this requires both modification of the UI and an additional API endpoint)
- Gather user feedback on the product via a survey and improve its usability according to the survey results

3.3 Tertiary

Throughout the development process, I came up with more feature ideas that could be considered tertiary requirements. These were not included in the original DOER.

- An /updateRightNow endpoint which accepts a podcast to receive a new chapter right away (useful for demoing the project as well as for letting users request a one-off chapter if they're caught up)
- Allow users to delete their podcasts
- Allow users to edit publication frequency
- Allow users to skip a certain number of chapter publications (ex: when they go on vacation)
- Give each podcast a secret edit code so that only the creator of the podcast can edit it

4 Software Engineering Process

4.1 Timeline

Since this is a minor project, I was provided with the option to do this project in either one semester or over the entire school year. I elected to use both semesters to work on this project, for two main reasons.

First, I wanted to give myself the maximum amount of time to develop this project.

Second, by stretching the work out over two semesters, it was less stressful in any given week and I was able to enjoy the development process rather than getting frazzled by impending deadlines.

4.2 Agile Development plan

Because this was a year-long project, it was crucial to find a development process that was sustainable and productive for the long term. My supervisor Michael helped a lot with this, and our weekly emails functioned as my scrum meetings. I started each meeting by discussing what I'd accomplished in the previous week, and what issues I had encountered and whether they were successfully dealt with. After he provided insight into any ongoing issues, we would then discuss my plans for the next week's sprint.

My development schedule was never as fine-tuned as it should have been. I tried setting scheduled goals a couple of times (finishing a certain endpoint by a specific week, etc.), but in reality, I was pretty terrible at predicting what would be very easy and what would hold me up for weeks. In particular, I lost a couple of weeks in the second semester due to some complex issues with hosting the UI on the school servers.

Overall, because I had so much time to do this minor project and such unpredictable progress, my casual scrum process proved to work well. My meetings with Michael helped me stay focused on producing tangible deliverables as often as possible, and my rough schedule kept me moving toward the final deadline.

As far as development planning went, my greatest focus was on keeping track of all the goals I wanted to accomplish. In the first semester, I did this by opening issues for every bug, feature, cool idea, etc. that I didn't want to forget about. In the second semester, I made a kanban board where I organised all of the remaining open issues into the following categories: Major Features, Small Features, Cleanup [essentially small fixes/debugs], Testing, and Done. I kept this board up to date throughout the semester and added new cards to it as necessary (for my purposes, I didn't see the point of going through the extra steps of opening issues). Figure 6 shows what it looked like in early April.

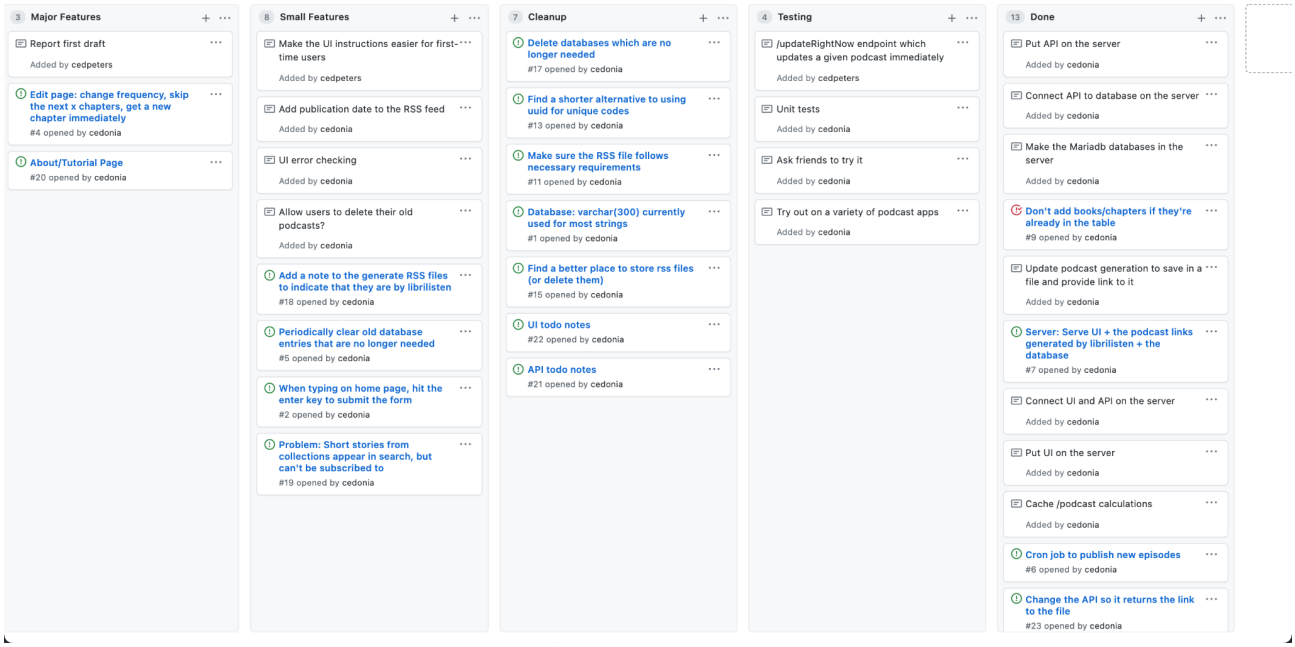


Figure 6: My Kanban board from Github

I made my repository public. The kanban board is available at https://github.com/cedonia/cs4098/projects/2. The latest version of the code is on the *second-sem* branch, and the master branch holds the state of the code at the end of the first semester.

4.3 Development Tools

I created a private Github repository for this project which I used both semesters. I committed regularly every time I worked on the code, and the version control features of git proved extremely useful. There were several times over the year when my code stopped working, especially when I tried to run it on the server, and reverting to an earlier version helped me figure out what change had broken everything.

I developed my code using Sublime Text on my Macbook Pro 2015. I ran everything on my host server by ssh-ing into it and using git to pull the latest version of my code. I used tmux to keep node sessions running even while I was disconnected from the server, and to open multiple terminal screens side by side without having to ssh multiple times. In the early planning days, I also used wireframe.cc to plan out what my UI would look like. See the *planningFiles* directory for all of the planning files I made. *uiGoals* and *wireframes* were both made in the first semester, during my initial planning period, and *DatabasePlanning* and *Pages* were made in the second semester.

5 Ethics

As far as the content of the project goes, there were no ethical concerns. The process of turning public domain audio recordings into scheduled podcasts is not stressful for users and not ethically dubious in any way. The recordings are freely available for use, and Librivox's permission was not even needed for this project.

The more complex issue was managing user data on the project, in a way that would allow podcasts to be accessed and edited by their creators without leaving the service open to attack or storing any personally identifiable information (PII). This was accomplished by generating two codes for each podcast: a podcast id that can be freely distributed to anyone the user is reading the book with and a secret edit code which they are asked not to share with anyone. They can use the edit code to make changes to the podcast. Thus no cookies are stored and no other form of authentication (such as logging in via another service) is ever necessary.

6 Design

This project required the development of several different component pieces, each of which has its own design decisions. Each will be discussed in its own section.

6.1 User Interface

I decided to write a very simple, clean UI for the project to make it as easy and straightforward to use as possible. The entire application consists of just a handful of pages, which the user visits by clicking buttons as they walk through the process of making a podcast.

The clean aesthetic of the UI follows minimalistic design principles. I chose to pursue a minimalist design because it would allow me to build a site that works on screens of all sizes, and on mobile devices, as well as keeping loading times quick [7]. I tried out the service on my iPhone, on Chrome, and it did indeed look and work right (Figure 7).

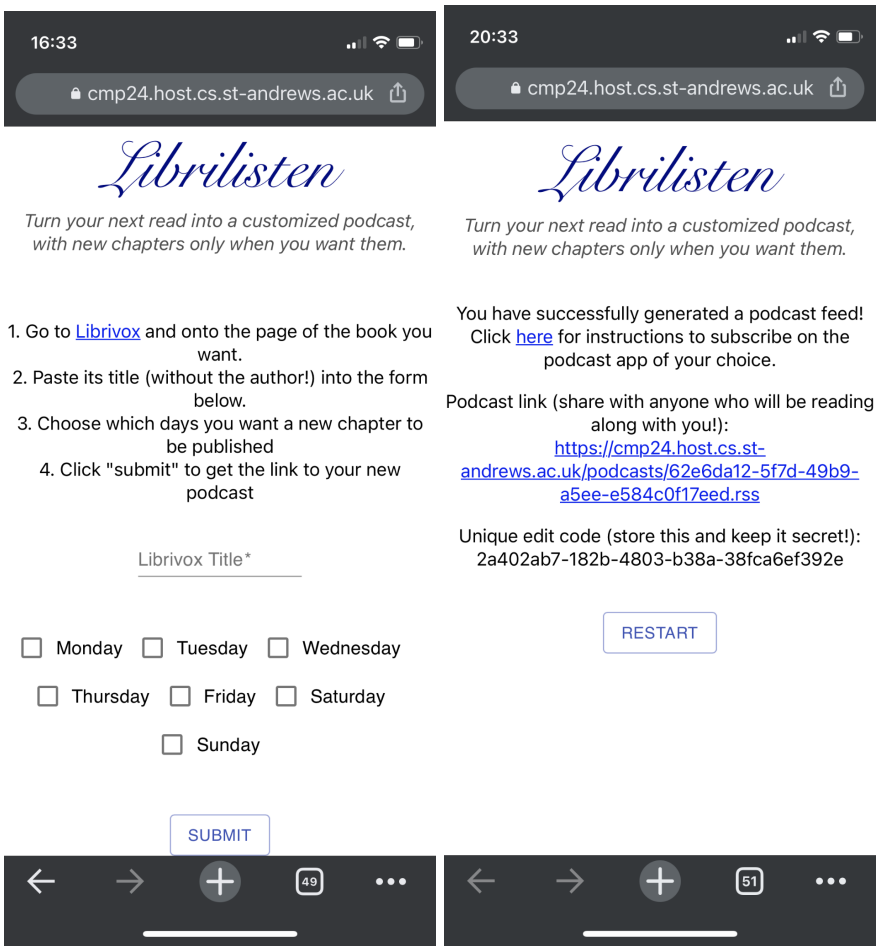


Figure 7: Screenshots of Librilisten on the ios Chrome app

In my initial design (see *planningFiles/uiGoals.txt*) I planned to include some pages which did not make the final design. These include:

- On the home page:
    - Users can choose what time their chapters get published
    - Users can request that the first x chapters already be published
    - Display how long a podcast will take to finish if chapters are published at the rate currently input
- Edit page
- Include a "Use guide" page that explains how to use everything
- Include a "Why" page to explain the motivation for the project and ideas for how to use

All of these were removed due to a combination of infeasibility and limited usefulness. I build a few edit endpoints (publishing a new chapter on command, and pausing for a certain number of days) which I didn't have time to make UI pages for. The "Use guide" page, in particular, seemed superfluous since there are useful instructions on the home and submission pages, plus a short instructional page with information about subscribing to the podcasts, so there didn't seem to be much need for instructions. The "Why" page would have been marginally interesting, but probably not useful for users. When I publish this site after graduation, I will probably add a footer with a one-line description of the project and a link to email me with questions.

6.1.1 Home Page

The following is the home page of the service (Figure 8):

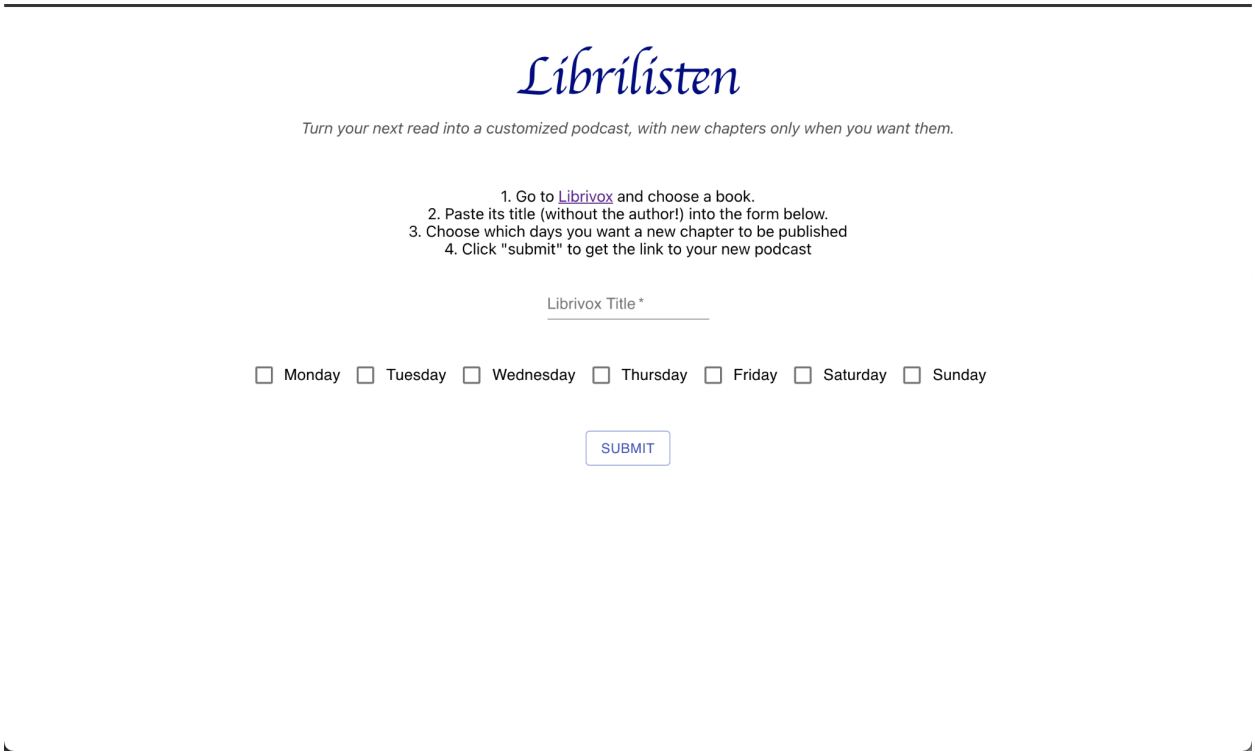


Figure 8: Home page

Note that the instructions for use are at the top of the page, so users don't need to navigate to a help page.

The biggest design decision was how to let users choose the frequency of the publication of new chapters. I initially planned to have a dropdown, from which they could choose options such as "daily", "biweekly", and "weekly". I decided that this was not detailed enough, since different people will want new reading material on different dates, so changed it to allow them to choose from any combination of days of the week.

6.1.2 Confirmation Page

After the user clicks "Submit", this is the Confirmation page (Figure 9):

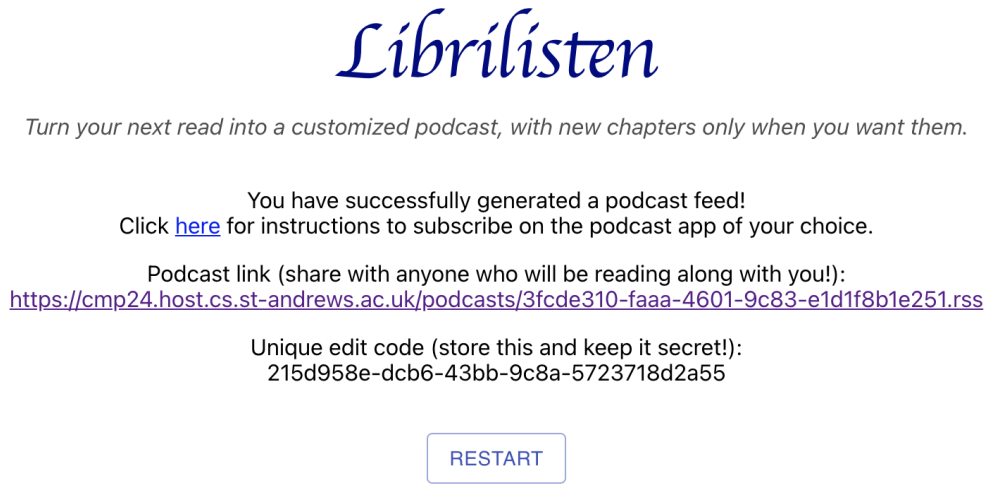


Figure 9: Confirmation Page

Note that the only way to navigate to a different page is to click "Restart" which will take the user back to the Home page.

6.1.3 Subscription Instructions

I initially wrote out a small tutorial for subscribing to an RSS link, which I put on the server at the following link: https://cmp24.host.cs.st-andrews.ac.uk/podcast-instructions.txt.

After receiving criticism of its brevity and ugliness, I made a Google Doc with more detailed instructions and put a link to it on the Confirmation page of the UI instead.

6.2 API

I decided to host the API on my host server as well and to make it public in case future developers might be interested in using it for other projects. It is hosted on the school server just like the UI, and all the API endpoints are reached at https://cmp24.host.cs.st-andrews.ac.uk/api/.

I wound up writing the following endpoints for the API:

- GET /genPodcast/title/:title
    - Accepts the title of a novel as a parameter, and the booleans of publication for each day of the week as queries.
    - Computes the new podcast by storing relevant data in the database tables and generating the initial RSS file.
    - Returns the librilisten id of the generated podcast and the secret edit code for the podcast
- GET /api/update
    - Does not require any specific data to be sent with the endpoint call
    - Loops through all podcasts eligible to be updated that day and updates them
    - Returns a status 200.
- GET /api/updateRightNow/:secret-edit-code
    - Accepts the secret edit code of a podcast as a parameter
    - Adds one chapter to the podcast associated with that secret code (even if it's already been updated today)
    - Returns a status 200
- GET /api/skipPubDays/:secret_edit_code
    - Accepts the secret edit code as a param, and the number of days to skip as a query
    - Stores the number of days to skip in the appropriate podcast
    - Returns a status 200

These three endpoints do everything required for the working of Librilisten. Note that all three of them are reliant on the Librivox API, and will not perform their respective file generation operations when that API is down. During the development of Librilisten, the Librivox API went down for several days. There is no way to divorce Librilisten from its reliance on Librivox's API, however, so this is simply a dependency to be aware of.

6.3 Database

My database design shifted halfway through the project. For a while, I was considering removing librivox_books and librivox_chapters, because I planned to store the Librivox RSS files in a way that would make it unnecessary to use the Librivox API when updating podcasts. However, this proved impractical. I wound up redeveloping a database design, incorporating appropriate constraints.

As seen in Table 1 below, all three tables received changes in the redesign.

	Original Design	Redesign
librilisten_podcasts	• Librilisten_podcast_id: varchar(1000) not null • Librivox_book_id: int not null • secret_edit_code: varchar(1000) • mon: bool • tues: bool • wed: bool • thurs: bool • fri: bool • sat: bool • sun: bool • is_done: bool • next_chapter: int not null • skip_next: int not null	• Librilisten_podcast_id: varchar(1000) not null [primary key] • Librivox_RSS_url varchar(1000) not null [foreign key to librivox_books] • secret_edit_code: varchar(1000) not null [unique] • mon: bool • tues: bool • wed: bool • thurs: bool • fri: bool • sat: bool • sun: bool • is_done: bool • skip_next: int not null
librivox_books	• Librivox_book_id: int not null • book_title: varchar(300) not null • book_author: varchar(300) not null • url_RSS: varchar(300) not null • quantity_chapters: int not null	• Librivox_RSS_url: varchar(1000) [primary key] • title: varchar(1000) • author: varchar(1000)
librivox_chapters/librilisten_chapters	• Librivox_book_id: int not null • chapter: int not null • url_recording: varchar(200) not null	• Librilisten_podcast_id: varchar(1000) not null [foreign key to librilisten_podcasts] • chapter_num: int not null • pub_date: varchar(1000) Primary key = Librilisten_podcast_id + chapter_num

Table 1: Original and final database designs

Figure 10 is an E-R diagram of the final database design.

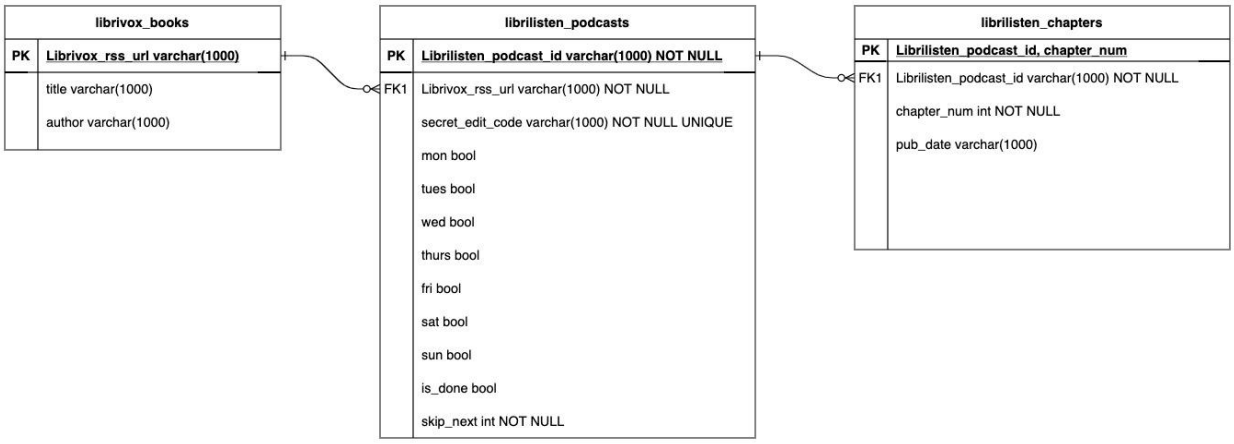


Figure 10: E-R Diagram of the final database design

See the *planningFiles/DatabasePlanning.txt* file to see the full plan for my final database design as well as the queries used to make the tables. Before I implemented the new design I wrote out pseudo-code for all of the scenarios I could think of involving the database, to make sure it would work well for the entire project.

The basic idea of my database design is simple:

1. Use *librilisten_podcasts* to keep track of all necessary information about each podcast including the link to the original Librivox RSS url, the secret edit code, and details for the publication schedule.
2. Use *librilisten_chapters* to keep track of all the chapters associated with a podcast, including both published and unpublished chapters. If *pub_date* is null, that chapter is unpublished. The only information stored here is the chapter number and the pub date, everything else will be extracted from the Librivox RSS files when required.
3. Use *librivox_books* to store pertinent information about books that can be easily referenced later. This was never used in the current design but provides functionality to easily display the title and author of a book in the UI, for example on the edit page or in the response sent by the */skipPubDays* endpoint. Note that no matter how many podcasts use the same book, there will only be one entry in *librivox_books* for that book.

6.4 RSS Files

6.4.1 Generating

My plan for generating the RSS files changed partway through the project. I initially intended to tweak my initial database setup to add a table where I would store every single detail from the original Librivox RSS file, allowing me to generate each subsequent Librilisten file from scratch using that data instead of querying Librivox for their file each time.

When I realised just how complex the Librivox RSS files are, and how difficult it would be to store every detail from their files in my database, I then considered storing the Librivox RSS files in a directory on the server, where I could refer to it each time I generated a new Librilisten variant of it. This would not only increase the speed of my file generation (since I would no longer need to query either Librivox *or* the database to retrieve the original file) but would also decrease my reliance on Librivox's API being up at all times. I also knew I could optimize the number of Librivox files stored in my directory - and the number of Librivox API calls - by implementing a cache system. I was still concerned, however, that I would have way too many Librivox files stored on my server if users requested too many different books. Plus, if I'd done this, I also would have needed to add some sort of periodic verification that my cached files were still the same as the Librivox files. I didn't want to deal with out-of-date files, especially if the audio file links could become incorrect and break the entire project.

Another variant of this idea was to simply save the full Librivox RSS file under the Librilisten id of each new podcast each time I generated a new podcast and to comment out all but the published chapters. To add a new chapter, I would uncomment the next chapter and edit the file to give it a publication date. This would have not only decreased the number of Librivox API calls (as the publication of new chapters would not require a new call) but would have also avoided the issue of storing huge amounts of Librivox RSS files as well as a file for each Librilisten podcast. It also would have avoided the extra time required to regenerate the Librilisten file each time, which includes a database query to retrieve chapter publication dates and a loop to assign each chapter its publication date again. However, by avoiding checking in with Librivox each time a new chapter is added, this still presents the issue of potentially becoming out of date with Librivox. It also seems like a pretty hacky solution to just comment out bits of a file that are not currently in use.

Due to these considerations, I eventually decided to go with the implementation described in the database design section.

6.4.2 Serving

The other major decision I made with the RSS files was how to serve them. I initially wrote an API endpoint called something like */api/getPodcast/:librilisten_id* which would calculate the required RSS file and return it in its response.

After some discussion with Michael, who pointed out just how time inefficient it would be to hit a GET endpoint every single time the RSS file was required and that there were other options due to using nginx, I decided to change this implementation. Instead, I made a folder inside nginx_default, called *podcasts*, which contains all of the RSS files which are each stored as *[librilisten_id].RSS*. The files can then be accessed by users, and podcast aggregators, at https://cmp24.host.cs.st-andrews.ac.uk/podcasts/[librilisten_id].RSS.

7 Implementation

7.1 User Interface

I chose to build the UI in React.js for a few reasons. First, I was already familiar (if a bit rusty) with it so the learning curve to start development would be pretty easy. Second, I wanted a very simple, straightforward framework which would do most of the heavy lifting for me so that I could write and host my web app with limited overhead.

I used the create-react-app default configuration to create the initial React app, and then edited the files to add the components which I needed. By the end of the project, I had just two components for the entire UI: Home.js and Confirmation.js. Both are pretty straightforward since the UI is so simple.

I chose to pass the librilisten id and the secret edit code around as simple uuid values, in both the API and the Home UI component, and to turn them into URLs only when I was sending them to the confirmation page. This makes them as short, simple, and easily manipulable as possible. It also means that there is only one place where the URLs are generated, which makes the code easy to edit if those URL formats need to change later.

In Home.js, I chose to treat each of the daily checkboxes like a separate entity. There are seven *handle[Day]Change* methods to change the checked status of each of the seven days. Similarly, each day gets its own boolean value in the state object to keep track of whether it's checked. I could have attempted a more complex approach, by storing a single array of seven booleans in the state and having a single *handleDayChange* method that accepted an index value of which day it should change, but that would have obfuscated the code by forcing the programmer to keep track of which index went with which day. Since the code specific to each day is so simple, and it's highly unlikely that it (or the number of days in the week) will change, I decided it was easiest to give each day of the week its own treatment.

I hosted the UI on my host server by running "npm run build" then copying the resulting build file into my nginx_default/ folder on my host. It was then automatically served by nginx whenever users visited https://cmp24.host.cs.st-andrews.ac.uk/build/index.html.

I initially planned to host the UI at the root of my server site, https://cmp24.host.cs.st-andrews.ac.uk/, and for each new page to have its own url (/home, /confirmation, etc.). I planned to have the home page reroute the user to /confirmation?librilisten_id=[id]&secret_edit_code=[code]. I intended to make the page to edit a podcast work similarly. The confirmation page would provide a link, in the format /edit?edit_code=[code], which the user would save and then visit whenever they wanted to edit their podcast.

However, I ran into severe difficulties with hosting my React web app on nginx in a way that allowed me to do all of that routing. Instead, the home page is now at https://cmp24.host.cs.st-andrews.ac.uk/build/index.html (which tells nginx to look in nginx_default/build/index.html to find the files to run the UI). To simplify the entire process, it's now a one-page app: when the user clicks "submit" the page renders the Confirmation component, but it never actually sends the user to a new page. The url does not change. Now the home page sends the relevant data (librilisten id and secret edit code) to the confirmation page by simply passing those details in as parameters when it renders the Confirmation component.

Note that if the Home page receives a response from the API which is not a 200 status, it will print an error to the console and not advance to the confirmation page. It does not, however, print anything to the screen. This would be an additional feature to implement before the publication of the project.

Similarly, if the user has not selected any days of the week, the UI simply won't move forward. Again, the addition of error messages was a feature that was not finished in time for submission but would be added.

7.2 API

I chose to write the API in Node.js, using the Express.js framework to build HTML endpoints. Since Node.js is single-threaded, it was perfect for writing the non-blocking event-driven endpoints required for this project. Express is a useful framework that greatly simplifies the development process of Node APIs by automatically parsing payloads, storing session cookies and more.

As stated in the Design section, there are only a few endpoints in Librilisten. Each of these endpoints is very powerful, however. I will discuss each of these in turn. First I will also discuss the *FileGenerator* class, which is used by both endpoints to generate the new/updated RSS files.

7.2.1 FileGenerator.js

The *genUpdatedFile* method is the only publicly exported method of this class, and since it performs the actual file generation for Librilisten it seems relevant to discuss it in more detail.

The three arguments accepted by the file generator are *dateTime* (the current date and time in a String format accepted by Apple Podcasts), *url_RSS* (the url of the original Librivox RSS file), and *librilisten_id* (the uuid id of the podcast which is currently being updated).

The file generation process is completed in two steps. First, the *librilisten_chapters* table is queried to retrieve the publication dates of all published chapters with the given Librilisten id.

These chapters are stored in an array, and then the current *dateTime* is added as the latest publication date to that array. A second database query is run, updating the *librilisten_chapters* table to add the latest publication date there as well.

The second step in the file generation process is to take the array of published chapters and use it to generate a new RSS file. This is done by retrieving the original RSS file from Librivox, parsing it, and splicing off all of the chapters which don't have a publication date. I then loop through the remaining chapters, adding a publication date to each one, then convert the entire parsed RSS file back into xml and write it to a file with the name *[Librilisten_id].RSS* which is stored in the nginx_default/podcasts folder on the server so that users can access it at https://cmp24.host.cs.st-andrews.ac.uk/podcasts/[Librilisten_id].RSS.

Note that it is in this step where I check to see if the newest chapter is the last one in the book, and if it is I update the entry for this podcast in *librilisten_podcasts* to set *is_done=true*.

One important decision during this process was whether to allow unlimited updates in a single day (so that several chapters could be added to a podcast on the same day if the */update* endpoint got called multiple times). I added in a boolean, *ignorePubDayDup*, which is set to false by the */update* endpoint. This prevents the publicly exposed endpoint to be exploited by bad actors (or misused by confused users) to publish all the chapters of many podcasts in a single day, removing the entire point of using Librilisten. Doing this has the beneficial side-effect of avoiding publication duplication on the day the podcast is generated: if, for example, a user generates a podcast on Tuesday morning before the updates of the day are performed, and they request that new chapters be added every Tuesday, then even when the *genUpdatedFile* method is called on their podcast a second chapter will *not* be added to their feed that day.

However, the /*updateRightNow* endpoint sets *ignorePubDayDup* to true. Because this endpoint only edits one podcast, based on that podcast's secret edit code, I assume that the person calling this endpoint really is the user of the podcast who is requesting more chapters because they're caught up on the book and eager for more. This endpoint will work an unlimited number of times in the same day.

7.2.2 /api/genPodcast

This endpoint requires one parameter, *title*, and seven queries: *mon, tues, wed, thurs, fri, sat,* and *sun* (each of which is a boolean). It returns a JSON with two items in it: the secret edit code, and the librilisten id of the newly generated podcast. These are necessary for referencing this podcast again in the future.

7.2.2.1 Using uuid

The first step in generating a new podcast is to generate the Librilisten id of the podcast and the secret edit code. I chose to make both of these uuid because that was a simple way to make sure that both these values would be unique strings of a fairly standardised length. I initially considered making the id a more meaningful string, such as a unique perversion of the

book title (ex: a podcast for *Autumn* by John Clare could be "autumn123"). This would have been more memorable for users, but since they really just need to store the RSS link in a podcast app and never use it again that didn't seem very important. The added complexity of generating these unique strings based on the titles did not seem worth the small benefit.

Another option for the id would have been to give each new podcast an id incremented from the previous one. This could have been strictly incremental ("1", "2", "3", etc.) or more complex, but any variation on this idea introduced two main weaknesses: first, it would allow any user to find and subscribe to any podcast every generated by Librilisten. This isn't a huge problem, since they wouldn't be able to edit it without the edit code, but it's still unwanted behaviour. The much bigger concern is that over time it would become increasingly difficult to generate new, unused variants on previous ids. The ids need to stay short enough to be included in the RSS urls. The use of uuid sidesteps these issues by simply generating - as uuid stands for - a "universally unique identifier".

As for the secret edit code, it is even more crucial that no one can guess it. While it's unfortunate that it's not easily memorable for users - since this is the code that they need to keep track of - any attempt to make it more memorable would unfortunately also make it more guessable for bad actors.

7.2.2.2 Querying the Librivox API

The Librilisten API uses the Librivox API endpoint */api/feed/audiobooks* to retrieve the necessary information about the book being requested by the user. The API identifies books by either their Librivox id or by their title. I wanted to query it using book id since there's too much risk of multiple books having the same name. Unfortunately, the Librivox book detail page does not mention the Librivox id assigned to it anywhere on the page. I then wanted to have users paste in the Librivox link to their preferred book, hoping I could use it to get the unique identifier of a book to then use with the API. This did not work either, because the urls of the books on the Librivox site do not include their id. For example, the full url for the Librivox page for Herman Melville's *Moby-Dick* is simply https://librivox.org/moby-dick-by-herman-melville.

Because of this, I gave up on the Librivox book ids. Instead, I have the user directly paste in the title of the novel and I use that as the unique identifier when contacting Librivox. It looks like Librivox has already put some effort into making titles on their site unique. For example, there are multiple recordings of *Pride & Prejudice* which are given the unique titles of 'Pride and Prejudice", "Pride and Prejudice (version 2)", etc., up to "Pride and Prejudice (version 6, dramatic reading)" [1].

As an example, a Librivox API call made by Librilisten for *Autumn* by John Clare would be https://librivox.org/api/feed/audiobooks?title=autumn&&fields={url_RSS,num_sections}. The result is Figure 11:

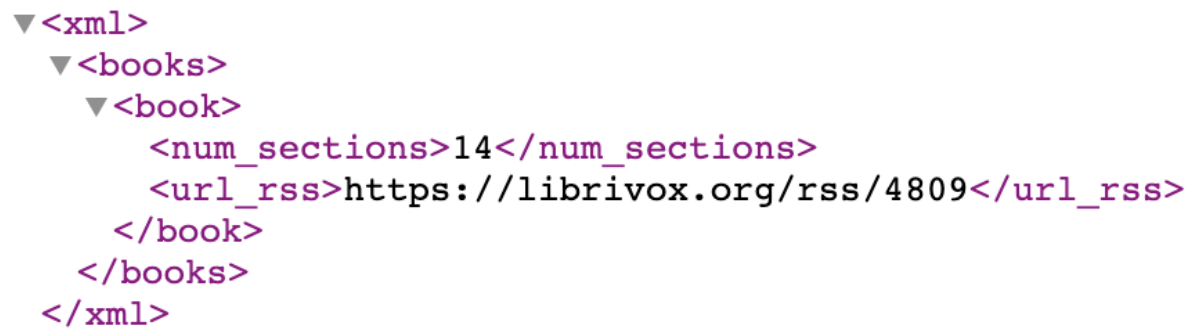


Figure 11: Autumn *Librivox API result*

7.2.3 /api/update

7.2.3.1 Design

The */update* endpoint requires no parameters or queries, and it returns simply a 200 status to indicate when it has finished. It is fairly simple in comparison to the */genPodcast* endpoint. Everything it does can be summarized in two steps.

First, I query the database to retrieve a list of the Librilisten podcasts which are eligible to be updated on the current day. These are the podcasts that are not done (meaning they still have unpublished chapters), where the number of days to skip is 0 (they're not on pause), and where the current day is true (the user has asked to receive new chapters on this day). I take the results of this query and loop through each of these podcasts, calling *FileGenerator.genUpdatedFile* on each one. As discussed in that class's section, this will generate a new RSS file for a podcast with one additional chapter, as long as it hasn't already been updated today.

Second, I decrement the number of days to skip for all podcasts which are currently on pause and which would normally receive a new chapter today.

7.2.3.2 Automated Daily Call

Since this endpoint needs to be called daily, I needed to find a way to automate an endpoint call every 24 hours.

I initially planned to set up a cron job that would make the daily endpoint call. Unfortunately, as noted on the Systems Wiki, crontabs are disabled on the host servers. I considered using the *at* command as they recommended, but decided ultimately to use an Azure Logic App [8] to do it instead. I was able to set up a free Azure account with my university account, and I appreciate the user-friendly UI to set up a task and monitor its history. For

example, here is what its reporting page looked like after about a week of being used (Figure 12):

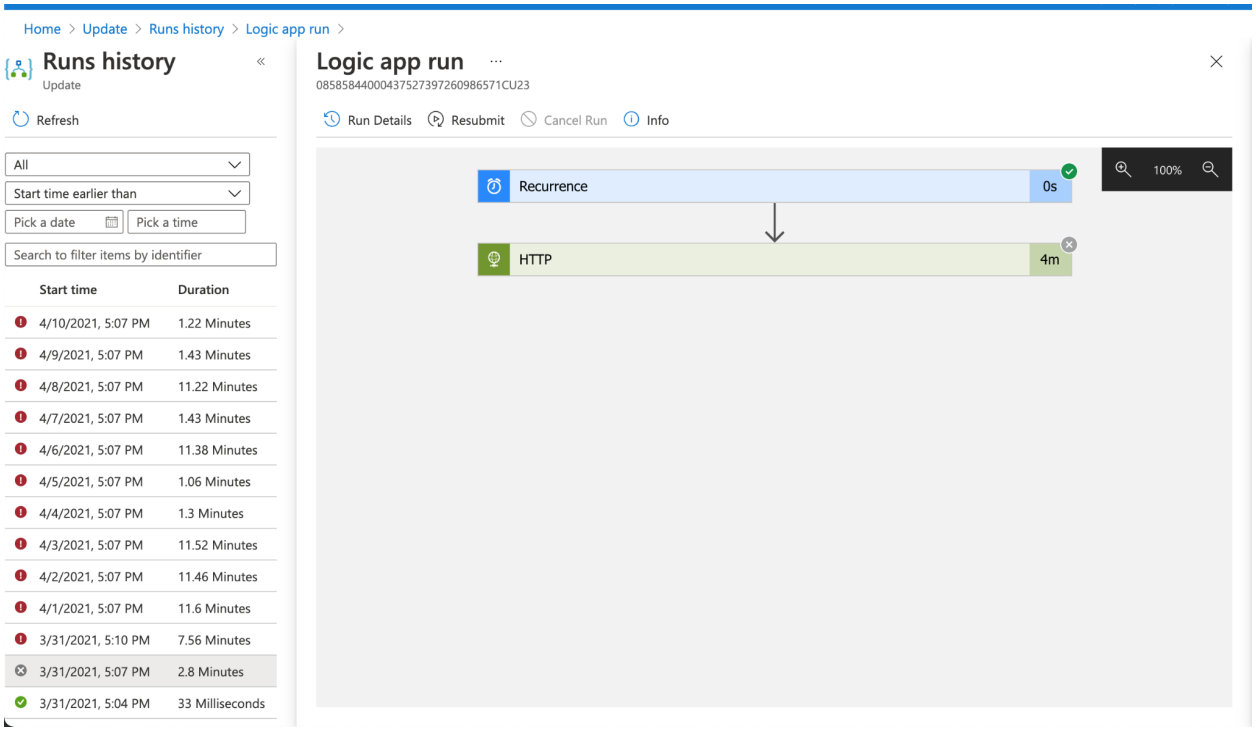


Figure 12: Azure Logic App to automate /api/update calls

I appreciate the detailed history of every past attempt to do the call, which took place every 24 hours, in an easy-to-navigate UI. The series of errors in the image above occurred due to a combination of debugging issues and the API being down for a few days. But it's still good to see how the app attempted to update the project every day. For example, here was the run summary from April 10 (Figure 13):

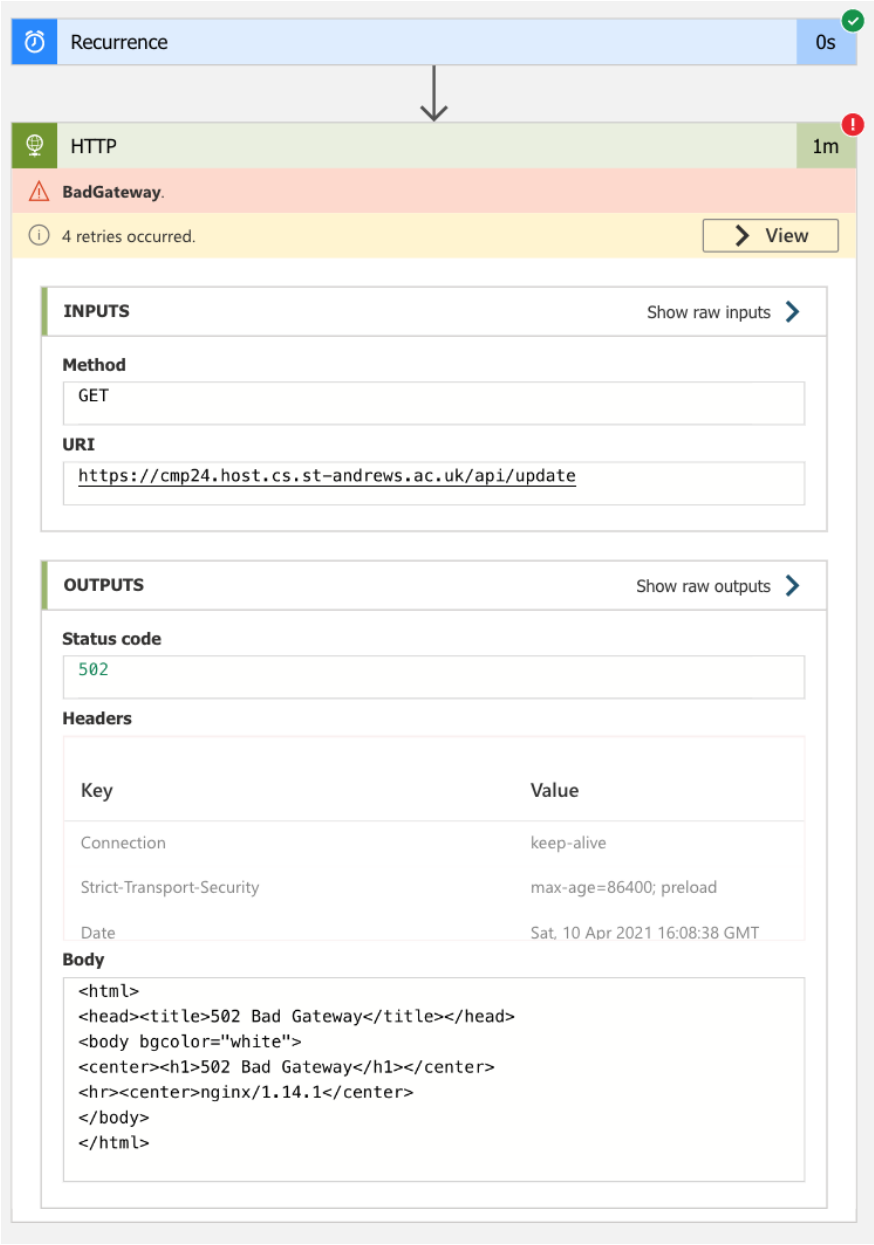


Figure 13: Error summary from Microsoft Azure Logic App

7.2.4 /api/updateRightNow

This endpoint is very straightforward, as it is mostly a slimmer version of */api/update* which updates a single podcast rather than looping through all eligible podcasts and updating all

of them. One interesting point to note, however, is that this endpoint makes use of the secret edit code to look up the podcast to be updated rather than using its id. This is because I want the user to be able to share their podcast with anyone they want, but I wanted to restrict editing rights to only the creator of the podcast. As long as the user only shares their RSS link, and not their secret edit code, they will be the only one able to add new chapters with this endpoint.

7.2.5 /api/skipPubDays

This is a very straightforward endpoint. The one important decision was how it would count the days skipped. I chose to only count days that would otherwise receive a new chapter. If I have a podcast that gets one new chapter a week, setting the days to skip to 3 will pause it for three weeks. If I get new chapters twice a week, I'd need to set it to 6 to pause it for those same three weeks.

7.3 RSS Files

Besides the issues discussed in the Design section, the biggest question about the RSS files was how I would deal with the publication date of each chapter. I wound up using Date to calculate the current date and time, and I chose to use UTC. While the time zone will not be accurate to whatever time zone the user was in when they made the podcast, it will be consistent and reliable. Since timezones are notoriously tricky to implement, I decided to leave it at that.

7.4 Database

The final database setup is the final design described in the design section. I am using the MariaDB service provided on the school servers, which is being queried from the API using the *MySQL* package. The database.js file in the api folder is the only place where connections to the database are opened. It uses environmental variables to set the host, database name, port, user, and password. In the instance of the API running on the server, these environmental variables have been set to:

- *host*: lyrane
- *database*: cmp24_librilisten
- *port:* 3306 [or left blank]
- *user*: cmp24
- *password*: [my MariaDB password]

database.js also provides some helper methods to execute queries in a few different ways such as: with a custom error message when something goes wrong, with a premade connection to a database, and without a connection (this method makes, uses, then ends its own connection to execute the query it receives). These are used as often as possible in the rest of the API when working with the database, to avoid code duplication wherever possible.

8 Evaluation

8.1 Evaluation Against Original Objectives

All primary objectives were accomplished, and a few each of the secondary and tertiary objectives were accomplished. I will go through all of the objectives from my Requirement Specification section and compare my results against them. The goals are in italics.

8.1.1 Primary Objectives

1. *An attractive, minimalist, user-friendly web application that allows users to generate a podcast feed by indicating a Librivox book and how often they wish to receive chapters (may include an About page if usage instructions are necessary)*

Yes. I did not write an About page because I deemed it unnecessary, but the podcast instructions Google Doc (linked from the Confirmation page of the UI) does provide some simple instructions for subscribing to an RSS url.

2. *Users can easily find their customized podcast in the podcast app of their choice; multiple users (e.g. a book group) can all access it from different apps*

Yes. I tested the project with several different podcast apps, and it works for all of them. See the testing section for an example of testing a podcast on multiple sites.

3. *An API server with an endpoint to generate a podcast with a specified Librivox book and publication frequency*

Yes. This is the */api/genPodcast* endpoint.

4. *A database that stores whatever data is necessary for maintaining the podcast publication schedule*

Yes. See the database design section for a description of how it works.

5. *A cron job, or something similar, which runs periodically to launch whatever activities are required to continue publishing podcast chapters*

Yes. I used an Azure Logic App. The details are in the implementation section.

8.1.2 Secondary

1. *Users may generate a podcast that begins on any chapter they choose (all previous chapters will be available on the RSS feed)*

Yes. This is not directly specified on the UI, but the user can add as many chapters as they want to their podcast by hitting the */api/updateRightNow* endpoint with their secret edit code. If the user wants to start the podcast with the first five chapters, they can generate it and then hit the */api/updateRightNow* endpoint five times.

2. *After a podcast has been created, its creator may edit the frequency its chapters are published (this requires both modification of the UI and an additional API endpoint)*

No.

3. *Gather user feedback on the product via a survey and improve its usability according to the survey results*

Yes. See Appendix A.1 for a discussion of the feedback I received.

8.1.3 Tertiary

1. *An /updateRightNow endpoint which accepts a podcast to receive a new chapter right away (useful for demoing the project as well as for letting users request a one-off chapter if they're caught up)*

Yes. This is complete.

2. *Allow users to delete their podcasts*

No. I ultimately decided that this was not important functionality, since there is no personal information in the podcasts users generate. If users get sick of their podcast, they can simply delete it from their app and forget about it.

3. *Allow users to edit publication frequency*

Sort of. Users can edit the frequency by publishing as many chapters as they want when they want bonus chapters. But they can't edit what days of the week chapters are published.

4. *Allow users to skip a certain number of chapter publications (ex: when they go on vacation)*

Yes. This is done in the */api/skipPubDays* endpoint.

5. *Give each podcast a secret edit code so that only the creator of the podcast can edit it*

Yes. This was done and is used by the */api/updateRightNow* endpoint.

8.2 Evaluation Against Related Work

Librilisten works well in conjunction with Librivox. While I received user feedback that it was a bit confusing to get an RSS link at the end of the process, this is what happens with Librivox too: you click the RSS link and it simply takes you to the RSS file. The sum total of advice for working with this file in their help page (https://librivox.org/pages/help/) is a single line that says "Subscribe to the RSS feed" (Figure 14).

Listen/Download Help
There are many options for downloading and listening to LibriVox audiobooks.

From the catalog page:

- Left Click on the Play button above the section number to play the 64kbps files individually in your browser.
- Right Click on the Play button to download the 64 kbps file to your computer to listen using another media player or to transfer to an mp3 player.
- Left Click on the Chapter Title to listen to the 128kbps files individually in your browse
- Right click on the Chapter Title to download the 128kbps files to your computer to listen using another media player or to transfer to an mp3 player.
- Download a zip file of the all the 64kbps mp3 files for an entire book by clicking on the green Download button next to Whole book (zip file). Please Note: Zip files can range in size from less than 100 MB to > 1GB depending on the length of the book. If you have a slow connection, larger files can take a long time to download.
- Click the iTunes button to download the audio files into iTunes.
- Subscribe to the RSS feed
- Download the files as a bit torrent
- M4B formats are available for most of our books. If there is an M4B format available for a book, there will be a link to the M4B catalog in our wiki under Links.

Other Options on the Internet Archive Page for each book

You can download the individual audio files as ogg files from the Internet Archive Page for each book or use the streaming player provided there to listen online. Newer books may also have VBR mp3 and a VBR zip file. Click on the Internet Archive Page link under “Links.”

Figure 14: Librivox help page [1]

As Figure 14 also shows, Librivox does provide many other ways to listen to the book. The only ways which matter when comparing to Librilisten, however, are their "RSS feed" and "iTunes" options. The "iTunes" button doesn't even work right now.

8.3 Comparison to Similar Work

There are no similar works in the public domain, so far as I can find, so there is no comparison to be done.

9 Conclusions

9.1 Possible Future Direction

This project is already functional, but it will not run forever on my host server since I'll be graduating this year. I plan to continue working on it, adding at least the remaining functionality, and hosting it myself so it can stay operational. I may also reach out to Librivox to see if they'd be interested in incorporating it into their service.

Some features which I would like to add, but which didn't quite make it into this submission, are:

- More interactive UI features on the Home page, such as calculating how long it will take to finish the book at the rate of publication chosen by the user
- Fix a bug in */api/update* which I only realised right before submission: while new chapters will only be published once a day, the *skip_next* variable for paused podcasts will be decremented every time the endpoint is called. This needs to be stopped.
- Print the title of the book on the UI confirmation page (something like "Your podcast for *Pride and Prejudice* has been created!")
- Add unit test coverage to the entire project, but especially to the API
- Add the API endpoint (and corresponding UI page) to let users edit what days they get books (they can currently only pause their podcasts and publish chapters on command)

9.2 Final Thoughts

The greatest achievement of this project is that I completed a product that is actually functional. It is a usable tool, simple and straightforward, which provides value to users. I am proud of the work I did, producing each piece of Librilisten's tech stack and tying them together to build a viable product. I dealt with a wide variety of bugs across several different languages and technologies, and I learned quite a bit through the process of fixing them.

There are no significant drawbacks to the project to which I need to draw attention. The only problems I have with Librilisten are the remaining features which I did not manage to fully implement, especially the objectives which I did not quite complete (see the Evaluation section for the list) and the UI pages to go with the edit endpoints that I wrote. I laid all of the groundwork for them, though, and am confident that they will be straightforward to implement in the future.

10 Appendices

A Testing Summary

For screenshots of the entire project working, through to subscribing on a couple of different podcast apps, see the User Manual in Appendix B.

A.1 User Feedback

Toward the end of the project, I reached out to friends and family and requested feedback on their attempts to use the product. Here are the issues they brought up, and how I addressed them:

One user noted that the "Librilisten" text was in Comic Sans for him, rather than in the font I had set it as in the CSS file. I decided not to pursue this because it's ultimately not a big deal and probably just a CSS configuration issue that would take longer than it was worth to fix.

Another user complained that the UI was ugly because it was so bare and that he found the instructions fairly confusing. I didn't want to redo my whole minimalist design, but I did make the font bigger on the UI to make it take up some more of the page. I also added more detailed instructions to both the Home and Confirmation page to make the steps more obvious to follow. After making these changes I sent it to him again and he confirmed that it worked for him, but that he found it a bit difficult to copy the title of the book because it was a link. I believe he must have been copying it from the search page (Figure 15).

Figure 15: Librivox search results for "Charles Dickens" [1]

Rather than clicking the page and then copying the non-hyperlinked title. I don't think there's much I can do about this since it's just how Librivox works, but I did make a minor modification to the UI. I changed "Go to Librivox and pick a book." to "Go to Librivox and onto the page of the book you want" (Figure 16):

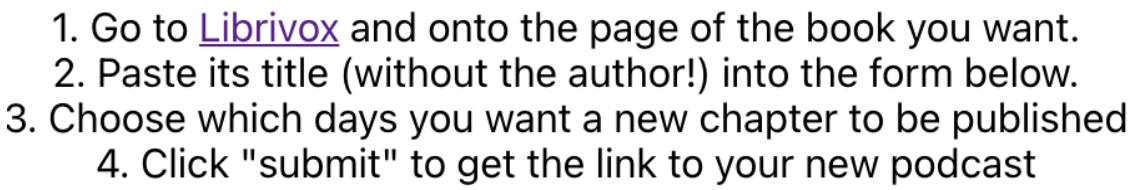


Figure 16: updated home page text

I also included instructions to visit the book page and verify that there is a list of chapters in my User Manual, Appendix B.

Another user successfully completed the podcast generation process but was confused about what to do next. She clicked the link to instructions for subscribing to a podcast, but because it was a short text file she assumed it was an error message from her computer rather than instructions. When I then manually walked her through the process of subscribing on Apple Podcasts, she was very confused about the entire process including how to find, open, and use Apple Podcasts. She suggested I make the instructions longer and more detailed, and that I include screenshots with big arrows to make it really obvious what users need to click. I appreciated her feedback, and I certainly agree that the user support elements of my UI are lacking. Several other users had similar complaints of not knowing what to do with their RSS link. If I had more time, this would be the first aspect I would focus on.

She also pointed out that the text was too small on her screen. This was after I had already increased the font size once due to user feedback. I believe this is an issue with scaling: the font stays the same size no matter the screen size, and because she was on a very wide screen the text seemed small. I added "Make font size dynamic" to the list of optional features on my kanban board, but did not get around to implementing it.

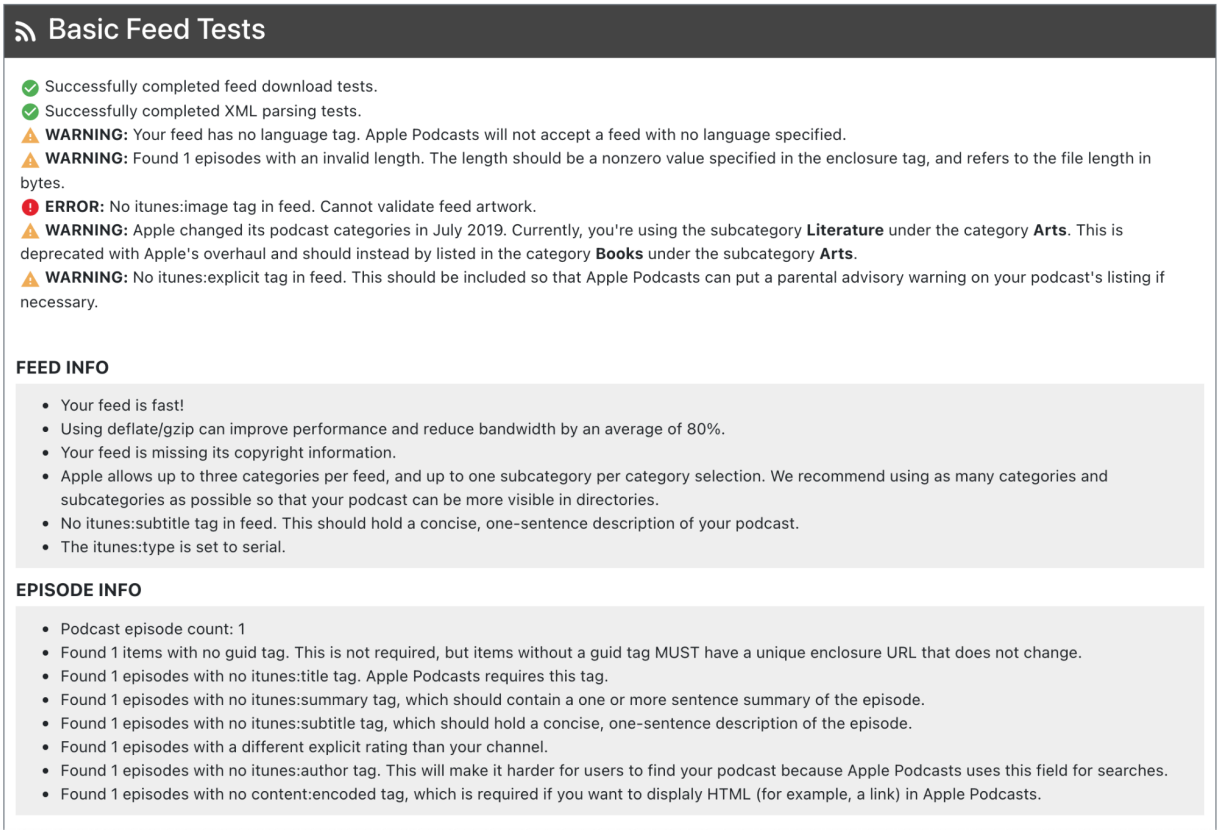
One other issue I noticed when helping this user walk through the process was that the first text she chose was not actually a book: she selected the first option she saw, which was an author (A Sister of Notre Dame*),* so the RSS url she was given was a dummy. A similar error appeared on my API console because someone else also tried to submit an author's name as the title of a book. I suspect this will be less of an issue when users are actually choosing books they specifically want to read, and not randomly pasting text from Librivox, but I have still added "make eligible titles more obvious" to my kanban board. I also added "deal more appropriately with bad API results on the UI".

I built a detailed user manual in Appendix B which addresses many of these user errors. Before fully launching the product, I will expand it and publish it somewhere users can follow it.

A.2 RSS Validation

I used Cast Feed Validator [9] to verify that the RSS files I generated were valid.

First, I tried it on a newly generated podcast for *Autumn* by John Clare. Here was the result (Figure 17):



Basic Feed Tests

Successfully completed feed download tests.
Successfully completed XML parsing tests.
WARNING: Your feed has no language tag. Apple Podcasts will not accept a feed with no language specified.
WARNING: Found 1 episodes with an invalid length. The length should be a nonzero value specified in the enclosure tag, and refers to the file length in bytes.
ERROR: No itunes:image tag in feed. Cannot validate feed artwork.
WARNING: Apple changed its podcast categories in July 2019. Currently, you're using the subcategory **Literature** under the category **Arts**. This is deprecated with Apple's overhaul and should instead by listed in the category **Books** under the subcategory **Arts**.
WARNING: No itunes:explicit tag in feed. This should be included so that Apple Podcasts can put a parental advisory warning on your podcast's listing if necessary.

FEED INFO

- Your feed is fast!
- Using deflate/gzip can improve performance and reduce bandwidth by an average of 80%.
- Your feed is missing its copyright information.
- Apple allows up to three categories per feed, and up to one subcategory per category selection. We recommend using as many categories and subcategories as possible so that your podcast can be more visible in directories.
- No itunes:subtitle tag in feed. This should hold a concise, one-sentence description of your podcast.
- The itunes:type is set to serial.

EPISODE INFO

- Podcast episode count: 1
- Found 1 items with no guid tag. This is not required, but items without a guid tag MUST have a unique enclosure URL that does not change.
- Found 1 episodes with no itunes:title tag. Apple Podcasts requires this tag.
- Found 1 episodes with no itunes:summary tag, which should contain a one or more sentence summary of the episode.
- Found 1 episodes with no itunes:subtitle tag, which should hold a concise, one-sentence description of the episode.
- Found 1 episodes with a different explicit rating than your channel.
- Found 1 episodes with no itunes:author tag. This will make it harder for users to find your podcast because Apple Podcasts uses this field for searches.
- Found 1 episodes with no content:encoded tag, which is required if you want to displaly HTML (for example, a link) in Apple Podcasts.

Figure 17: Cast Feed Validator result for a podcast with one chapter [9]

Then I published all of the chapters for the book, so that the entirety of the podcast was in the file, and ran it again (Figure 18):

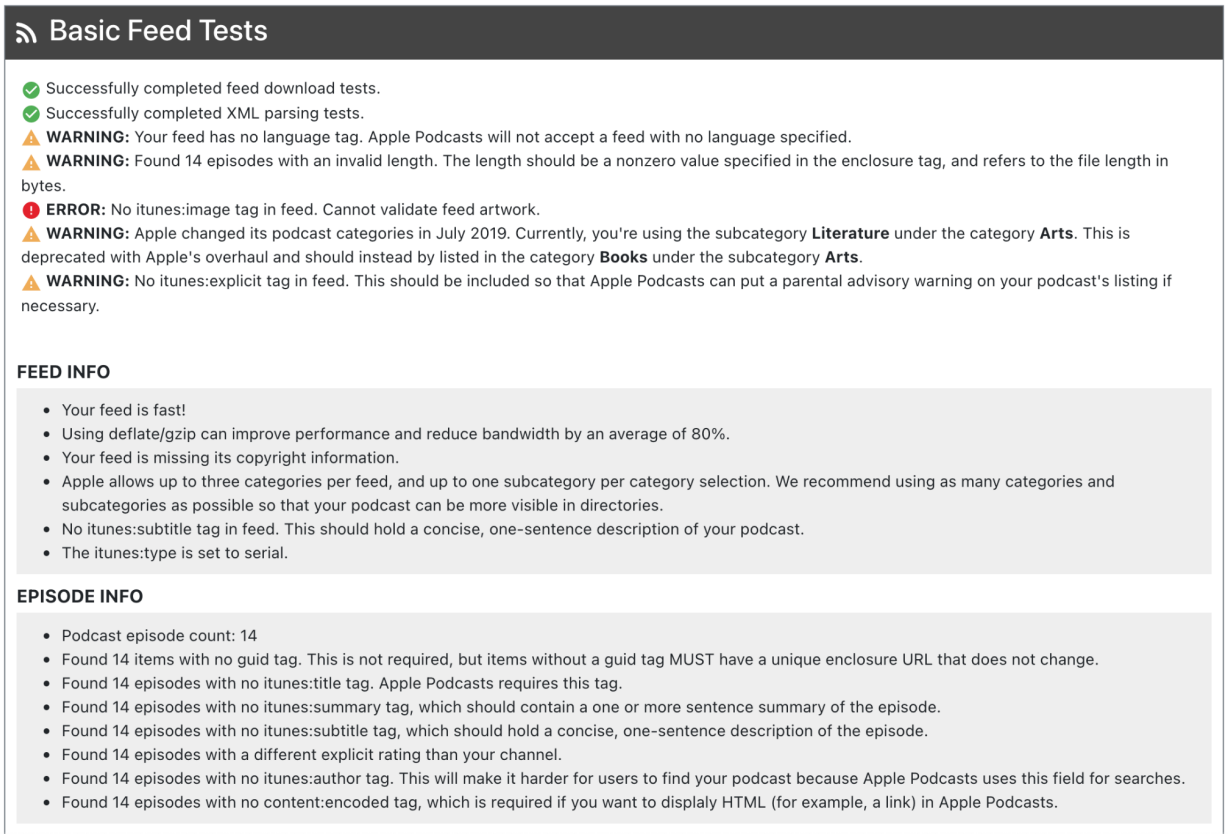
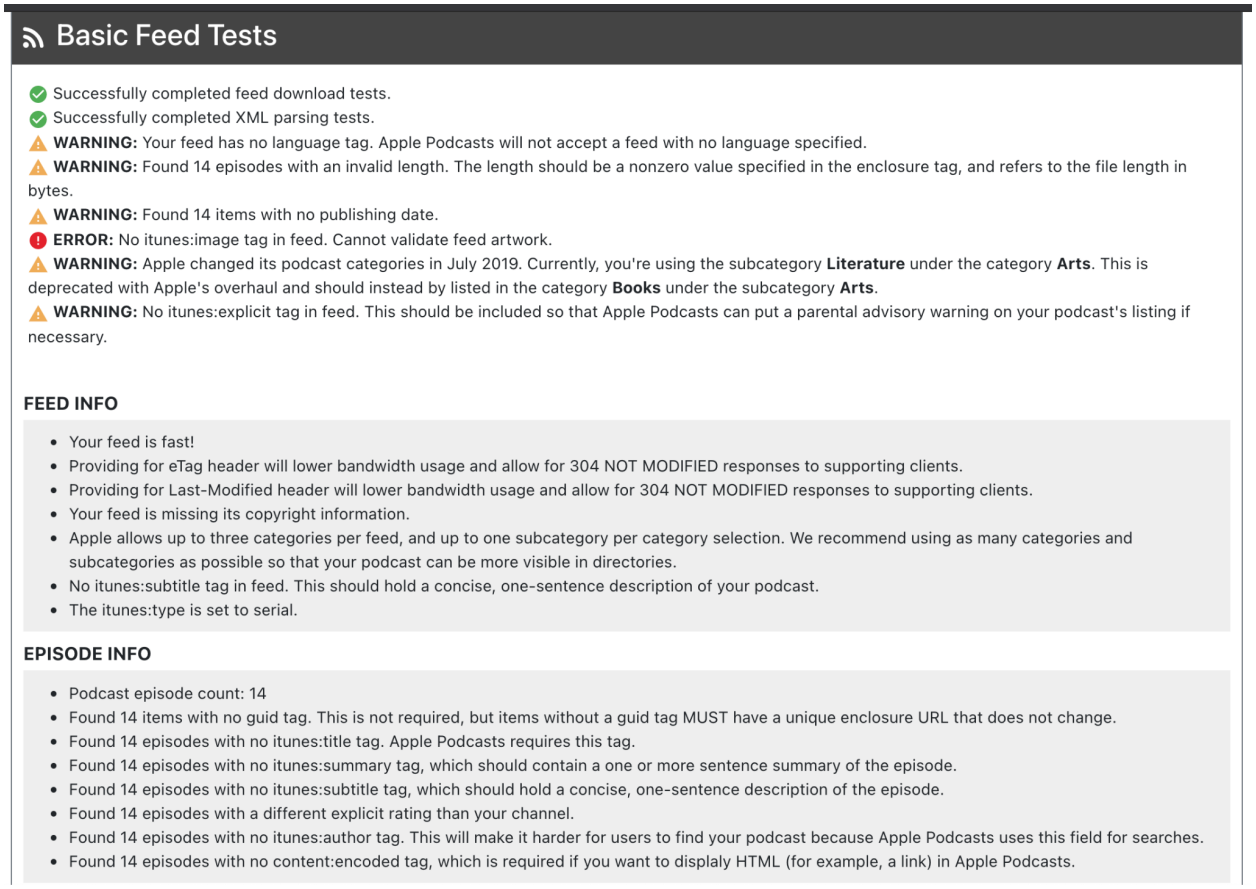


Figure 18: Cast Feed Validator result for a podcast with all fourteen chapters [9]

The results were essentially the same.

I then ran the same check on the original RSS file from Librivox, https://librivox.org/rss/4809 (Figure 19). Note that its report is almost identical to the previous report, except that it also warns there are no publication dates. This warning does not appear for the Librilisten files because I add a publication date to each one as it is published.



Basic Feed Tests

Successfully completed feed download tests.
Successfully completed XML parsing tests.
WARNING: Your feed has no language tag. Apple Podcasts will not accept a feed with no language specified.
WARNING: Found 14 episodes with an invalid length. The length should be a nonzero value specified in the enclosure tag, and refers to the file length in bytes.
WARNING: Found 14 items with no publishing date.
ERROR: No itunes:image tag in feed. Cannot validate feed artwork.
WARNING: Apple changed its podcast categories in July 2019. Currently, you're using the subcategory **Literature** under the category **Arts**. This is deprecated with Apple's overhaul and should instead by listed in the category **Books** under the subcategory **Arts**.
WARNING: No itunes:explicit tag in feed. This should be included so that Apple Podcasts can put a parental advisory warning on your podcast's listing if necessary.

FEED INFO

- Your feed is fast!
- Providing for eTag header will lower bandwidth usage and allow for 304 NOT MODIFIED responses to supporting clients.
- Providing for Last-Modified header will lower bandwidth usage and allow for 304 NOT MODIFIED responses to supporting clients.
- Your feed is missing its copyright information.
- Apple allows up to three categories per feed, and up to one subcategory per category selection. We recommend using as many categories and subcategories as possible so that your podcast can be more visible in directories.
- No itunes:subtitle tag in feed. This should hold a concise, one-sentence description of your podcast.
- The itunes:type is set to serial.

EPISODE INFO

- Podcast episode count: 14
- Found 14 items with no guid tag. This is not required, but items without a guid tag MUST have a unique enclosure URL that does not change.
- Found 14 episodes with no itunes:title tag. Apple Podcasts requires this tag.
- Found 14 episodes with no itunes:summary tag, which should contain a one or more sentence summary of the episode.
- Found 14 episodes with no itunes:subtitle tag, which should hold a concise, one-sentence description of the episode.
- Found 14 episodes with a different explicit rating than your channel.
- Found 14 episodes with no itunes:author tag. This will make it harder for users to find your podcast because Apple Podcasts uses this field for searches.
- Found 14 episodes with no content:encoded tag, which is required if you want to displaly HTML (for example, a link) in Apple Podcasts.

Figure 19: Cast Feed Validator result for original Librivox podcast [9]

While there are some concerning error messages in these validation systems, they all originate from Librivox. I do not introduce any new issues. In the future, I may go back and try to fix some of these for Librilisten.

B User Manual

B.1 UI

Note that steps 8 onward are stored in an instructions Google Doc which is linked on the Confirmation page to help people subscribe to their new podcast.

1. Go to https://cmp24.host.cs.st-andrews.ac.uk/build/index.html

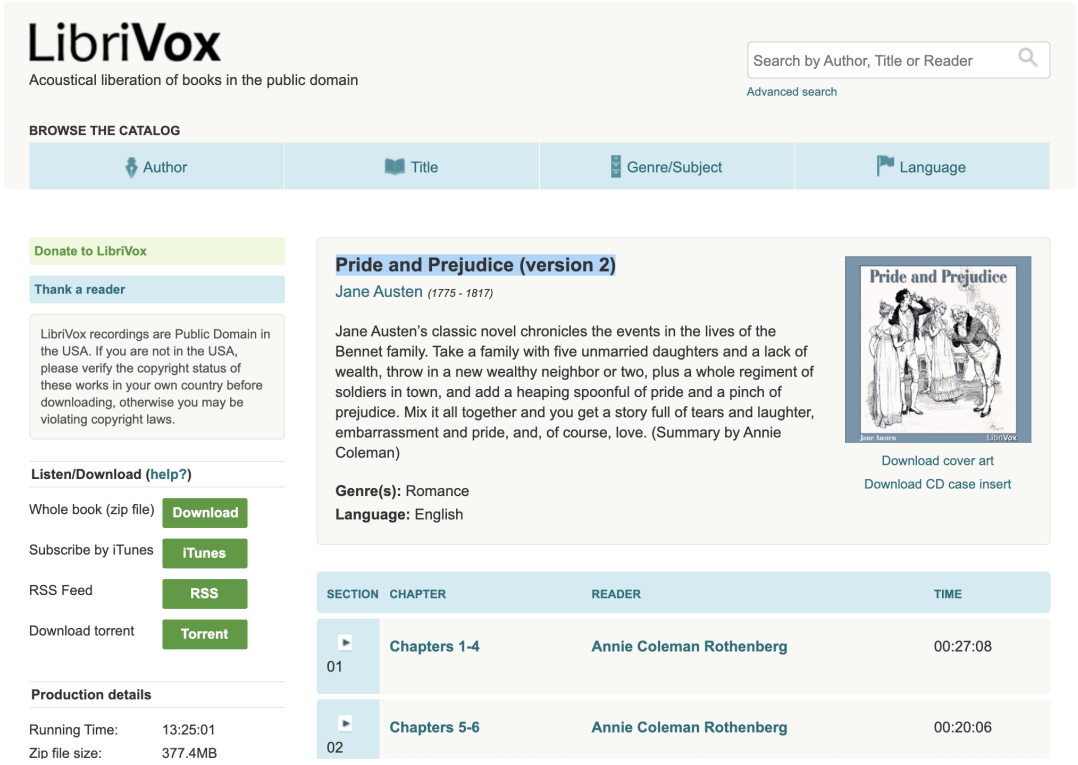
2. Click the "Librivox" link to be taken to the Librivox site

3. Browse or search the site until you find a book you want to read.

Here is an example search results page:

The first two results are **not** books in their own right, they're references to sections in other books. These will not work with Librilisten.

All book titles will be links. If you click one, you will be taken to a page that looks like this:

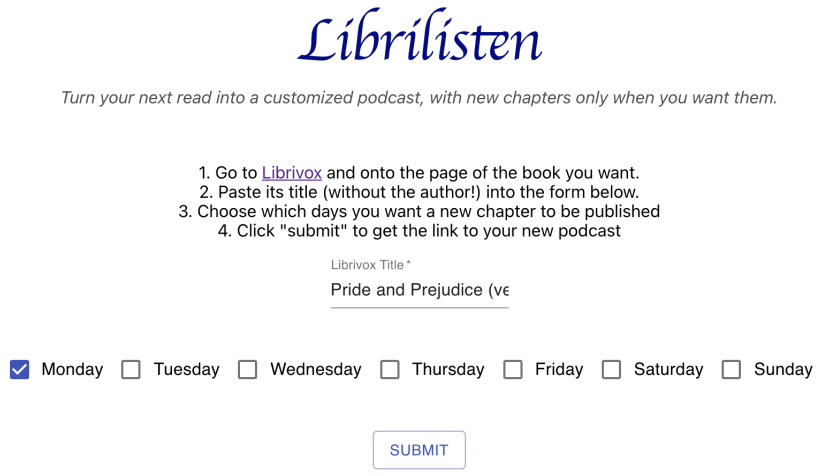


Notice that there is a table below the book, with the columns "Section", "Chapter", "Reader" and "Time". This means that I am on the page for a book.

4. Copy the title of the book (the highlighted part of the last image) by typing ctrl + c

5. Close Librivox and go back to the Librilisten web page

6. Paste the book title into the text box and select at least one day of the week when you'd like to receive new chapters:



7. Click "Submit" to be taken to the confirmation page

8. Copy the highlighted link in the following image (by either right-clicking and selecting "copy link address", or by doing ctrl + c):

Librilisten

Turn your next read into a customized podcast, with new chapters only when you want them.

You have successfully generated a podcast feed!
Click here for instructions to subscribe on the podcast app of your choice.

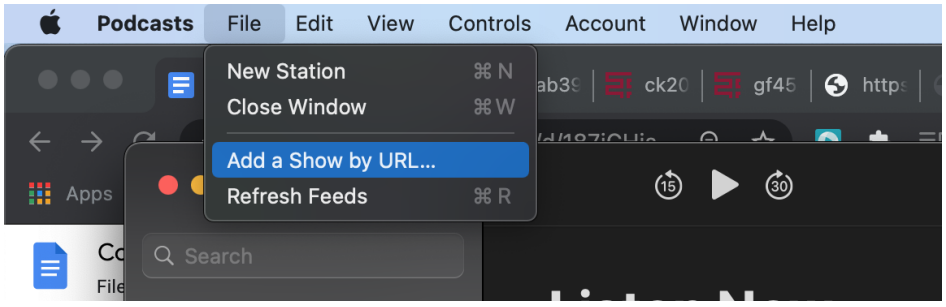
Podcast link (share with anyone who will be reading along with you!):
https://cmp24.host.cs.st-andrews.ac.uk/podcasts/3fcde310-faaa-4601-9c83-e1d1f8b1e251.rss

Unique edit code (store this and keep it secret!):
215d958e-dcb6-43bb-9c8a-5723718d2a55

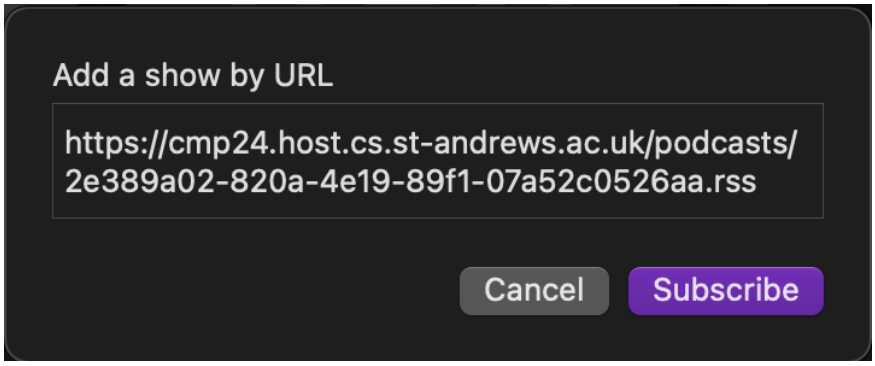
RESTART

9. Paste this into the podcast app of your choice to subscribe to your custom podcast.

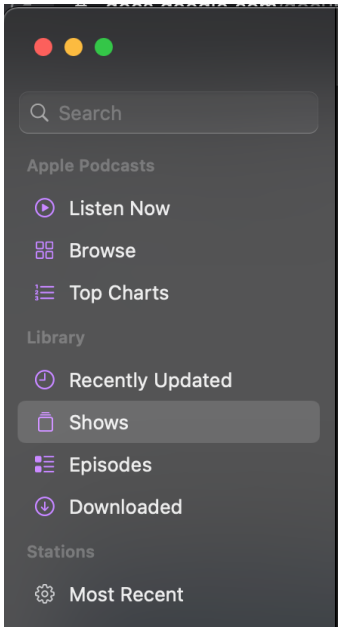
9.1 On Apple Podcasts (Desktop Version):
Click File -> "Add a Show by Url . . . "



Paste in the url you copied (ctrl + v) and click "Subscribe"

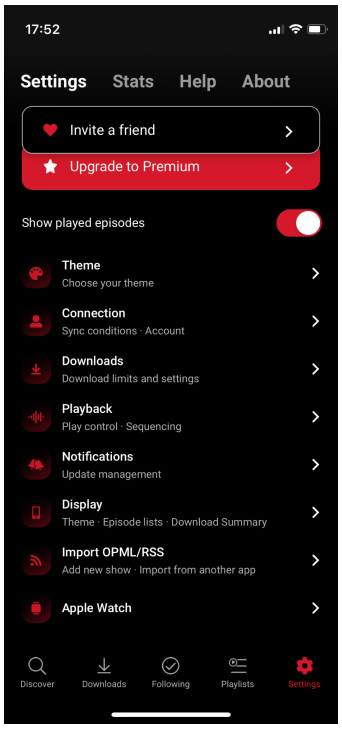


It is now in your podcast library. Click "Shows" on the left sidebar to see all podcasts in your library.

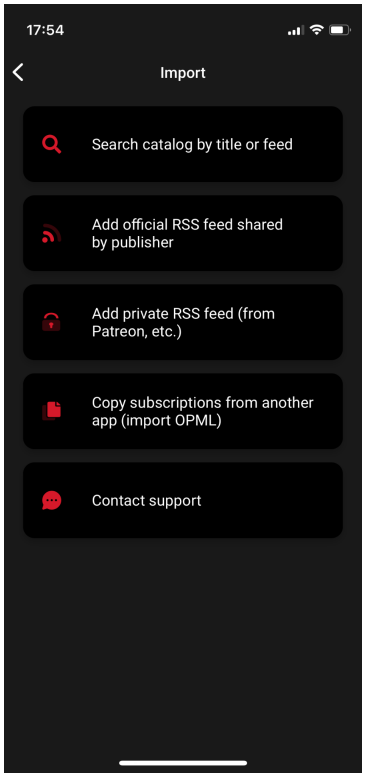


9.2 On the Player FM ios Mobile App

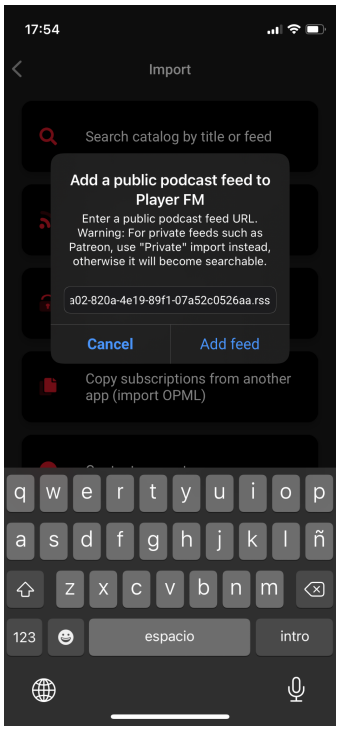
Tap the "Settings" icon on the bottom right of the screen:



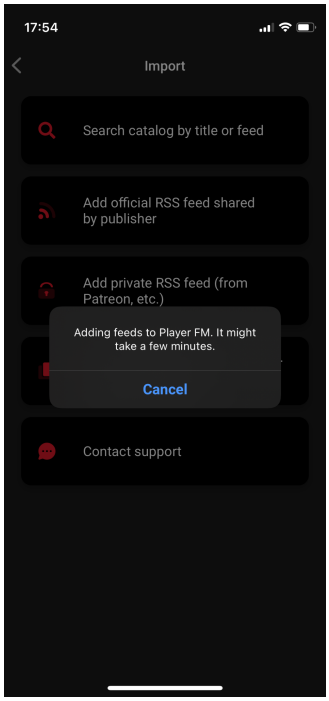
Tap the "Import OPML/RSS" option near the bottom of the screen, then tap "Add official RSS feed shared by publisher":



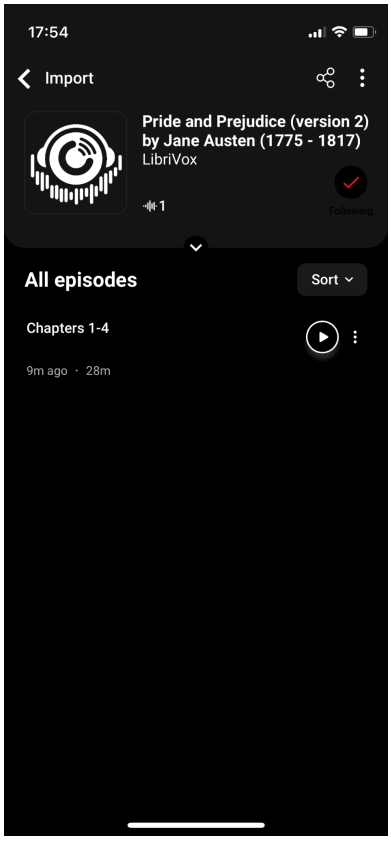
Paste in the link from the Librilisten service and click "Add Feed":



You may see a loading screen:



After it finishes loading, you are now subscribed to your own personal podcast!



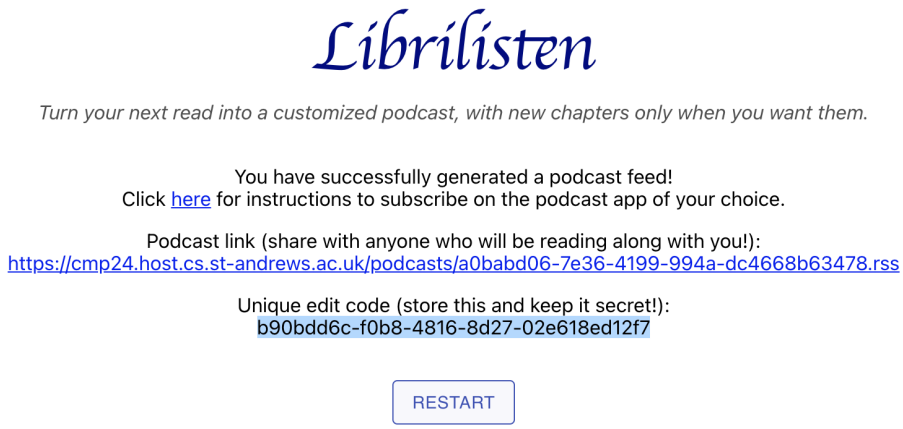
B.2 Additional API Endpoints

B.2.1 /api/update

Go to https://cmp24.host.cs.st-andrews.ac.uk/api/update to update all podcasts which are eligible to be updated today. You won't receive anything back, but if your podcast is supposed to be updated today and hasn't been it will receive a new chapter.

B.2.2 /api/updateRightNow

Take the secret edit code from the UI:



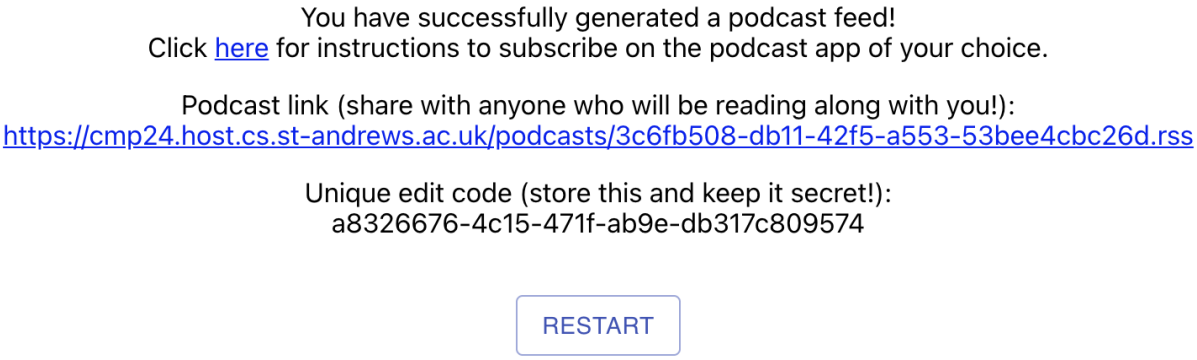
Visit the following url: https://cmp24.host.cs.st-andrews.ac.uk/api/updateRightNow/[edit code]

A new chapter will be added to your podcast.

B.2.3 /skipPubDays

This is only easily testable if you have access to the database. I will walk through the steps of testing it.

First, generate a new podcast:



Then, visit https://cmp24.host.cs.st-andrews.ac.uk/api/skipPubDays/[secret_edit_code]?toSkip=[number of

days to skip]. In my example, that is:
https://cmp24.host.cs.st-andrews.ac.uk/api/skipPubDays/a8326676-4c15-471f-ab9e-db317c809574?toSkip=3

The following message should be displayed on the screen (with a status of 200):

Your podcast has successfully been paused for the following number of days: 3

Checking in the database, you can see that this was successfully set:

```
MariaDB [cmp24_librilisten]> SELECT skip_next
 FROM librilisten_podcasts WHERE secret_edit_
code = 'a8326676-4c15-471f-ab9e-db317c809574'
;
+-----------+
| skip_next |
+-----------+
|         3 |
+-----------+
1 row in set (0.001 sec)
```

For that example, I chose to update on Thursdays. Hitting the */api/update* endpoint makes no difference to the number of days to skip.

However, I remade a new podcast that gets updates on Monday, following the same steps as above to skip three days. Then I hit the *api/update* endpoint once, and now you can see that the number of days to skip has decreased:

```
MariaDB [cmp24_librilisten]> SELECT skip_next
 FROM librilisten_podcasts WHERE secret_edit_
code = '3adbd42f-7d0b-4831-8059-598a01d90df4'
;
+-----------+
| skip_next |
+-----------+
|         2 |
+-----------+
1 row in set (0.001 sec)
```

This feature is functional on the API, just not implemented on the UI.

C Ethical Artifact Evaluation Form

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
ARTIFACT EVALUATION FORM

Title of project

CS4098 Audiobook Project

Name of researcher(s)

Cedonia Peterson

Name of supervisor

Michael Torpey

Self audit has been conducted **YES** ☒ **NO** ☐

This project is covered by the ethical application CS12476 (amended for 2019/20 due to COVID-19)

Signature Student or Researcher

Print Name

Cedonia Peterson

Date

23/09/2020

Signature Lead Researcher or Supervisor

Print Name

MICHAEL TORPEY

Date

25/09/2020

11 Bibliography

[1] “LibriVox | free public domain audiobooks.” https://librivox.org/ (accessed Apr. 12, 2021).

[2] “Tips for Creating a Podcast From Your Book | NY Book Editors.” https://nybookeditors.com/2019/05/how-to-turn-your-book-into-a-podcast/ (accessed Apr. 12, 2021).

[3] “The Ultimate 11-Step Guide on How to Turn a Podcast into a Book | Publishing Parrot.” https://www.publishingparrot.com/the-ultimate-11-step-guide-on-how-to-turn-a-podcast-into-a-book/ (accessed Apr. 12, 2021).

[4] “1,000 Free Audio Books: Download Great Books for Free | Open Culture.” https://www.openculture.com/freeaudiobooks (accessed Apr. 12, 2021).

[5] “Latest Public Domain podcasts (2021) | Listen Notes.” https://www.listennotes.com/latest-podcasts/public-domain/ (accessed Apr. 12, 2021).

[6] “Audiobooks Daily, presented by Public Domain Media | Podcast on Spotify.” https://open.spotify.com/show/0jMBisG6qQDisp7X1JB2Fu (accessed Apr. 12, 2021).

[7] “Beautiful examples of minimalist website design | Creative Bloq.” https://www.creativebloq.com/web-design/25-websites-use-minimalism-91516685 (accessed Apr. 12, 2021).

[8] “Overview for Azure Logic Apps - Azure Logic Apps | Microsoft Docs.” https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview (accessed Apr. 12, 2021).

[9] “Cast Feed Validator.” https://castfeedvalidator.com/ (accessed Apr. 12, 2021).