UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Development of a Backend for a podcast hosting platform with integration of many media services

Author:
Denys IVANENKO

Supervisor:
Maksym KORYTNIUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Denys IVANENKO, declare that this thesis titled, "Development of a Backend for a podcast hosting platform with integration of many media services" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"The medium of podcasting and the personal nature of it, the relationship you build with your listeners and the relationship they have with you — they could be just sitting there, chuckling and listening… there's nothing like that."

Marc Maron, Host of "WTF with Marc Maron"

"How has podcasting changed things? A lot of people ask me if I feel I should be more famous."

Paul F. Tompkins

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Development of a Backend for a podcast hosting platform with integration of many media services

by Denys IVANENKO

Abstract

In the last few years, a lot of different life-changing events happened. They made many people change their daily routine: all work that can be done remotely is done this way now. These changes affected people and their routine heavily: the amount of live socialization drastically decreased. People seek some replacement or imitation of live talking. In this background, the podcasting industry started to develop rapidly.

Podcasts distribution flow includes two types of platforms: podcast hosting, which actually hosts audio files, and listening services. While we have a lot of excellent platforms on the podcast consumer side, we have a wide variety of services of different quality on the creator's side. There are many complaints about creators' tooling: they are not made to be convenient for the podcasters; they are made only to generate profit. Among these problems are strange quotas connected with pricing plans, lack of convenient tools to manage, search and access resources uploaded on the platform, inconvenient ad tools, and much more.

This situation means that there is ample space to improve the solutions that the market can currently offer. My intention in this work is to solve the problems of recent podcast hosting platforms by creating a hosting platform that will include extensive automatizing of tedious work and excellent UX.

Referenced links:

- Project repository link: https://github.com/LilJohny/podcast-hosting-api

Acknowledgements

Firstly, I would like to thank my supervisor Maksym Korytniuk for advising me on the project technical stack, engineering decisions, and organizing the development and research process. He helped me a lot to design and develop a project and learn a lot from it.

Also I want to thank Roman Blahuta, my friend, student of Ukrainian Catholic University and member of the development team. We did a product research, architecture development and system development together. We also worked on finding a podcasters to collaborate with.

We want to express gratitude to Mr. Joe Lindsey, professional journalist and host of the podcast "Speak freely". He helped us a lot in the process of the research of problems in podcasts distribution.

Contents

List of Figures

List of Tables

List of Abbreviations

PWA	**P**rogressive **W**eb **A**pplication
MVP	**M**inimum **V**iable **P**roduct
CRUDL	**C**reate **R**ead **U**pdate **L**ist
GUI	**G**raphical **U**ser **I**nterface
UX	**U**ser E**x**perience
UI	**U**ser **I**nterface
WSGI	**W**eb **S**erver **G**ateway **I**nterface
ASGI	**A**synchronous **S**erver **G**ateway **I**nterface
ORM	**O**bject **R**elational **M**apping
RSS	**R**eally **S**imple **S**yndication
RDBMS	**R**elational **D**atabase **M**anagement **S**ystem
UUID	**U**niversal **U**nique **Id**entifier
TTL	**T**ime **t**o **L**ive
REST	**Re**presentational **S**tate **T**ransfer
DNS	**D**omain **N**ame **S**ystem
AWS	**A**mazon **W**eb **S**ervices
JWT	**J**SON **W**eb **T**oken
S3	Amazon **S**imple **S**torage **S**ervice

Dedicated to all my loved ones.

Chapter 1

Introduction

Podcasts are getting more and more popular nowadays. There are plenty of reasons for this: they offer a wide variety of topics and formats, are easily accessible, can be used as background noise, and give a feeling of socialization to the creators and listeners. Podcasts take the best from the television and music worlds.

There is a big difference between video and audio content. When we are talking about modern ways to create, choose and consume content, web platforms are the best. We have YouTube as a leading platform for long video content. This platform is great; it provides a simple yet powerful interface for video distribution, subscribers and watchers analytics, tools for monetizing content, and social tools. It is an excellent platform for communication between creator and audience.

Audio content is different: we have a lot of more or less globally known platforms to consume podcasts, a bunch of local sites for podcasts distribution in every country, some specialized information platforms also provide podcast-like audio content, store and distribute it, and YouTube has podcasts video versions – total mess. To add more, there are a lot of different hostings with different self-written integrations and pricing plans on the creator's side, and content monetizing is the creators' responsibility. Another problem is how hosting and podcast-consuming platforms interact – to connect these two, one needs to interact with the RSS feed directly. Interfaces of podcast-consuming platforms change sometimes; consequently, some episodes can disappear for listeners.

To solve these problems, we need to do technical research on how podcast hosting works, product research, and implementation features will not just solve the problems of podcasters but also do it conveniently. This requires generating and serving RSS feed, working with APIs of prominent services, designing an API for its own service, implementing an efficient server with all needed logic, and hosting it. The main idea of the designed platform is to be creator-centring. We plan to achieve this by collaborating closely with podcast creators while developing the service.

1.1 Motivation

The last few years changed the day-to-day routine of most people due to COVID-19 lockdowns, various limitations, and the Russian-Ukrainian war. A considerable amount of jobs is done remotely: studies, meetups, and conferences are held online. Many people lack live communication and socialization. Against this background, people are searching for new content. Video streaming services and game industry gained a big boost, and also podcasting industry started to develop rapidly. On par with these processes, local Ukrainian market of podcasting is developing very fast. From podcasts listening experience and interviews with podcast authors, it is evident that the most challenging part of creating a podcast is distribution: listening platforms are very fragmented, and working with big platforms like Spotify and Apple Podcasts requires some level of technical knowledge and doing tedious manual work. There is ample space for improvements that can be done with automation and providing a convenient interface for APIs of popular services. Developing integration layers with big services and providing a handy interface for podcast creators is not only an exciting engineering task but can also make a difference in the cultural sphere and help develop podcasting.

1.2 Technical Background on Podcast Distribution Flow

When the podcast is recorded, the author has no platform that is considered de facto standard, and there are a lot of podcast-hosting platforms. Chosen hosting requires setting up an account and podcast, uploading an audio file, and setting the properties of an episode. After this, hosting will generate an RSS feed. RSS feed is basically an XML file that contains properties of the podcast and updates if resource was changed in some way. One needs to set up the creator's dashboard on every podcast platform needed and provide an RSS feed link to the platform. After this episode will go through platform moderation and appear on the podcast platform if everything goes well. On par with podcast-specialized services, creators also often use services like YouTube, SoundCloud, or Telegram channel for podcast needs. These services are managed separately with separate interfaces, hence fragmentation of tools podcasters need to use to manage is increased by this services.

Some services require an RSS feed link, others require file uploading to the platform, and almost all of them require manual setup.

1.3 Methodology

1.3.1 Product research

The idea for the project came from reading podcasting tutorials, talking with a podcast author, and analyzing their own experience. There are a lot of platforms for listening to podcasts, and monetizing schemes for podcasting are so unclear and complicated that the reasons why podcast host do their job are often unclear to the listener.

The following steps were done to create a general concept of the project and get a list of its' needed features:

- Interviews with podcast creators in different spheres
- Testing of available solutions
- Reading articles and videos on podcasting

1.3.2 Development

The first priority was developing an MVP product that could be used to validate hypotheses for further product improvement and collaborate with podcasters on further improvements and new features.

1.3.3 Testing

As there is no significant resource to test the service, testing was done iteratively in multiple stages:

- **Development testing**

  While developing a component of the system, the developer tests it locally before pushing and deploying it.

- **Associates manual testing** Associates, who were familiarized with the user flow of the system, but not with podcasting, were involved in testing system work-ability and UX. Their feedback helped us to improve the system's UX.

- **Testing of the stable version of the system by podcast authors**

  Podcast creators used our system to host their podcasts and gave their feedback on the experience.

1.4 Goals of bachelor's thesis

Thus, the purpose and objectives of this study are:

- Explore needs of podcasters.
- Explore available solutions.
- Identify key pain points of podcasters.
- Design a service that satisfies as many needs of podcasters as possible.
- Develop a podcast hosting service.

1.5 Contributions

- To our knowledge, the first product of a kind on local market.
- Strict structure for uploaded content separated by Shows and Series of the Show.
- Searching system for creator's content.
- Express recording functionality.
- Automated verification of episode moderation.
- Integration with very popular services for supporting creators on regular basis.
- Integration with local platforms.

1.6 Structure of the thesis

- Chapter 2. **Research**

  This chapter contains research results on points of pain podcasters experience while distributing their content.

  Also, this chapter contains a brief description of existing solutions and their pros and cons.

- Chapter 3. **Proposed Solution**

  This chapter contains a description of features that can solve most podcasters' problems discovered in the research process and technical decisions made to implement needed logic.

- Chapter 4. **Perspective**

  This chapter contains the vision of further project development, both technical and product aspects.

- Chapter 5. **Conclusions**

  This chapter contains summarizing and conclusion on what was done in the scope of this work and a review of achieved results.

Chapter 2

Research

2.1 Market Research

Market research was held to determine features that are needed for our solution. Solutions that are trying to solve similar problems are listed below.

2.1.1 Anchor by Spotify

Anchor is a mobile app that provides tools to record a podcast on the go via a phone's microphone and publish it to the most prominent platforms. It also integrates ads and podcast hosting functionality. The problem here is that all podcasts are published on Anchor's account; hence does not belong entirely to the author. This service lacks configurability and integration with less known or local platforms and social channels but is a good choice for beginners due to its simplicity.

TABLE 2.1: Anchor features summary

	Anchor by Spotify
Simple recording/editing studio	Yes
Own/integration with ads platform	Yes, own
Flexible pricing plans	Yes
Mobile version	Yes
Integration with social platforms	No
Author fully owns the content	No
Own/integration with membership service	No

2.1.2 RSS.com

RSS.com is a service that provides hosting and integration with big listening platforms and some less known platforms but lacks integration with social channels. It provides monetizing features, but only for the most expensive pricing plan. There are also limitations on the number of podcasts for one author. Also, getting the best pricing plans requires contacting support of the service. This hosting has strange limitations depending on pricing plans and lacks integration with local listening platforms. RSS.com is a good choice for people with a medium amount of experience and technical skills.

TABLE 2.2: RSS.com features summary

	RSS.com
Simple recording/editing studio	No
Own/integration with ads platform	Yes, integration with Podcorn
Flexible pricing plans	No, just subscription plans
Mobile version	No
Integration with social platforms	No
Author fully owns the content	Yes
Own/integration with membership service	No

2.1.3 Podbean

Podbean is a service that provides hosting and integration with big listening platforms and some social channels: Facebook, Twitter, and YouTube. This service also has its own monetizing platform, but all propositions are actual for local markets. The pricing plan limits access to the monetizing platform, which is strange. Also, Podbean lacks integration with more globally known and used membership services, for example, Patreon.

TABLE 2.3: Podbean features summary

	Podbean
Simple recording/editing studio	Yes
Own/integration with ads platform	Yes, own
Flexible pricing plans	No, just subscription plans
Mobile version	No
Integration with social platforms	Yes, partly
Author fully owns the content	Yes
Own/integration with membership service	Yes, own

2.2 Problems

Today recording a podcast takes less effort than its distribution. Distribution of the podcast is a problem, not only the first time one is doing a podcast. One can either set up their own local server and store the file there or use one of the available hosting platforms. A lot of these podcast hosting services are ancient and have many problems. After researching solutions available on the market and interviewing podcast authors following problems were singled out.

2.2.1 Variety of Admin panels to set up

Every service that can be used via RSS feed requires an additional setup from the author. Setting up a show on every platform and communicating with support if needed is a lot of tedious work.

2.2.2 Interface Updates

The accepting side can update its interface in this distribution flow without notifying the other side. Something similar happened recently with the Apple Podcasts platform: a big update was rolled out, the interface changed, and consequently, some podcasts or single episodes were unavailable for some time.

2.2.3 Support in case of problems

Even if the podcast is set up correctly on the platform, the RSS feed is appropriately generated and passed to the podcast listening service, it needs to pass through content moderation and appear on the platform. There are known cases when the podcast author needs to contact the listening platform support for the podcast to appear/update on the platform. The problem here is that an author needs to check the platform as a listener to find out about any problem of such type.

2.2.4 Pricing Plans

Existing hosting solutions have strange restrictions on the file storage available a month. For example, one of our researched solutions has a 400 Mb quota per month. This way creator is either limited to chosen pricing plan quotas or pays more to have more storage that can be not used next month. Creators need to compress audio, which leads to worse audio quality or produces less content than they want. There are also quotas on download numbers or episodes number a month. As these are variable things, it is not easy to choose the right plan.

2.2.5 Monetizing

Most podcast authors earn almost nothing for their shows. This problem is mainly caused by a wide variety of places to consume podcasts; consequently, attracting sponsors is extremely difficult due to the inability to count the numbers of unique listeners and get analytics on listeners' interests. Due to this problem, many great creators stopped producing exciting content.

2.2.6 Migration

Another problem is migrating podcasts to new hosting. Two issues can be found here: one needs to take every audio file from the old hosting and move it to the new one. After such migration, there is a significant probability of losing proper episode numeration. Due to this, the podcast author is tied to the service chosen some time ago.

Chapter 3

Proposed Solution

3.1 Idea

In the research process, it was determined that most of the described problems could be solved by adding more automation to the manual processes and providing integration with big, globally known services. The project's main idea is to create a podcast hosting service that will also be a hub that will provide integrations with a considerable amount of services to solve as many podcast authors' problems as possible. Interfaces of prominent services cause some limitations, but there is ample space for improvement in usability for modern podcast hostings. In the scope of users' experience needed, maximum configurability with the most straightforward option as default is a priority. This way service will be usable for both experienced users and computer newbies. So, we want to automate as much stuff as possible and also be a great choice for computer and podcasting newbies while having ability to configure everything.

3.2 Minimum viable product

The first thing needed to implement is MVP or Minimum viable product. Selected features for this version of the product are the following:

- Podcast storing system with strict categorization by Shows and Show Series.

  One of the features that differ our podcast hosting from most of the existing solutions is providing a strict structure to the user's data. In our system user has a Show or podcast; this show contains episodes. Show also contains a list of series. Functionally, series is a tag that can be assigned to the episode. This will give the user ability to group some episodes of the show into some groups. This way, it will be easier for creators to do some special formats for their shows. For example, this way, creators can separate special episodes for people who support them with a subscription.

- Content searching and filtering.

  This also differs our solution from existing, as most podcast hostings do not expect authors to have a really massive number of episodes: they don't have a comfortable searching system for the content user hosts on the platform.

- Mailing.

  This includes setting up a custom domain for mailing and implementing asynchronous mailing logic.

- Password renewal.

  This includes implementing logic to recover account credentials in case the user forgot his password by providing the JWT token that was sent to the mail of the user.

- User system with account verification.

  This includes implementing logic to verify user's account by sending the JWT token to the email and validating it via the system.

- Creators dashboard functionality.

  This is a basic screen for the creator with access to accessible features of the platform.

- Generation and serving of adequate RSS feed for podcast serving.

3.3 Further features

There is an extensive list of features that are needed to lighten the podcasters' burden.

This features are:

- **Automatic episode submission where possible.**

  Most listening platforms work via RSS feed and require manual passing of an RSS feed link, but some support automatic episode submission.

  Supporting such ability will significantly improve UX.

- **Notifications of problems with episode publishing on listening platforms.**

  Each episode submitted should go through listening platform moderation and can be removed/modified by administrators. The creator is not notified if something is changed on any listening platform. There are two ways for the creator to discover the issue: check the podcast as a listener or get a ton of emails, messages, and comments with listeners complaining.

  Problems with presenting episodes to the listeners can be caused not only by changes from administration but also by interface updates on the accepting side.

  The solution here is to create a "watcher" job that will notify them via email if something goes wrong with a podcast episode on any of the platforms. If episode issues are not massive, the only creator will be notified, and this way author will be able to react rapidly in case of problems. If issues are present with a significant number of episodes, developers will be notified, too; this will mean that the interface on the accepting side is updated, and some work is needed to support it.

- **Sharing page.**

  Due to the segmentation of podcast-listening services, finding the podcast on the platform one needs is difficult, but promoting a podcast is also complicated. Creators need to give links to all platforms every time promotion text is shared, or hope listeners will remember the podcast title, open the platform of choice, and find the platform. Modern people usually do not like to do such amount of job.

Another use case is finding a particular episode of a podcast. In this case, listeners either need to remember the podcast's title and the episode's title, but the episode title is not always informative and unique enough to identify one.

The solution here is to generate a one-pager website that will contain links to listening platforms and optional social channels. This one-pager can also solve the problem of finding a particular podcast episode. This also gives the creator the ability to easily promote every episode separately and share it on social media.

- **Grace period while deleting user resources.**

  If a user wants to delete some resource that he created in the system, the resource can be restored until the grace period is active.

- **Search by name among episodes of the show and shows of the user.**

  Existing solutions on the market lack categorization and handy management tools for administering a podcast. Search for shows and episodes will significantly improve UX for users who have a large amount of content to host.

- **Integration with services that require file uploading.**

  A lot of podcasts have video and audio versions. So, to support both versions, the creator needs to prepare two files and upload them on different platforms.

  The solution here is to implement integration with YouTube and SoundCloud with the ability to use the audio part from the video uploaded to YouTube as a source for audio-only platforms.

- **Integration with membership platforms.**

  Relatively small podcasts often struggle to find sponsors and can stop producing exciting content; to avoid this, listeners can support them financially to motivate creators.

  Integration with membership services like Patreon and Buy Me a Coffee will help support small creators even in case of sponsors' absence.

- **"Bridge" for easy migration from other hostings.**

  Podcasters need to do much tedious manual work if they want to migrate from one hosting to another. In this case, they need to move every file to new hosting manually. Also, they can experience numbering problems when moving files is finished.

  Implementing the ability to move files from other hosting with a user-friendly interface will significantly improve UX.

- **Simplify Admin panels setup on listening platforms.**

  Creators need to set up every platform and fill in many data that was already entered on the hosting platform, which is tedious manual work.

  Implementing an automatic setup of Admin panels will significantly improve UX.

- **Flexible pricing plans with the ability to bypass the quotas once if necessary.**

  Modern platforms for podcast hosting use subscription models as a monetizing model. This implies fixed quotas on creators' abilities. They need to compress or shrink audio files or produce less content.

The solution here is to add the ability to exceed quotas for additional pay. The service will have no strange limitations for situations when creators need extra.

- **Integration with ads-platform to give creators the ability to cooperate with sponsors.**

  Integrating with ad platforms will help podcasts with a big audience to monetize — this way, the creator will be paid for his work.

- **Mobile version of website/ PWA application.**

  This will not just improve UX but allow to implement of audio recording and editing tools and use them in various circumstances.

- **Express podcast studio.**

  Implementing tools for recording a podcast and simple editing will help lower the level of technical skills needed to record a podcast.

  This feature will also allow experienced users to do an express recording if needed. This way, a more comprehensive set of topics will be available for podcast consumers. Also, such functionality will ease the creation of gonzo content.

- **Listeners analytics.**

  Creators need analytics on podcast listeners to analyze audience responses to their actions and attract sponsors.

  Implementing analytics for the listening platform and aggregating it will significantly improve UX.

- **Social platforms integration.**

  Creators also want to get in touch with their audience.

  Implementing integration with platforms like Telegram will significantly improve the platform's UX.

- **Integration with local listening platforms.**

  Local platforms often produce quality content and attract great creators.

  Implementing integration with local platforms will significantly improve the platform's UX.

- **Admin panel.**

  With the growth of the service auditory, it will be helpful to have an administrator who can help to solve simple users' problems. These kinds of people need some UI; hence admin panel will be handy.

- **Replacing session tokens with JWT tokens**

  Migrating from authentication with session tokens and storing them in the database to JWT tokens will reduce disk space usage and improve the overall speed of the service. We don't need to store JWT tokens in any kind of storage because the data inside is self-contained and cryptographically signed.

- **Account search in case of forgotten email address**

  We should consider a case when the user can't remember the email address they used to register an account on the website. In this case, we want the user to have the ability to find an account and get the email he used.

- **Secondary email and changing the primary email address tied up to the account**

  We need the ability to add a secondary email address that can be used in the case when the user loses access to one of the email addresses. This will also allow us to change the primary email address in case if the user wants to.

3.4 Technical Decisions

3.4.1 DBMS Choice

DBMS Type

There are two types of DBMS: relational and non-relational. Non-relational databases suit the needs of working with less structured/confined data. This implies more configurability and flexibility in storing and modifying data. Relational databases mean a firm way of storing data as tables, rows, and relations between them. Good tables design will allow us to quickly sort, filter, and do computations. Another essential point is ACID compliance. Relational databases have a higher level of ACID compliance. On the other hand, non-relational databases work well for simple queries. But the normalized database will deliver similar or even better performance. Another difference between these types of databases is that relational databases are vertically scalable, and non-relational databases are horizontally scalable.

Most of the project functionality is tied around three entities: User, Episode, and Podcast. These entities will not change their structure, and relationships between them can be defined quickly and firmly. Also, podcast hosting will include filtering data and needs reliable and persistent data storage. Hence, relational databases are the right choice for the project.

DBMS choice

Amphora will use a relational database, and now we need to choose a DBMS system. Let's consider SQLite, MySQL, and PostgreSQL.

SQLite

Advantages of SQLite:

- Serveless

  Due to this, it is effortless to set up and needs almost no configuration.

- File-based system

  Due to this is highly portable.

- Great for testing and prototyping.

Disadvantages of SQLite:

- Does not support access via network.
- Is not made for large data storing.
- Does not support user management.

Conclusion
SQLite is a small DBMS that is an excellent choice for educational projects or prototyping and development in the first stages. But it is not made for developing actual production applications.

MySQL

Advantages of MySQL:

- Free and open-source.
- Support of replication and sharding.
- Available for all major platforms.
- Covers a wide range of DBMS use cases.
- Stability and wide usage.
- Speed.

  MySQL is very fast on read operation.

Disadvantages of MySQL:

- Has reliability problems.
- Badly supported by Oracle.
- Medium concurrency support.

Conclusion
MySQL is a relatively old and widely used RDBMS. It suits the need of production apps with simple, read-heavy data workflows, as it is optimized the best for reading operations.

PostgreSQL

Advantages of PostgreSQL:

- Object-oriented database management system.

  This means it is a hybrid SQL/Non-SQL DBMS.
- Free and open-source.
- Compatibility with a wide range of operating systems and services.
- Pure SQL as query language.
- High level of ACID compliance.
- Optimized to work with extra-large databases and execute complicated queries.
- Can handle cases when data does not fit the relational model entirely.
- Stability.
- Support of advanced datatypes.
- Widely supported by Heroku.
- Effective concurrency support via MVCC.

  PostgreSQL implements Multiversion Concurrency Control algorithm, this allows to use PostgreSQL concurrently in a non-blocking way.

Disadvantages of PostgreSQL:

- Can be a resource hog.

  On every new connection, PostgreSQL forks the main process and allocates 10MB of memory to the new process.

- Less effective on reading queries then MySQL

Conclusion

PostgreSQL is an excellent choice for production-scale services as it provides flexibility in storing data, and powerful query language, supported by big amount of operating systems and services. This is also a great choice because of its hybrid nature, support of advanced datatypes, stability, speed, and great extensibility.

DBMS Conclusion

SQLite definitely does not suit the needs of production version podcast hosting due to the data amount needed to store. MySQL and PostgreSQL are difficult: MySQL is older; hence has more third-party tools available; performance of these two are almost equal, but MySQL is a bit better in reading operations, but there are benefits that Heroku provides for users, who choose PostgreSQL, PostgreSQL also supports MVCC, advanced types and has a hybrid nature, which in general helps to store any kind of data and access it concurrently and fast. So, taking all the above into account, read operation efficiency is not enough to outweigh the benefits of using PostgreSQL.

3.4.2 Database Schema Design

Primary key

The first decision that needed to be made was choosing the type for the primary key: integer or UUID.

Advantages of UUID:

- UUIDs are unique across all tables in the system and applications in general.

- UUIDs are more portable then integers.

  With UUIDs, there is less probability of collision while migrating data.

- UUIDs are generated before insert time.

- UUIDs do not expose internal information.

  The frontend of the application needs some way to interact with the backend. This means that systems that use integers as the primary key might become vulnerable to brute-force attacks—using UUIDs as the primary key will help protect systems from such attacks.

Disadvantages of UUID:

- UUIDs are longer, hence take more memory to store.

- UUIDs aren't ordered.

- UUIDs might sort slower then integers.

Primary Key Conclusion

The system does not imply sorting entities by primary key, and using a bit more space is not a problem for the system. Hence disadvantages of UUIDs are not painful, especially counting in advantages that they provide.

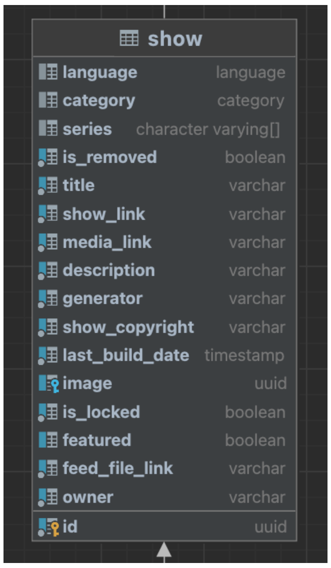
So, UUID is an excellent choice in current circumstances.

Tables design

Podcast/Show

The first entity to design is the main for the project – podcast entity. To be more technical, let us use the term show. Defining the initial set of fields here is easy – we need to include data needed to generate an RSS feed for the show.

There are two specific fields here – image and podcast owner. We want users' images to be reusable in the system – let us move them into a separate table and reference them in a show table as a foreign key. The owner field will be simply a UUID of the user who owns the show. There is also one field that will help us make the system strict and categorized – series. PostgreSQL supports string arrays, which is the perfect choice for such tag-like logic.



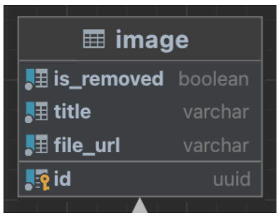
show

language	language
category	category
series	character varying[]
is_removed	boolean
title	varchar
show_link	varchar
media_link	varchar
description	varchar
generator	varchar
show_copyright	varchar
last_build_date	timestamp
image	uuid
is_locked	boolean
featured	boolean
feed_file_link	varchar
owner	varchar
id	uuid

FIGURE 3.1: Show table in database

Image

The following entity we need is the image. This entity is quite simple – the only thing we need here is a direct file link and the title of the image that will be used as its name.



image

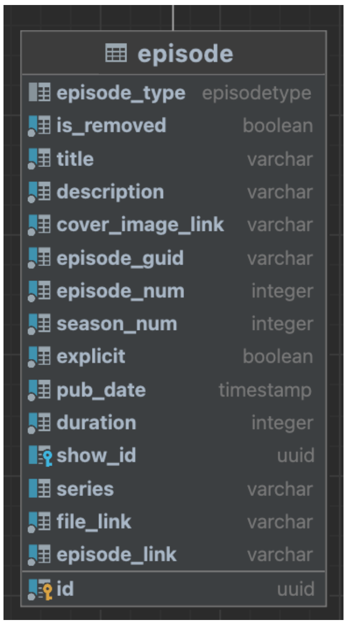
is_removed	boolean
title	varchar
file_url	varchar
id	uuid

FIGURE 3.2: Image table in database

Episode

For this entity, we also need some fields that are needed to generate an RSS feed.

One specific decision is made – store the link to the cover image in the table, as this will take away the need to join the images table when we want to display episodes. As the number of episodes can be vast, this will decrease database load and keep queries simple. The episode must belong to a show; hence we need a foreign key that will reference a show row.

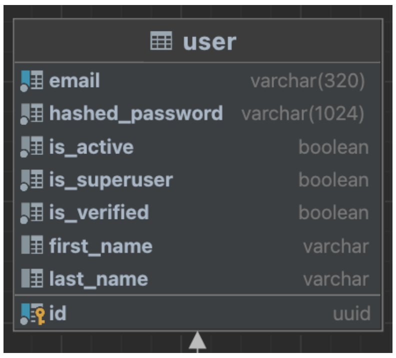


episode	
episode_type	episodetype
is_removed	boolean
title	varchar
description	varchar
cover_image_link	varchar
episode_guid	varchar
episode_num	integer
season_num	integer
explicit	boolean
pub_date	timestamp
duration	integer
show_id	uuid
series	varchar
file_link	varchar
episode_link	varchar
id	uuid

FIGURE 3.3: Episode table in database

User

This entity is plain simple as we do not plan any social features. We want users to provide unique email, their first and last names, and need to store their passwords safely. We also want only real users to have the ability to store their files with the service; hence we need the "is verified" field.

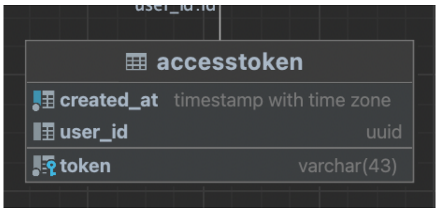


user	
email	varchar(320)
hashed_password	varchar(1024)
is_active	boolean
is_superuser	boolean
is_verified	boolean
first_name	varchar
last_name	varchar
id	uuid

FIGURE 3.4: User table in database

Access token

We use JWT tokens to authenticate the user on their attempt to log in with their credentials. These JWT tokens are stored inside a database table called "accesstoken".



accesstoken	
created_at	timestamp with time zone
user_id	uuid
token	varchar(43)

FIGURE 3.5: Access token table in database

All tables and their relations

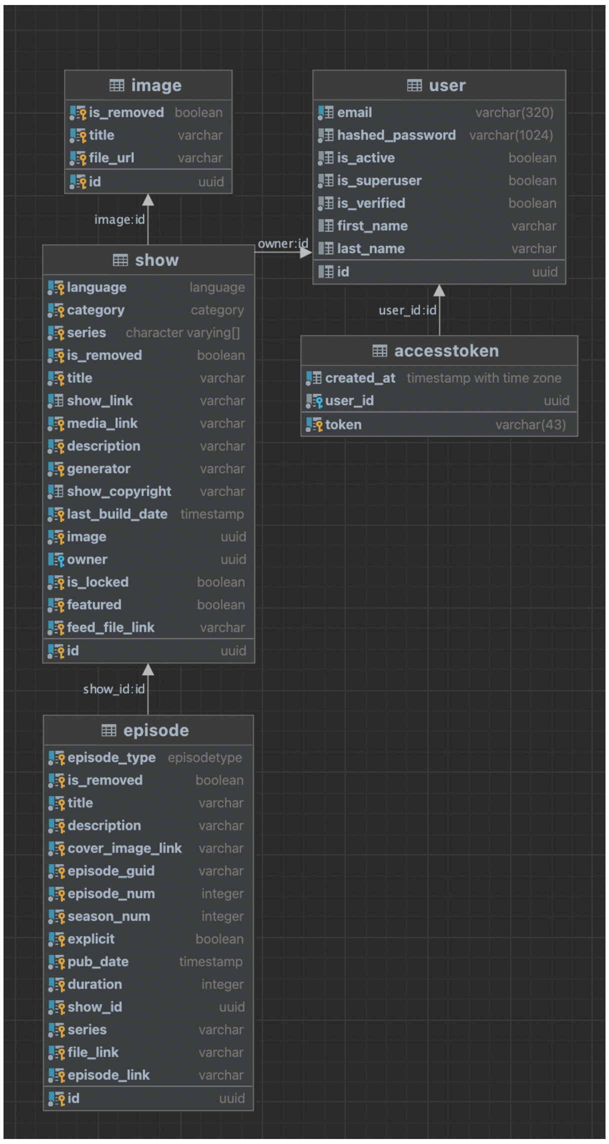


FIGURE 3.6: Database schema

3.4.3 RSS Generator and Server

Definitions

RSS is a Web content syndication format. On a technical level, RSS is an XML dialect; hence it should conform to XML 1.0 specifications.

An RSS feed is an RSS document that informs the reader about updates that happened on some resource.

RSS generator

Custom RSS generator was implemented with the usage of Python std xml package.

RSS server

To serve the generated file in a way that the accepting side can read properly, we need to serve it with a content-type application/xml. On the creation of a new show in a database, systems start to serve a new RSS feed.

3.4.4 Framework Choice

Available Choices

Before development started, there was three main option for a web framework to use: Flask, Django, and FastAPI.

Flask

Flask is a microframework developed in 2010. This framework support only synchronous execution. The main feature of this framework is flexibility and lightweightness. It is classified as a microframework because it does not includes a database abstraction layer, form validation, or any other additional components; hence it does not implies any dependency installation.

Advantages of the Flask:

- Simplicity and minimalism of the framework
- The Flask developed application is more scalable than the monolithic application.
- Used extremely widely.

Disadvantages of the Flask:

- Supports only synchronous execution.
- Requires a lot of dependencies to implement needed logic.

Conclusion

Flask is an excellent microframework for developing applications of all sizes when flexibility is the main priority. A lot of compatible packages are available that provide various functionality.

TABLE 3.1: Flask summary

	Flask
Asynchronous execution support	No
Bundled ORM	No
Performance	Same as Django in synchronous mode, slower then FastAPI
Wide usage	Yes
Background tasks support	No
Automatic request validation	No
Automatic documentation generation	No

Django

Django is a web framework developed in 2003. This framework recently added support for asynchronous views but did not fully support asynchronous execution. The main feature of this framework is "batteries included" concept: one can find much logic needed and even full reusable apps available for usage. This feature also implies a disadvantage – need to include all apps and logic available in small projects; this can lead to excessive distribution package growth.

Advantages of the Django:

- Batteries included.
- Powerful ORM included.
- Extensive contrib package.
- Automatic security against most types of web attacks.
- Support for asynchronous views.

Disadvantages of the Django:

- Bad performance.
- A lot of unused code in case of usage in small projects.
- Inconsistent support of asynchronous execution.

Conclusion

Django is an excellent framework for developing big platforms with various functionalities due ability to reuse applications and a considerable amount of logic provided out-of-the-box. Despite the fact that Django 4.0, the last major release, supports asynchronous views, it does not support asynchronous ORM and some other parts of framework usage. The project implies a lot of communication with external services; hence asynchronous execution is crucial for service speed. To add more, the asynchronous mode of Django is way less efficient than FastAPI *Python Async (ASGI) Web Frameworks Benchmark.* 2022.

TABLE 3.2: Django summary

	Django
Asynchronous execution support	Yes
Bundled ORM	No
Performance	Same as Flask in synchronous mode, slower then FastAPI
Wide usage	Yes
Background tasks support	Yes
Automatic request validation	No
Automatic documentation generation	No

FastAPI

FastAPI(*FastAPI reference* 2018) is relatively new, developed in 2018. It is high - a performance web - framework based on Python type hints. One of its best features is being short, hence less space for developer errors and producing bugs. Its second prominent feature is the intense use of Python type, hinting at automating manual development work.

Advantages of the FastAPI:

- Support of ASGI(*ASGI reference* 2018) standard.

  ASGI is a spiritual successor of WSGI and is intended to provide a standard interface between async-capable Python web apps. ASGI also stays compatible with WSGI.

- Compatibility with WSGI(*WSGI reference* 2010) views.

  FastAPI has the ability to mount Flask, Django, or other WSGI applications through middleware. This is great for project expansibility and gives the ability to develop apps in a modular way.

- Extensible validation models through type hints.

  Defining request parameters via classes gives the ability to reduce code duplication while defining models significantly.

- Automatic OpenAPI documentation generation.

  FastAPI uses type hints to generate OpenAPI compatible documentation. This helped develop frontend and backend parts in parallel easily because all logic updates are reflected in documentation immediately.

- Automatic error responses for invalid requests

  FastAPI validates requests and gives a meaningful response if the request violates the declared schema.

- High performance.

  FastAPI provide provides performance on par with Node.js and express.js *FastAPI performance benchmarks and comparison.* 2014.

- Ability to execute not computation-intensive background tasks without a task queue.

- Own Dependency Injection system.

- Adopted extremely wide

  FastAPI is a new framework, but over the period of its' existence its' popularity has grown extremely fast; hence there is a lot of experience available.

- WebSockets client.

  FastAPI provides WebSockets client that can be used to implement an interface for communicating with mobile app, that can implemented as part of Amphora development.

Disadvantages of the FastAPI:

- Absence of class based views.

- Less used now compared to other described frameworks.

  FastAPI is a relatively new framework compared to Flask and Django; hence, less experience and third-party tools are available on the web. Even though there is still a difference, this is not a big problem, as FastAPI is adopted quickly by engineers and even by big corporations.

Conclusion

FastAPI is an excellent framework for developing high-performance APIs of small and medium sizes. It also fits the needs of the project because the most effective way to implement the communication with external services is to do this asynchronously. As our service will do such communication a lot, this is a good argument for choosing this framework.

TABLE 3.3: FastAPI summary

	FastAPI
Asynchronous execution support	Yes
Bundled ORM	No
Performance	Faster than Django in either of the execution modes and Flask
Wide usage	Yes
Background tasks support	Yes
Automatic request validation	Yes
Automatic documentation generation	Yes

Frameworks Conclusion

The first decision made is that Django is the wrong choice for the project as there will be a small amount of logic that can be plugged in directly from the Django library; hence we want to avoid having a lot of unused code as part of dependency. Also, the level of configurability that Django provides does not suit the needs of rapid project development and reacting to interface updates on the listening platform side.

The choice between Flask and FastAPI was obvious: the platform will communicate with many external services; hence asynchronous execution will provide a significant performance boost. In terms of configurability, these two frameworks are more or less the same. The significant advantages of FastAPI are background tasks and web sockets. These features will be handy in implementing podcast episode status checks and effective communication with a mobile application. Also, FastAPI is more efficient than Flask.

So, FastAPI was chosen as the framework for the project.

3.4.5 ORM choice

As chosen API has no bundled ORM, we have a variety of options to choose from. The go-to solutions usually, in this case are SQLAlchemy or Peewee. SQLAlchemy provide high level features as eager loading, declarative model syntax, expression building via operator overloads and other advanced querying features. Another big advantage of SQLAlchemy is alembic: a database migration tool that makes working with migrations very comfortable. Peewee's two main features are being small and efficient: it's often used in AWS lambda functions or in other situations when being small is crucial. Both ORMs provide support for asynchronous execution.

But let's consider features of the web framework we decided to use – requests validation and documentation generation with Pydantic models and Python type hinting. Authors of the FastAPI framework developed SQLModel: ORM, which is built on top of SQLAlchemy. This ORM's main features are being intuitive, short, easy to use, and highly compatible with FastAPI, Pydantic, and SQLAlchemy. On a technical level, it means that SQLModel helps to minimize code duplication in the logic that is connected with validating requests and interacting with the database or eliminate it at all. Considering all the benefits that SQLAlchemy provides and SQLModel being a thin layer on top of SQLAlchemy and Pydantic, SQLModel was chosen as ORM for the Amphora project.

Managing migrations

Database migration is a set of changes to modify the structure of objects within the relational database. There are multiple ways to do database migrations:

- Writing them in plain SQL.
- Using a database migration tool that will automate the migration files generation process.
- Checking that the database schema is up to date on the server start and updating it if needed.

This will not just make applying database schema changes fast and easy; this will also allow saving database migration to have the ability to recreate database schema on other instances with ease. If we talk about the disadvantages of this decision – a bit complex setup and additional table in the database. Alembic uses this table to save id of last applied migration. Hence, Amphora uses alembic to manage database migrations.

3.5 Implementation details

Amphora's server provides CRUDL functionality for User, Episode, and Show entities, generating and serving RSS feed and more. Let us go through some implementation details and technical decisions.

3.5.1 User system

Users table has two boolean fields: is verified and is superuser. The first field is needed to implement protection against using service by bot users. Only verified users have the ability to create a show and episodes via Amphora. Superuser status is meant for administrator users who will help to run Amphora.

3.5.2 Users verification, authorization, and password renewal

Verification is done by sending JWT token to the email of the user. If users provide the token they received to the system correctly, their account is verified, and podcast creation functionality becomes accessible.

Password renewal is done with the same workflow and usage of JWT token, but in this case Amphora also requests for new password that satisfies security requirements.

When it comes to user authorization, on login user receives a fresh session token that has TTL and will become inactive when this time expires. These tokens and their creation time is stored in the database. Storing and retrieving access tokens in the database has obvious downsides, but they will not harm the stability of the system while the amount of users is not very big.

3.5.3 Deleting rows from database

To give users the ability to restore resources they want to delete, rows are not actually deleted on rows delete request, just marked as deleted. In order not to clutter the repository with old data, periodic jobs will delete old rows in a database that are marked removed after the grace period.

3.5.4 Database communication

All database communication logic is implemented in one place and done asynchronously. Due to reusable database communication logic, the grace period for user resources is implemented everywhere by writing it just once.

3.5.5 File storage

We use S3-compatible storage for storing images and audio files. In order for the system to stay responsive and effective, communication with the storage is done asynchronously via aioboto3.

3.5.6 Pydantic and SQLModel models

In the implementation of Amphora's logic, we use Pydantic models to define and validate the request schema for the endpoints. Pydantic models syntax is very close to Python std dataclasses. The beauty of this approach is that we can use inheritance to define schemas for various needs. FastAPI will not only validate requests against provided schema and give a meaningful response but also will generate OpenAPI documentation for the endpoints with the examples of requests. This documentation is beneficial in developing the front-end part of the system. The only requirement is to use type hinting to be consistent and make the development of the system more convenient; all code of the project uses type hinting.

Also, the ORM we chose, SQLModel is highly compatible with Pydantic models; hence we can use inheritance from the Pydantic models we defined to define database tables needed for the Amphora logic.

So, by choosing and using Pydantic and SQLModel, we managed to accomplish the following :

- Request validation for all endpoints.
- Automatic OpenAPI documentation generation that is always up to date.
- Significantly reduce code duplication.
- Type hinting for all codebase of the project is imposed, which is a great technique in Python development.

3.6 Infrastructure and System Architecture

3.6.1 System Architecture

Amphora podcast hosting consists of FastAPI RESTful API as the backend and Angular app as a frontend. Also, a cron job is set up that removes old content marked as deleted.

REST was chosen as architecture for the backend because Amphora's primary function is manipulating a Podcast and connected entities; hence our system is built around entities and their states.

Amphora is fast and highly responsive because frontend and backend parts were built with speed and responsiveness as priorities in mind and modern, effective technologies as tooling. The backend part of the project is implemented via FastAPI, which by itself is a very fast Python web framework, to add more all input and output: uploading files, sending emails, and communicating with the database are done asynchronously. Regarding the frontend part – it is done in a way that all updates on the UI are performed dynamically by JavaScript; hence no reloads needed to update the data or change UI elements. So, the combination of a single-page application frontend with a heavily asynchronous backend is a fast and responsive combination.

3.6.2 Infrastructure

The system has a custom domain set up, which is used for all emails to users and as DNS for accessing a website. Also, the backend part of Amphora is built with probable migration to Cloud in mind. The database is organized in a way to be portable. FastAPI server is also built as a bunch of asynchronous procedures; hence its logic can be easily migrated to AWS lambda functions and other supporting services or analogs from other hostings.

Chapter 4

Conclusions

4.1 Brief conclusions

In this work, we put an assumption on problems in the sphere of podcasting. Interviews with podcast authors were held, and research of video, audio, and text materials on podcasting available on the web was done. We also did research on existing solutions on the market, and it was found that there are a lot of problems with UX, lack of automatizing, and integrations with some services. A list of features needed for the MVP version of the project was defined. We collaborated with podcasters one more time to define a features list that will improve the UX of the system and put priorities on these features.

After that, extensible and potable architecture for the Amphora system was designed. After the implementation of the MVP version, both parts of the platform were hosted to be publicly available on the web. MVP version of the project support integration through RSS feed with the Apple Podcasts, Spotify and Pocket Casts. We bought a custom domain name and set it up to be used with the mailing system of the Amphora. We held a manual testing session of the MVP with the associates, and the feedback from them on the UX was taken and evaluated. After this, we implemented part of the additional features defined, deployed the changes to the system iteratively, and held manual testing on every stage. We set up DNS for our app and gave the podcasters a stable version of Amphora for testing. Our product received positive feedback and some advice on the project's further development.

We managed to find a problem in the real world, describe it as the pain points of the user, transform it into software and plan features for further development of the product, organize more or less effective testing of the system, host it, buy and set up a custom domain.

4.2 Further perspective

We managed to design a product and roadmap for its development based on the users' pain points. We finished a complete development cycle on the basic functionality and part of the roadmap. Also, the service was hosted and fully set up for production use. After the testing with podcast creators, we received positive feedback and ideas for further development.

From our research and testing software that was created in the scope of this work, the idea of the project is relevant, and the product needs only a pricing model to be launched to the market. From the interviews and other research we conducted, it is evident that podcast creators are willing to pay for the features we provide.

So, the product, which is a result of this work:

- Solves a problem it was designed for.

- Potentially profitable.
- Has a feature map ready for project development.
- Is publicly available.
- Production-use ready in free mode.

In technical context, the project is designed and implemented with portability and rapid development in mind. To became feature-rich service and best podcast hosting service project needs more time and work on further features.

Other aspect of further development is communication work with ads providers to ensure that every podcast author has ability to get paid for their work regardless of location, topic or audience size.

The general vision of further project development is reacting to podcasters issues and appeals as fast and qualitatively as possible. This should help to create a podcast hosting that is really comfortable for podcast creators.

This project has the potential to become a profitable business, and with the growth of the audience and service development will help podcast creators more and more over time.

Bibliography

Adeshina, Abdulazeez Abdulazeez (2022). *Building Web APIs with FastAPI and Python: A fast-paced guide to building high-performance, robust web APIs with very little boilerplate code*. Packt Publishing.

ASGI reference (2018). URL: `https://asgi.readthedocs.io`.

FastAPI performance benchmarks and comparison. (2014). URL: `https://www.travisluong.com/fastapi-vs-express-js-vs-flask-vs-nest-js-benchmark/`.

FastAPI reference (2018). URL: `https://fastapi.tiangolo.com`.

Python Async (ASGI) Web Frameworks Benchmark. (2022). URL: `https://klen.github.io/py-frameworks-bench`.

Skvortsov, Victor (2021). *How Python async-await works behind the scenes*. URL: `https://tenthousandmeters.com/blog/python-behind-the-scenes-12-how-asyncawait-works-in-python/`.

Why ORM is an offensive anti-pattern. (2014). URL: `https://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html`.

WSGI reference (2010). URL: `https://wsgi.readthedocs.io`.