UNIVERSITY
INSTITUTE
OF LISBON

Self-hosted Podcast Library Management System

António João Casado Pereira

Master in Computer Science and Business Management

Supervisor:
PhD. Sancho Moura Oliveira, Associate Professor
Iscte - Instituto Universitário de Lisboa

December, 2025

[This page has been intentionally left blank]

TECHNOLOGY
AND ARCHITECTURE

Self-hosted Podcast Library Management System

António João Casado Pereira

Master in Computer Science and Business Management

Supervisor:
PhD. Sancho Moura Oliveira, Associate Professor
Iscte - Instituto Universitário de Lisboa

December, 2025

[This page has been intentionally left blank]

Self-hosted Podcast Library Management System

[This page has been intentionally left blank]

The power of Open Source is the power of the people

- Philippe Kahn

[This page has been intentionally left blank]

Acknowledgment

I would like to begin by expressing my gratitude to my family for all their support and encouragement, not only in helping me complete another stage of my academic journey but also throughout my life, through their values and education.

A very special thank you goes to my girlfriend for always being by my side, for her dedication and support at all times, helping me to stay focused and balance my personal, academic, and professional life in order to make this project possible.

I also thank Professor Sancho Moura Oliveira for embracing this project and for his invaluable guidance throughout its development, as well as his insightful and relevant contributions. I would also like to thank the Instituto de Telecomunicações for the excellent conditions provided.

[This page has been intentionally left blank]

Resumo

Os *podcasts* tornaram-se um meio amplamente adotado para a educação, entretenimento e partilha de informação, oferecendo um formato flexível e acessível em diversas plataformas. Contudo, a maioria dos clientes de *podcast* modernos depende de modelos baseados em subscrição ou de ecossistemas proprietários que limitam o controlo do utilizador, levantam preocupações de privacidade e restringem o acesso a *feeds* RSS privados. Para solucionar estes problemas, este projeto apresenta o design e a implementação de um sistema de gestão de biblioteca de *podcasts self-hosted* de código aberto, para proporcionar aos utilizadores um controlo total sobre os seus dados e experiência auditiva.

A solução proposta tem uma arquitetura modular composta por um *backend* baseado em *Go*, um *SPA frontend* em *Vue.js* e um sistema de *plugins* extensível para o *backend* e o *frontend*. As principais funcionalidades incluem suporte a múltiplos utilizadores, subscrição e sincronização de *podcasts*, acompanhamento do progresso da reprodução e gestão automatizada de episódios. O *backend* utiliza interfaces *gRPC* e *REST* para uma comunicação eficiente, enquanto o *frontend* garante uma interface de utilizador responsiva e otimizada para dispositivos móveis, servida diretamente pelo *backend*.

Através da integração de tecnologias de código aberto como o *Docker*, *PostgreSQL* e *Extism* para a execução de *plugins* baseados em *Wasm*, o sistema promove a escalabilidade, a facilidade de manutenção e a transparência. Este trabalho não só entrega uma plataforma funcional de gestão de *podcasts*, como também demonstra o valor da colaboração em código aberto para o desenvolvimento de aplicações *web* extensíveis e que respeitam a privacidade.

Palavras-chave: *"Self-hosted", "Podcast", Código aberto*

[This page has been intentionally left blank]

Abstract

Podcasts have become a widely adopted medium for education, entertainment, and information sharing, offering a flexible and accessible format across multiple platforms. However, most modern podcast clients rely on subscription-based models or proprietary ecosystems that limit user control, raise privacy concerns, and restrict access to private *RSS* feeds. To address these issues, this project presents the design and implementation of an open-source, self-hosted podcast library management system aimed at providing users with full ownership of their data and listening experience.

The proposed solution is built around a modular architecture comprising a *Go*-based backend, a *Vue.js*-powered *SPA* frontend, and an extensible plugin system supporting both backend and frontend extensions. Key features include multi-user support, podcast subscription and synchronization, playback progress tracking, and automated episode management. The backend leverages *gRPC* and *REST* interfaces for efficient communication, while the frontend ensures a responsive, mobile-friendly user interface served directly by the backend.

Through the integration of open-source technologies such as *Docker*, *PostgreSQL*, and *Extism* for *Wasm*-based plugin execution, the system promotes scalability, maintainability, and transparency. This work not only delivers a functional podcast management platform but also demonstrates the value of open-source collaboration in fostering privacy-respecting and extensible web applications.

Keywords: *Self-hosted, Podcast, Open-source*

[This page has been intentionally left blank]

List of Tables

[This page has been intentionally left blank]

List of Figures

[This page has been intentionally left blank]

Acronyms

ACL: Access Control List. 26
AGPL: GNU Affero General Public License. 41

CLI: Command Line Interface. 18
CORS: Cross-Origin Resource Sharing. 25–27
CPD: Continuing Professional Development. 1

ERD: Entity Relationship Diagram. 18

FSF: Free Software Foundation. 2, 40, 41

GPL: GNU General Public License. 40

HMAC: Hash-based Message Authentication Code. 25
HTTP: Hypertext Transfer Protocol. 21, 24, 26, 34

IDL: Interface Definition Language. 20

JWT: JSON Web Token. 24–26, 34

LGPL: GNU Lesser General Public License. 40

MAC: Message Authentication Code. 24
MPL: Mozilla Public License. 40

OSI: Open Source Initiative. 2, 40
OSL: Open Software License. 41

PoC: Proof of Concept. 30

RBAC: Role-Based Access Control. 23
RDBMS: Relational Database Management System. 17

SHA-256: Secure Hash Algorithm 256-bit. 25
SOP: Same-Origin Policy. 26
SPA: Single-Page Application. v, vii, 16, 19
SPDX: Software Package Data Exchange. 40

TLV: Tag-Length-Value. 21

URI: Uniform Resource Identifier. 24
USA: United States of America. 1

VM: Virtual Machine. 36

XML: Extensible Markup Language. 6, 30

Glossary

ABI: Application Binary Interface (ABI) defines the low-level interface between two binary program modules, typically between an application and the operating system or between different components of a program. An ABI specifies details such as data type sizes, memory alignment, calling conventions, and system call mechanisms, ensuring that compiled code can interoperate across different modules and environments. 30

API: Application Programming Interface (API) is a defined set of rules and protocols that allows software applications to communicate with each other. An API specifies how requests and responses should be structured, enabling interoperability between different components, systems, or services. In the context of media management systems, APIs are commonly used to expose backend functionality to frontends or external applications, often through REST or gRPC interfaces. 3, 10, 12, 21, 35, 36, 45

Embed: A directive that allows the inclusion of a file's content in the source code of a Go program. 18

Extism: An open source framework to build cross-language plug-in systems powered by WebAssembly. v, 16

Go: Also known as Golang, it is a statically-typed, compiled programming language designed by Google. It is syntactically similar to C, but provides memory safety, garbage collection, structural typing, and CSP-style concurrency. v, vii, 15, 29, 33, 34, 37

gRPC: gRPC Remote Procedure Calls (gRPC) is a high-performance, open-source universal RPC framework that can run in any environment. It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and it provides features such as authentication, load balancing, and bidirectional streaming. v, vii, 19, 20, 22, 29

JSON: JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. 21, 24, 25

kernel: The core component of an operating system responsible for managing system resources, such as the CPU, memory, and input/output devices. It provides a bridge

between hardware and software by offering low-level services like process scheduling, memory allocation, and device communication, ensuring that applications and system processes run efficiently and securely. 36

MIME: Multipurpose Internet Mail Extensions, a standard that defines the format of files transmitted over the internet. It specifies the type and nature of a file (e.g., `audio/mpeg`, `image/png`), allowing applications to correctly handle and interpret the content. 6

MySQL: An open-source relational database management system based on SQL (Structured Query Language). 18

NAS: Stands for *Network Attached Storage*. A specialized file storage device that provides centralized data access to multiple clients over a network. NAS devices are typically designed for ease of use, offering features such as redundancy, fault tolerance, and remote access, making them suitable for both home environments and enterprise applications. They are widely used to support self-hosting by providing reliable, always-on storage infrastructure with relatively low maintenance requirements. 9

OPML: Stands for *Outline Processor Markup Language*, an XML-based format commonly used to exchange lists of structured data. In the context of podcast applications, OPML is widely adopted for importing and exporting subscription lists, allowing users to migrate or share their podcast feeds across different clients. 30

PDK: A collection of language-specific libraries and tools that simplify the development of plugins for a given framework or platform. A PDK typically provides runtime bindings, memory management utilities, and abstractions for interacting with the host environment, allowing developers to export functions and integrate custom logic without dealing directly with low-level details. 29, 30

PostgreSQL: A powerful, open source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance. 16–18

Protobuf: Protocol Buffers, often abbreviated as *protobuf*, are a language-neutral, platform-neutral extensible mechanism developed by Google for serializing structured data, commonly used for communication protocols, data storage, and RPC systems. 19–21

protoc: A tool used to compile `.proto` files (Protocol Buffers definition files) into code in a variety of programming languages. The compiled code provides a way to serialize structured data, enabling communication between applications across different platforms. The `protoc` compiler generates source files that implement data serialization and deserialization routines based on the schema described in `.proto` files. 21

REST API: Also called a RESTful API, it is an application programming interface (API) that conforms to the design principles of the representational state transfer (REST) architectural style. REST APIs provide a flexible, lightweight way to integrate applications. v, 16, 19, 22

RPC: Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located on another computer in a network. It abstracts the details of the network communication, allowing developers to call functions as if they were local. 19, 29

RSS: Rich Site Summary or Really Simple Syndication (RSS) is a type of web feed that allows users and applications to access updates to websites in a standardized, computer-readable format. v, vii, 2, 3, 5–7, 9, 16

SDK: A collection of software tools, libraries, documentation, and example code that facilitates the development of applications for a specific platform, framework, or service. An SDK commonly provides application programming interfaces (APIs), language bindings, utility libraries, and developer tools (e.g., compilers and debuggers) that abstract platform details, promote consistency, and accelerate development while reducing integration errors. 29, 30

Socket.IO: A library that enables real-time, bidirectional and event-based communication between the browser and the server. 16

SQLite: A lightweight, self-contained, serverless relational database engine that stores data in a single file. It is widely used for embedded systems, mobile applications, and small to medium-scale web projects due to its simplicity, reliability, and minimal configuration requirements. 18

URL: Uniform Resource Locator, the standard address used to identify and access resources on the internet, such as web pages, files, or media streams. 6, 35

Vue.js: An open-source frontend JavaScript framework for building user interfaces and single-page applications. v, vii, 11, 16, 31

Wasm: Often shortened to Wasm, it is a low-level, binary instruction format designed for efficient execution and compact representation. WebAssembly enables code written in multiple programming languages to run at near-native speed in web browsers and other environments. v, vii, 13, 16, 29, 30

YAML: Stands for "YAML Ain't Markup Language" and is a human-readable data serialization standard that can be used in conjunction with all programming languages and is often used to write configuration files. 16, 28

[This page has been intentionally left blank]

Contents

CHAPTER 1

Introduction

Podcasts are becoming mainstream, with more listeners than ever. As of 2023, considering the United States of America (USA) population above 12 years old, when queried, about 64% (estimated 183 million people) have listened to a podcast until then, 42% (estimated 120 million people) in the last month and 31% (estimated 89 million people) in the last week (Edison Research, 2023b), "reaching the highest numbers ever" by Edison Research, 2023a.

Listeners are spending more time than before, listening about 9 hours per week, averaging 9 podcasts' episodes per week, with 19% listening 11 or more episodes. Most frequently consumed using portable listening devices such as smartphones and tablets (Edison Research, 2023b).

Podcasts have grown rapidly in both popularity and diversity, reaching an affluent, employed, and educated audience. Their genres span from comedy, society & culture, and news, asthe most popular categories (Edison Research, 2023b), to business, history, science, technology, and education. Research in the medical field has highlighted their potential for enhancing *Continuing Professional Development (CPD)* (Morris et al., 2015; Newman et al., 2021), while studies with undergraduate students suggest that podcasts can improve the learning experience by serving as revision tools and are often preferred over traditional textbooks (Evans, 2008; Fernandez et al., 2009). Combined with the flexibility of allowing learning anytime, anywhere, and in multiple formats, podcasts demonstrate significant potential as an innovative medium for education and professional development, offering unique and powerful opportunities for information sharing. However, realizing this potential depends not only on podcast content but also on the availability of accessible, customizable, and privacy-conscious management systems, which motivates the exploration of self-hosted and open-source alternatives.

1.1. Motivation

Currently, most multiplatform podcast clients require an active subscription to synchronize episodes and playback progress. For example, services like Pocket Casts[1] offer cross-device synchronization only to paid users, while ad-free listening typically requires an additional one-time payment or subscription fee on platforms like Spotify[2]. Many clients also raise privacy concerns due to telemetry collection, as observed in apps that track listening habits for recommendations or analytics. Furthermore, some platforms provide

[1] `https://pocketcasts.com/`
[2] `https://open.spotify.com`

access to a limited podcast library and do not support private *RSS* feeds, preventing users from subscribing to self-hosted or niche content that is not publicly listed.

Following Plex[3] discontinuing podcast support in April 2022 (Plex inc., 2022), many Plex users and self-hosters lost a viable option to overcome some of these limitations. This situation underscores the need for alternative solutions that provide full control over podcast subscriptions, ensure privacy, support private *RSS* feeds, and allow cross-device synchronization without mandatory recurring payments.

Beyond these limitations, there are additional motivations for adopting a self-hosted library management system. Self-hosting enables complete ownership and control of media content, ensuring that users are not dependent on third-party services or subject to service discontinuations. It allows customization of features, user interfaces, and workflows to suit individual preferences or organizational policies. Finally, self-hosting enhances privacy and security by keeping user data, listening habits, and subscription information under the direct control of the host, reducing exposure to advertising or tracking.

Open-source software offers further advantages in this context. It ensures transparency, allowing users to audit the code and verify that no unwanted data collection or restrictions are imposed. Community-driven development encourages rapid iteration, bug fixes, and feature enhancements, often outpacing proprietary alternatives in flexibility. Moreover, open-source solutions can be extended or customized to meet specific requirements, making them adaptable to diverse environments and sustainable over the long term. These principles align with the definitions established by the *Open Source Initiative (OSI)*[4] and the *Free Software Foundation (FSF)*[5], which emphasize user freedoms, collaboration, and independence from vendor lock-in.

1.2. Objectives

Building on the limitations identified in existing podcast clients and leveraging the power of the open-source community, this project aims to develop a free, self-hosted podcast library management system that addresses the most pressing user needs. In contrast to commercial alternatives, the proposed system will be entirely ad-free, ensuring an uninterrupted listening experience. It will support multiple user accounts, a key requirement for shared environments such as families or small communities. The system will enable podcast subscriptions and automatic episode downloads, reducing manual effort and ensuring timely access to new content. In-browser playback through a web interface will make the solution widely accessible without the need for additional software. Synchronization of playback progress across devices will provide a seamless user experience, particularly for users who alternate between desktop and mobile devices. Support for private *RSS* feeds will guarantee unrestricted access to both public and privately hosted podcasts, a feature often absent from commercial platforms. Finally, the system will include an extensible

[3] `https://www.plex.tv/`
[4] `https://opensource.org/osd`
[5] `https://www.gnu.org/philosophy/free-sw.en.html`

architecture with plugins for both backend and frontend components, allowing users to adapt and expand the platform to evolving requirements. Collectively, these objectives seek to deliver a platform that maximizes user control, safeguards privacy, and provides long-term flexibility beyond the constraints of proprietary solutions.

1.3. Structure

The present document is organized into four main chapters, each addressing a different stage of the research and development process, from conceptualization to implementation and reflection on results.

Chapter 1 – Introduction outlines the motivation behind the project and establishes its objectives. It contextualizes the problem that the solution aims to address, emphasizing the need for a self-hosted, open-source podcast management platform and defining the goals that guided its development.

Chapter 2 – Background presents the theoretical and technical foundations supporting the project. It begins by exploring the history and technological principles of podcasts and their reliance on *RSS* feeds. The chapter then examines the self-hosting paradigm, existing podcast management solutions, and the concept of plugin-based architectures, providing the context necessary to understand the rationale behind the chosen design and technologies.

Chapter 3 – Development describes the implementation of the proposed solution, outlining the main technical decisions and methodologies adopted throughout the process. It details the system's architecture, database design, and *API* structure, including authentication mechanisms and real-time communication. The chapter also discusses the plugin framework for both backend and frontend components, the scheduler, and the frontend implementation. It concludes with the integration of *Docker* for containerization, as well as licensing aspects and contributions to the open-source ecosystem.

Chapter 4 – Conclusions summarizes the outcomes of the project, evaluating how the defined objectives were achieved and reflecting on the limitations identified during development. It also proposes potential directions for future work and improvements to further enhance the functionality and adaptability of the solution.

[This page has been intentionally left blank]

CHAPTER 2

Background

This chapter provides the foundational context necessary to understand the development and challenges of self-hosted podcast library management systems. It begins by tracing the origins and evolution of podcasting, highlighting the technical innovations and cultural shifts that enabled its widespread adoption. The discussion then explores the role of *RSS* feeds as the backbone of podcast distribution, followed by an examination of self-hosting practices, their benefits, and associated technical hurdles. A review of existing open-source solutions is presented, comparing their features and implementation technologies. Finally, the chapter delves into plugin architectures, illustrating how extensibility and modularity are achieved in modern software systems. Together, these sections establish the groundwork for the subsequent analysis and design of a self-hosted podcast management platform.

2.1. Podcast

The origins of podcasting can be traced back to the early 2000s, when experimentation with *RSS* feeds and digital audio distribution laid the foundation for what would later become a global phenomenon. In 2000, software developer Dave Winer introduced *RSS* 0.92, adding support for `enclosures` that enabled the distribution of digital audio files (Winer, 2000b). This capability was implemented at the request of former MTV VJ Adam Curry, who envisioned a way to automatically deliver audio content to portable media players. Winer's technical contribution provided the infrastructure needed for audio content to be syndicated and consumed automatically, setting the stage for the emergence of podcasting (Bottomley, 2015; Winer, 2000a).

The concept of podcasting evolved through pioneering work by Christopher Lydon, a former radio journalist. In 2003, he began publishing audio conversations with internet visionaries, political figures, and futurists on a Harvard University weblog, creating one of the first examples of an online audio series distributed via the web. Later that same year, he founded *Radio Open Source*, an internet radio project that further demonstrated the potential of audio conversations distributed online (Kuchta, 2001; Frizzell, 2016).

Around the same period, former MTV video jocker Adam Curry introduced a script at BloggerCon 2003 that enabled the automatic downloading of audio enclosures and their transfer to Apple's *iPod*. This breakthrough bridged the gap between web syndication and portable listening, paving the way for podcasting as we know it today (Kuchta, 2001). In 2004, Curry partnered with Winer to release *iPodder* an *RSS-to-iPod* application and launched the influential podcast *Daily Source Code*, which not only showcased the new

medium but also inspired others to experiment with podcast production (Bottomley, 2015).

The term "podcasting" itself entered the lexicon in 2004, when journalist Ben Hammersley published an article in *The Guardian* titled *Audible Revolution* (Hammersley, 2004). In it, he coined the term "podcasting" as a blend of "iPod" and "broadcasting", while noting alternative labels such as "audioblogging" and "guerrilla media" that ultimately did not gain traction. In the same year, Dave Mansuet and Dave Chekan founded *Libsyn.com* (short for Liberated Syndication)[6], one of the first podcast hosting services, offering tools to facilitate *RSS* feed creation and distribution.

In 2005, as a sign of public recognition, the New Oxford American Dictionary declared "podcast" the word of the year (Oxford, 2024; Bowers, 2005), reflecting its rapid adoption and cultural impact. That same year, Steve Jobs announced that *iTunes* would officially support podcasts, launching with 3,000 free shows available in the *iTunes Store*. This move by Apple significantly accelerated mainstream adoption by lowering the barrier to podcast discovery and consumption (Frizzell, 2016; Bottomley, 2015).

These formative years established the technical, cultural, and commercial foundations of podcasting. From experimental *RSS*-based audio syndication to widespread availability through platforms like *iTunes*, podcasting evolved from a niche innovation into a recognized and rapidly growing medium for information sharing, storytelling, and entertainment.

2.1.1. RSS Feed

Podcasts fundamentally rely on *RSS* feeds as their primary distribution mechanism. An *RSS* feed is an *XML*-based file format originally designed to syndicate regularly updated content, such as blog posts or news articles, across different platforms. With the addition of the `enclosure` element (Winer, 2000b), *RSS* feeds were extended to support media distribution, enabling the automatic delivery of audio files that became the foundation of podcasting.

Technically, an *RSS* feed is a structured *XML* document that includes metadata about the channel, such as its title, description, author, and publication date. Along with a list of items representing individual episodes. Each item can include attributes such as the episode's title, publication date, summary, and most importantly, an `enclosure` tag that specifies the *URL* of the media file, its *MIME* type, and file size. This standardized structure allows podcast clients to parse the feed, display available episodes to the user, and download or stream audio content directly.

For interoperability across podcast clients and platforms, podcasts must adhere to the *RSS* 2.0 specification authored by Winer, 2003 and currently maintained by the RSS Advisory Board, 2009, which has become the de facto standard for podcast distribution. The *RSS* 2.0 format provides the structural requirements for defining a channel, metadata, and individual items (episodes), while supporting the `enclosure` element that links audio

[6] `https://libsyn.com/`

files. Being an open standard means that not all feeds have consistent practises, which in some cases might brake podcast clients. In an attempt to solve these divergences the *Podcast Standards Project*[7] compiled a set of requirements and recommendations[8] on how a podcast *RSS* feed document should be structured. A partial *RSS* feed of the *Self-hosted* podcast listing its first episode can be seem in Listing 1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rss version="2.0" encoding="UTF-8" xmlns:atom="http://www.w3.org/2005/Atom/"
   ↪  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   ↪  xmlns:content="http://purl.org/rss/1.0/modules/content/"
   ↪  xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"
   ↪  xmlns:podcast="https://podcastindex.org/namespace/1.0">
3    <channel>
4      <title>Self-Hosted</title>
5      <link>https://selfhosted.show</link>
6      <pubDate>Fri, 30 May 2025 03:45:05 -0700</pubDate>
7      <description>Discover new software and hardware [...]</description>
8      <language>en-us</language>
9      <copyright>© 2025 Jupiter Broadcasting</copyright>
10     <itunes:type>episodic</itunes:type>
11     <itunes:subtitle>A chat show between Chris and Alex [...]</itunes:subtitle>
12     <itunes:author>Jupiter Broadcasting</itunes:author>
13     <itunes:summary>Discover new software and hardware [...]</itunes:summary>
14     <itunes:image href="https://media24.fireside.fm/file/fireside-images-2024/podcast
       ↪  s/images/7/7296e34a-2697-479a-adfb-ad32329dd0b0/cover.jpg?v=2"/>
15     <itunes:explicit>no</itunes:explicit>
16     <itunes:keywords>Self-Hosting, DIY, Linux, open source, [...]</itunes:keywords>
17     <itunes:owner>
18       <itunes:name>Jupiter Broadcasting</itunes:name>
19       <itunes:email>chris@jupiterbroadcasting.com</itunes:email>
20     </itunes:owner>
21     <podcast:locked email="chris@jupiterbroadcasting.com">yes</podcast:locked>
22     <itunes:category text="Technology"/>
23     <generator>Fireside (https://fireside.fm)</generator>
24     <item>
25       <title>Self-Hosted Coming Soon</title>
26       <link>https://selfhosted.show/0</link>
27       <guid isPermaLink="false">d1f1ab78-f5b4-441a-87f8-64fff2b78d1c</guid>
28       <pubDate>Tue, 27 Aug 2019 04:00:00 -0700</pubDate>
29       <author>Jupiter Broadcasting</author>
30       <enclosure url="https://aphid.fireside.fm/d/1437767933/7296e34a-2697-479a-adfb-
         ↪  ad32329dd0b0/d1f1ab78-f5b4-441a-87f8-64fff2b78d1c.mp3" length="2239054"
         ↪  type="audio/mp3"/>
31       <itunes:episodeType>full</itunes:episodeType>
32       <itunes:author>Jupiter Broadcasting</itunes:author>
```

[7] https://podstandards.org/
[8] https://github.com/Podcast-Standards-Project/PSP-1-Podcast-RSS-Specification

```
33        <itunes:subtitle>A new show that is your gateway to [...]</itunes:subtitle>
34        <itunes:duration>2:11</itunes:duration>
35        <itunes:explicit>no</itunes:explicit>
36        <itunes:image href="https://media24.fireside.fm/file/fireside-images-2024/podca⌋
          ↪  sts/images/7/7296e34a-2697-479a-adfb-ad32329dd0b0/cover.jpg?v=2"/>
37        <description>A new show that is your gateway to [...]</description>
38        <itunes:keywords>Self Hosted Podcast[...]</itunes:keywords>
39        <content:encoded>
40          <![CDATA[<p>A new show that is your gateway to [...]</p>]]>
41        </content:encoded>
42        <itunes:summary>
43          <![CDATA[<p>A new show that is your gateway to [...]</p>]]>
44        </itunes:summary>
45        <podcast:person email="" href="https://alex.ktz.me" role="host">Alex
          ↪  Kretzschmar</podcast:person>
46        <podcast:person email="" href="https://www.jupiterbroadcasting.com/"
          ↪  role="host">Chris Fisher</podcast:person>
47      </item>
48    </channel>
49  </rss>
```

LISTING 1. Partial *Self-hosted*'s *RSS* podcast feed

2.2. Self-hosting

Self-hosting refers to the practice of running and maintaining software services on infrastructure directly controlled by the user or organization, rather than relying on third-party providers. This approach typically requires access to reliable hardware, sufficient network bandwidth, a secure operating environment, and technical expertise in system administration and maintenance. While self-hosting offers advantages such as increased control, data ownership, and customization, it also presents significant challenges. These include ensuring system security and availability, managing regular updates and patches, handling scalability as demand grows, and addressing the risk of downtime due to hardware failures or misconfigurations. Consequently, self-hosting was historically limited to technically proficient users and organizations with the resources necessary to manage these complexities (Gröber et al., 2023).

Over time, however, several technological developments have lowered the barriers to adoption of self-hosted open-source solutions. Containerization technologies such as *Docker* simplified deployment by encapsulating applications and their dependencies, ensuring consistent execution across heterogeneous environments and reducing the need for manual configuration (Julia Wilson et al., 2025). Collaborative platforms like *GitHub*[9] accelerated the growth of open-source ecosystems by providing centralized repositories, issue tracking, and automated pipelines, making it easier for communities to develop, distribute, and maintain self-hosted projects, like the ones listed at *Awesome-Selfhosted*

[9] https://github.com/

repository[10]. At the hardware level, the increasing affordability and availability of *Network Attached Storage (NAS)* devices, along with low-cost single-board computers such as the *Raspberry Pi* and similar compact systems, have enabled individuals and small organizations to host services and manage data on reliable, low-maintenance infrastructures deployed at home or in small offices.

These advances transformed self-hosting from a niche practice reserved for experts into a more accessible and mainstream option for a wide range of users. As a result, a number of prominent open-source projects have risen to widespread popularity:

- *Plex Media Server*, which enables users to organize and stream personal media collections (audio, video, images) across devices, often installed on home servers or *NAS* systems.
- *Nextcloud*[11], an open-source and self-hosted content collaboration platform for file storage, calendars, document editing, and more through a unified interface.
- *Gitea*[12], a lightweight, open-source and self-hosted *Git* service offering code hosting, collaboration tools, continuous integration, and issue tracking.
- *Home Assistant*[13], an open-source home automation platform that prioritizes local control and privacy, integrating with a wide range of devices and protocols through a modular architecture.

These examples demonstrate the breadth and maturity of modern self-hosted ecosystems, showcasing the practical viability of self-hosted solutions across domains, from productivity and media to development and smart home automation.

Deploying and maintaining a self-hosted podcast library management system presents several challenges. Unlike centralized platforms, where scalability, reliability, and content discovery are managed by the provider, a self-hosted solution requires the administrator to handle all aspects of the system lifecycle. Ensuring reliable podcast feed synchronization is non-trivial, as it involves periodically fetching and parsing large numbers of *RSS* feeds that may vary in structure, stability, and compliance with standards. Storage management is also a concern, since audio files can grow rapidly in size, requiring efficient use of disk space, support for cleanup and archival strategies, and integration with backup solutions to prevent data loss. In addition to managing the media itself, the system must also store and synchronize user-specific data, such as listening history and playback progress across multiple devices, which introduces further complexity in data consistency and state management. Providing seamless access across devices demands careful configuration of networking, authentication, and authorization mechanisms, while balancing security with usability. Finally, maintaining up-to-date software and managing dependencies is essential to prevent vulnerabilities, yet it places a continuous operational burden on the host. These

[10] `https://github.com/awesome-selfhosted/awesome-selfhosted`
[11] `https://nextcloud.com/`
[12] `https://about.gitea.com/`
[13] `https://www.home-assistant.io/`

challenges highlight the trade-off between the control and independence offered by self-hosting and the convenience of relying on third-party podcast platforms.

2.3. Existing Solutions

A self-hosted podcast library management system is not only for media management but also for media consumption. Therefore, this type of system can generally be divided into two main components: the backend and the frontend. The backend typically runs as a service on a server and is responsible for resource-intensive tasks such as managing media files, handling user accounts and permissions, and exposing media content through well-defined interfaces. The frontend, in turn, provides the user-facing layer of the system, which may take the form of a web application or a native client, and interacts with the backend through an *Application Programming Interface (API)* to, among other things, enable the reproduction of the media content. Together, these components enable efficient organization, secure access, and seamless consumption of media across different platforms and devices. A diagram of a high-level architecture of such a system is shown in Figure 1.

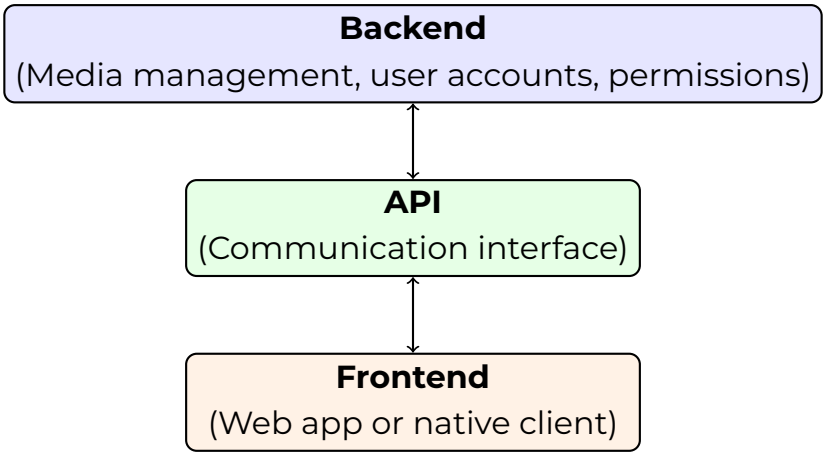


FIGURE 1. High-level architecture of a media management and consumption system

Before starting any planning or development, a search was conducted to identify open-source and self-hosted solutions and compare functionalities between them. As a result, the following three solutions were found: Audiobookshelf[14], PodFetch[15] and Podgrab[16].

Despite the name, Audiobookshelf (created on 2021-08-17) is a server not only for audiobooks, but also for podcasts, as of 2022-04-22, featuring: auto-download of new episodes, multiple accounts support with custom permissions, directly playing episodes through the web application, synchronisation of listening progress but relies on its own apps (Android and iOS) for mobile playback and synchronisation.

[14] `https://github.com/advplyr/audiobookshelf`
[15] `https://github.com/SamTV12345/PodFetch`
[16] `https://github.com/akhilrex/podgrab`

Podgrab (created on 2020-10-06) is the oldest and first most popular open-source project focused on auto-downloading episodes, with support to playback through the integrated player.

PodFetch (created on 2023-02-18) looks to be the sucessor of Podgrab, having the same features with some great additions such as save of listening progress and synchronisation through the Gpodder web-service.

Table 1 shows a comparison of features and other information between the three mentioned solutions as of March 9, 2024.

	Audiobookshelf	**PodFetch**	**Podgrab**
Auto-download episodes	Yes	Yes	Yes
Play episodes	Yes	Yes	Yes
Sync progress	Yes	Yes	No
Multiple-accounts	Yes	No	No
Gpodder integration	No	Yes	No
Extensions support	No	No	No
GitHub Stars[4]	4.9k	277	1.5k
Created on	2021-08-17	2023-02-18	2020-10-06
Last update	1 day ago	4 days ago	2 years ago

TABLE 1. Comparison of existing solutions

In terms of implementation technologies, the three solutions take different approaches. Audiobookshelf employs a *Node.js* backend, a widely used JavaScript runtime environment designed for scalable network applications. Its frontend is implemented with *Vue.js*, a progressive JavaScript framework, and is complemented by mobile applications built with *Nuxt.js* and *Capacitor*, which enable cross-platform deployment for Android and iOS. For persistence, Audiobookshelf uses *SQLite*, a lightweight relational database, while media files are stored directly in the file system.

PodFetch, by contrast, implements its backend in *Rust*, a systems programming language that emphasizes safety and performance. Its frontend is written in *React*, a declarative and component-based *JavaScript* library, with *TypeScript* providing static type checking for improved maintainability. For database support, PodFetch offers both *SQLite*, suited for lightweight deployments, and *PostgreSQL*, a robust relational database with advanced features for scalability. Media storage can be handled by the local file system or through *S3-compatible object storage*, enabling integration with cloud providers.

Podgrab provides a simpler architecture with a backend written in *Go*, a statically typed language developed by Google known for concurrency support and ease of deployment. Like Audiobookshelf, its frontend is developed with *Vue.js*, ensuring a reactive and lightweight user interface. Data persistence relies on *SQLite*, while media files are

[4] Starring a repository shows appreciation for the maintainer's work (GitHub, Inc., 2020)

managed directly in the local file system. A breakdown of the technologies employed by the solutions is displayed in the Table 2.

Solution	**Backend**	**Frontend**	**Database**	**Storage**
Audiobookshelf	Node.js	Vue.js (Web) Nuxt.js/Capacitor (Mobile)	SQLite	File system
PodFetch	Rust	React (TypeScript)	SQLite PostgreSQL	File system S3-compatible
Podgrab	Go	Vue.js (JavaScript)	SQLite	File system

TABLE 2. Technologies in self-hosted podcast and media management solutions

2.4. Plugins

In software development, *plugins*, also known as *extensions*, are modular components that integrate with an application to extend or customize its functionality. They enable developers and users to enhance core systems with additional features only when needed, providing flexibility, scalability, and a more tailored experience for different use cases. This modular architecture promotes reusability, simplifies maintenance, and allows applications to evolve without modifying their core codebase.

Plugin systems have become a cornerstone of modern software ecosystems, enabling communities to develop new capabilities while maintaining a clear separation between the core platform and third-party extensions. Their design and implementation, however, vary significantly depending on the goals, programming language, and runtime model of each application. For example, *Home Assistant* implements a plugin architecture through custom integrations written in Python, leveraging the interpreted nature of the language to dynamically load and execute code at runtime. This approach provides great flexibility for rapid prototyping and community-driven extensions but can introduce performance overhead and dependency management challenges (Home Assistant, 2025).

Similarly, *WordPress*[17], one of the most widely used content management systems, relies on plugins written in *PHP*. Its architecture allows plugins to hook into core functionality using predefined filters and actions, making it highly extensible without requiring recompilation. However, since all plugins share the same runtime environment, poor coding practices or security vulnerabilities in a single plugin can affect the stability of the entire system (WordPress, 2023; Titterington, 2025).

In contrast, *Visual Studio Code*[18] adopts a more structured extension framework built around *JavaScript* and *TypeScript*. Extensions run in isolated processes and communicate with the main application through a well-defined *API*, ensuring both safety and consistency. This model strikes a balance between performance and flexibility, allowing

[17] `https://wordpress.org/`
[18] `https://code.visualstudio.com/`

developers to modify the user interface, add language support, or introduce new tools without risking the stability of the editor (Microsoft, 2025).

Multimedia applications such as *VLC Media Player*[19] employ a hybrid approach that supports both native plugins written in *C* or *C++* and interpreted *Lua* scripts. Native extensions offer superior performance and deeper integration with system resources, but they require compilation for multiple operating systems and architectures, complicating distribution and maintenance. *Lua* extensions, by contrast, are cross-platform and easier to deploy, although they provide limited access to low-level functionality (VLC, 2025; verghost, 2022).

More recently, platforms such as *Shopify*[20] have embraced *Wasm* as a foundation for their plugin ecosystems. By supporting extensions compiled to *Wasm*, *Shopify* allows developers to build secure, high-performance backend extensions using multiple programming languages. This approach combines the safety of a sandboxed environment with near-native execution speed, while also ensuring that plugins remain portable and consistent across different environments. The adoption of *Wasm* for extensibility marks a significant shift in modern plugin architecture, offering a promising solution to long-standing challenges related to language interoperability, isolation, and deployment complexity (Uszkay, 2020; Shopify, 2025).

These examples illustrate the diversity of plugin architectures and the trade-offs between flexibility, performance, and maintainability. The choice of implementation depends on the application's requirements, the programming languages involved, and the desired balance between extensibility and security.

[19] `https://www.videolan.org/`
[20] `https://www.shopify.com/`

[This page has been intentionally left blank]

CHAPTER 3

Development

The development of the self-hosted podcast library management system was divided into two main parts: the backend and the frontend. In the following sections, a deep dive into all the components of each part is presented.

3.1. Planning

The planning phase played a crucial role in defining the scope of the project and establishing a clear implementation strategy. This stage began with the analysis of the state of the art, from which the main requirements and design objectives were identified. These findings were then translated into a set of core features that define the foundation of the solution. To ensure efficient development and alignment with user needs, the features were subsequently prioritized as follows:

1. Open source
2. Self-hosted
3. Support for multiple user accounts
4. Fetching of podcast episodes
5. Episode playback functionality
6. Synchronization of playback progress
7. Plugin / extension support
8. Automatic download of new episodes

The prioritization of these features reflects both the project's guiding principles and its intended use cases. Emphasizing an open-source and self-hosted design ensures transparency, user autonomy, and long-term maintainability, which are key values for software intended to empower its community and operate independently of third-party services. Core functionalities such as multi-account support, podcast retrieval, and playback were prioritized next to establish a solid foundation for usability. Features like progress synchronization and automatic downloads were included to enhance user experience and convenience, while plugin / extension support were planned to promote scalability and future extensibility of the system.

3.2. Architecture

As illustrated in Figure 2, the solution is built around a modular client–server architecture designed for performance, scalability, and extensibility. The backend is implemented in *Go*, a modern programming language known for its simplicity, concurrency support, and efficient memory management. *Go* was chosen not only for its strong ecosystem in

server-side development but also to provide an opportunity to explore a new language well-suited for building lightweight, reliable network services. While alternatives such as *Rust* offer superior execution speed and stronger guarantees of memory safety, *Go*'s straightforward syntax and ease of onboarding make it particularly effective for rapid development and maintainability in a small-scale, open-source environment. The backend application is configured through environment variables, which define the *PostgreSQL* database connection and other operational parameters, and it is responsible for fetching and processing podcasts' *RSSs* feeds.

The frontend is developed as a *Single-Page Application (SPA)* using *Vue.js*, a progressive *JavaScript* framework known for its reactive, component-based architecture and ease of integration with backend services. While other modern frameworks such as *React* or *Angular* offer similar capabilities for building dynamic and modular user interfaces, *Vue.js* was selected due to prior experience with its ecosystem and its balance between simplicity and flexibility. The frontend is served directly by the *Go* backend, allowing the system to operate as a unified service without requiring an external web server. Communication between the client and the server occurs through a combination of a *REST API* for standard data operations and a *Socket.IO* channel for real-time updates, ensuring a responsive and consistent user experience across devices.

A key aspect of the system's design is its unified plugin architecture, which supports extensibility on both the backend and frontend. On the backend, plugins are implemented as *Wasm* modules executed in a sandboxed environment managed by the *Extism* framework, ensuring secure and language-agnostic integration with the core application. On the frontend, extensibility is achieved through *Vue.js* components, enabled by the `vue-extensions`[21] module, allowing developers to augment the user interface with new views or features. Both types of plugins are described and configured using a *YAML* manifest file, providing a consistent and declarative mechanism for integration.

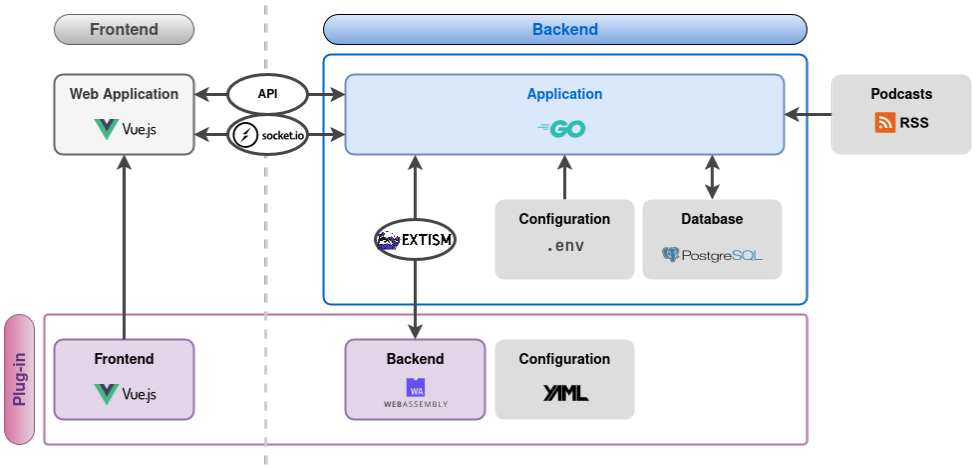


FIGURE 2. Solution architecture

[21] `https://www.npmjs.com/package/vue-extensions`

3.3. Configuration

As with most self-hosted applications, before deploying the solution, it's required to set some configurations. The `viper`[22] *Go* module, allows this configuration to be made through environmental variables (as in docker deployments) and through an `.env` file.

When the server is launched, it first checks for the presence of a file named `.env` in the same folder as the server and loads its configured values. Environmental variables can be used in the absence of the `.env` file, to overwrite any of its configurations or set further configurations.

To assist the administrator during the configuration process, a `.env.example` template file, shown at Listing 2, is provided. This file includes all the configuration fields, organized by related sections, and accompanied by helpful comments.

```
# Connections
BASE_URL=http://localhost
SERVER_PORT=9002

# Database
DB_URL=postgres://username:password@localhost:5432/cassette

# General
LOG_LEVEL=info # Log level can be 'debug', 'info', 'warn', 'error', 'fatal'

# Scheduler
SCHEDULER_MODE=periodic # Mode can be either 'daily', 'periodic' or 'disabled'
SCHEDULER_TIME=02:00 # Time in HH:MM format for daily runs or time interval for
↪  periodic runs

# Session Web token
JWT_SIGNING_KEY=jwt_signing_key
JWT_SESSION_LENGTH=10080
```

Listing 2. Content of .env.example file

3.4. Database

For storage purposes, the solution uses *PostgreSQL*, an open-source *Relational Database Management System (RDBMS)*. A relational database was chosen because the data to be processed and stored is organized in a structured format. This approach takes advantage of the consistency, integrity, and powerful querying capabilities provided by relational database systems. Additionally, using a relational model facilitates future support for other *RDBMS* solutions such as *SQLite*, *MariaDB*, or similar systems. The databse contains tables for podcasts and their episodes, as well as users, their subscriptions, and episode progress tracking. There are also two auxiliary tables: one for keeping track of

[22] `https://github.com/spf13/viper`

the applied database schema migrations, and another to allow plugins to retain data in the form of a key-value store.

The schema and relationship between the database tables are illustrated in the *Entity Relationship Diagram (ERD)* shown in Figure 3.

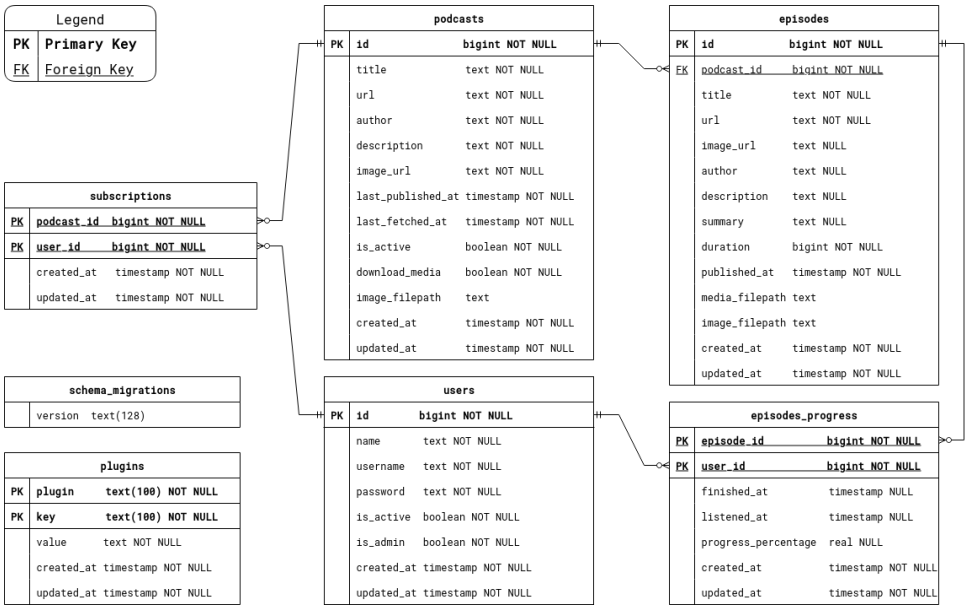


FIGURE 3. Solution database Entity Relationship Diagram

3.4.1. Schema Migrations

Database schema migrations enable versioned changes to the database structure, such as adding new tables or columns. This ensures that the database evolves in a controlled and consistent manner.

To aid in such tasks the solution uses `dbmate`[23], a *Command Line Interface (CLI)* database migration tool that supports *PostgreSQL*, *MySQL*, and *SQLite* databases, that can also be used as a *Go* module to *Embed* the migration files within the compiled application. The migrations are stored in the `migrations` folder and are executed in order by the `dbmate` tool. Each migration file contains a *SQL* script that describes the changes to be made to the database schema, as shown in the migration file at Listing 3 for the user table. Where the *up* and *down* sections specify the actions to take during the migration and the rollback, respectively.

```
1  -- migrate:up
2  CREATE TABLE "users" (
3      "id" bigint GENERATED BY DEFAULT AS IDENTITY NOT NULL,
4      "name" character varying NOT NULL,
5      "username" character varying NOT NULL,
6      "password" character varying NOT NULL,
7      "is_active" boolean DEFAULT true NOT NULL,
```

[23] `https://github.com/amacneil/dbmate`

```
8       "is_admin" boolean DEFAULT false NOT NULL,
9       "created_at" timestamptz NOT NULL,
10      "updated_at" timestamptz NOT NULL,
11      CONSTRAINT "users_pkey" PRIMARY KEY ("id"),
12      CONSTRAINT "users_username_key" UNIQUE ("username")
13  );
14
15  -- migrate:down
16  DROP TABLE IF EXISTS "users";
```

LISTING 3. Content of create users table migration file

3.5. API

Unlike the traditional websites where to display new content the browser requests and refreshes the whole page, in a *Single-Page Application (SPA)* frontend, as the name implies, the whole page is only loaded once and the content can be dynamically changed through requests via *REST API* exposed by the backend. This approach provides a more fluid and responsive user experience, as only the necessary data is fetched and updated without the need for full page reloads, preventing a pause and a *flash* while the browser retrieves and redraws the whole page (Mikowski et al., 2013). From the development perspective, this architecture allows for a clear separation of concerns between the frontend and backend, enabling independent development and maintenance of each part. Additionally, it facilitates the creation of more interactive and engaging user interfaces, as the frontend can update content in real-time based on user interactions or data changes.

The *API* is exposed through *gRPC* and the `gRPC-Gateway` plugin, which automatically generates a *REST API* from the *gRPC* service definitions. This approach ensures a strongly typed interface between the backend and frontend, as well as providing backwards and forwards compatibility when changes are made to the data structures. The *API* documentation is automatically generated using *Swagger UI*, allowing developers to easily explore and test the available endpoints.

3.5.1. gRPC

During the development phase, specially for the first plugin, it was identified the need to assure a unified schema between the main application and the plugins. This was to let the plugins know the fields and types each entity contains, as well as in the scenario of a new field or property being added later and ensure backwards compatibility with existing plugins and future integrations.

In the search for an appropriate communication framework, *gRPC (Google Remote Procedure Call)* was identified as a suitable approach. It is an open-source *Remote Procedure Calls (RPC)* framework originally developed by Google that employs *Protobuf* to define service and message types. This design enables a strongly typed interface between the backend and the plugins, ensuring consistency in data structures and communication.

Once the *gRPC* service was configured, it became evident that it could also serve as the foundation for the *REST API*.

gRPC supports a variety of programming languages and allows a server application to expose some of its internal methods to be called remotely by multiple client applications, which can be written in other languages. In order to do this, a service must be defined along with its underlying methods and corresponding parameters and response types. A diagram of the *gRPC* workflow is shown in Figure 4.

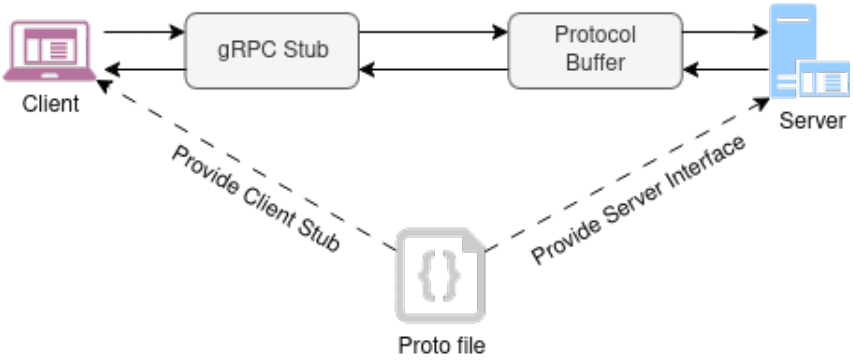


FIGURE 4. gRPC workflow diagram

3.5.2. Protocol Buffers

Protocol Buffers, or *Protobuf* for short, are the *Interface Definition Language (IDL)* used by default by *gRPC* to serialise and define its data structures and methods interfaces. These definitions are written in text files ending with a `.proto` extension, where objects are identified as *messages* that contain fields, as shown in Listing 4. For each field, a cardinality can be set to single (default) or repeated, and if appropriate the field can be set as optional. Additionally, each field must include its data type, name and a field number.

```
message User {
  int64 id = 1;
  string name = 2;
  string username = 3;
  string password = 4;
  bool is_admin = 5;
  bool is_active = 6;
  google.protobuf.Timestamp created_at = 7;
  google.protobuf.Timestamp updated_at = 8;
  optional string role = 9;
}

message UsersPaginated {
  repeated User users = 1;
  int32 total = 2;
}
```

LISTING 4. Sample .proto file

Field numbers allow *Protobuf* to be both backwards and forwards compatible by assigning new numbers for new fields and reserve the numbers of deprecated fields. With this system, previous versions of the code can ignore the new fields and deleted fields will have their default values, while new code can read old messages by ignoring the nearly deprecated fields and new fields get their default values assigned.

Serialization is the process of encoding the data, in this case the message, in raw bytes. One of the big advantages of *Protobuf* is how much faster and smaller its serialization is compared to other formats like *JSON*, allowing for both a more efficient transmission and storage of messages.

As documented by Google LLC, 2022, *Protobuf* turns each key-value pair into a record consisting of the field number, a wire type and a payload. The wire type tells the parser how big the payload after it is. This allows old parsers to skip over new fields they don't understand. This type of scheme is sometimes called *Tag-Length-Value (TLV)*.

In order to integrate *Protobuf* into a project, the standard libraries providing generic serialization and parsing methods for the target programming language should be imported. Additionally, the *Protocol Buffer Compiler (`protoc`)* tool can automatically generate data access and helper classes tailored to the defined data structures in the `.proto` files.

3.5.3. gRPC-Gateway

To bridge the gap between *gRPC* and traditional *HTTP*-based communication, the solution integrates the `gRPC-Gateway`[24] plugin for protoc. `gRPC-Gateway` is an open-source project that acts as a reverse proxy, translating *RESTful HTTP API* requests into *gRPC* messages and forwarding them to the corresponding *gRPC* services. This approach, depicted in Figure 5, allows a single backend implementation to serve both *gRPC* and *REST* clients, ensuring compatibility with standard web technologies such as browsers and *JavaScript*-based frontends.

[24] `https://github.com/grpc-ecosystem/grpc-gateway`

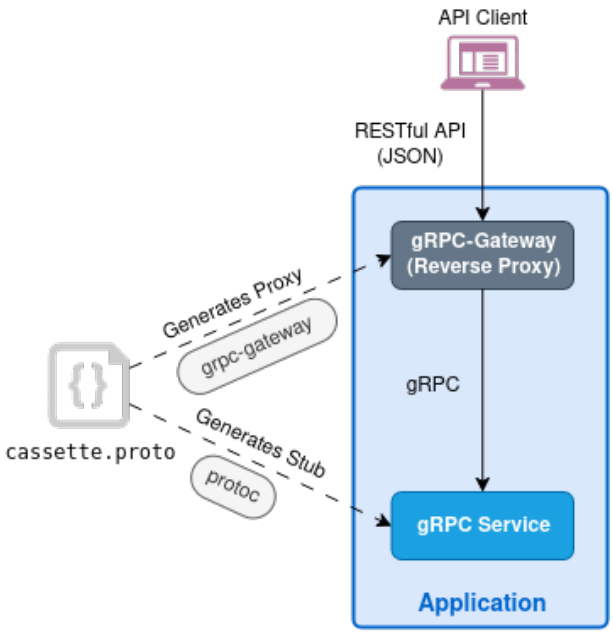


FIGURE 5. gRPC-Gateway workflow diagram

The framework achieves this dual compatibility through `google.api.http` annotations defined within the `.proto` files, where developers can specify the mapping between *gRPC* service methods and their corresponding *HTTP* endpoints, including request paths, methods, and query parameters. As exemplified in the Listing 5 where the *gRPC GetUser* method will be accessed by a *REST API* endpoint at `/user/current`. By leveraging `gRPC-Gateway`, the solution benefits from the efficiency, scalability, and type safety of *gRPC*, while preserving the accessibility and interoperability offered by conventional *HTTP* endpoints.

```
import "google/api/annotations.proto";

message GetUserRequest {}
message GetUserResponse {
  User user = 1;
}

service CassetteService {
  rpc GetUser(GetUserRequest) returns (GetUserResponse) {
    option (google.api.http) = {
      get: "/user/current"
    };
  }
}
```

LISTING 5. Sample protobuf service file

3.5.4. Swagger UI

Swagger UI[25] is an open-source tool that provides a user-friendly interface for visualizing and interacting with *REST API* endpoints. It dynamically generates interactive *API* documentation based on an *OpenAPI* specification, allowing the exploration of the available *API* methods, as well as the visualization of their parameters and example responses. *Swagger UI* also facilitates testing *API* calls directly from the interface.

Using the `gRPC-Gateway` plugin, it is possible to automatically generate the *OpenAPI* specification needed for *Swagger UI*. When combined with the `swgui`[26] package, this enables the solution to expose its *API* documentation seamlessly. A screenshot of the *Swagger UI* is shown in Figure 6, where the available endpoints are listed.

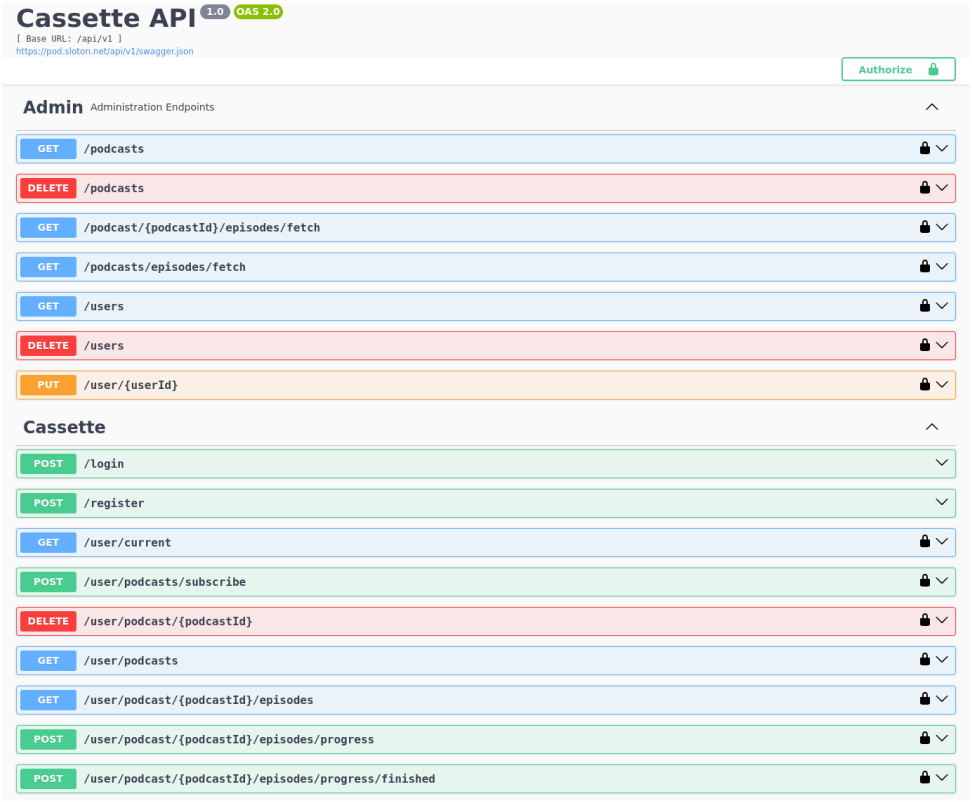


FIGURE 6. Screenshot of the Swagger UI

3.6. Authentication & Authorization

As a multi-user web application that may be exposed to the public internet, the solution must ensure that access to resources is properly secured. This requires mechanisms not only to verify the identity of users (authentication) but also to enforce fine-grained access control to guarantee that each user can only interact with their own data (authorization). The solution implements a *Role-Based Access Control (RBAC)* system, where permissions are assigned to specific roles such as regular users and administrators. For instance, a regular user should be able to subscribe to podcasts, track listening progress, and manage

[25] `https://swagger.io/tools/swagger-ui/`
[26] `https://github.com/swaggest/swgui`

personal preferences, while an administrator may additionally create, edit, or remove podcasts for the entire system. To address these requirements, the solution incorporates a dedicated authentication and authorization system as a core component of its architecture.

3.6.1. Password Hashing

For authentication, the solution uses password hashing to securely store user passwords. The `bcrypt` algorithm is employed for this purpose, which is a widely accepted and secure method for hashing passwords. `bcrypt` is a key derivation function that applies a one-way hash function to the password, incorporating a cryptographic salt to ensure that identical passwords result in different hashes. It also uses a configurable work factor, which determines the computational cost required to compute the hash, making brute-force attacks significantly more difficult. This process makes it computationally expensive to reverse-engineer the original password from the hash, helping to protect user passwords even if the database is compromised (Auth0, 2021).

When a user registers, the plaintext password is hashed using `bcrypt` before being stored in the database. During login, the provided password is hashed again and compared with the stored hash to verify the user's identity.

3.6.2. JSON Web Token (JWT)

After the user is authenticated, the solution generates a *JSON Web Token (JWT)* containing the user's ID to manage user sessions, i.e. to keep the user authenticated. *JWT* is an open standard (RFC 7519) that defines a compact claims[27] representation format intended for space constrained environments such as *HTTP* `Authorization` headers and *URI* query parameters. *JWTs* are used to securely transmit information between parties as a *JSON* object. They can be digitally signed or integrity protected with a *Message Authentication Code (MAC)* and/or encrypted to provide secrecy between parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it. A sample *JWT* in the compact serialization form is shown in Listing 6, where the token, normally in one line, is split in multiple lines for readability.

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
2 eyJleHAiOjE3NTQ0MTA0NTMsInN1YiI6IjIifQ.
3 rgM3Na11bKxKeYCfbTWNiuyHazF5501lfUrOavvOu6A
```

Listing 6. Sample JWT

This approach enables client-side sessions (also know as stateless sessions) by making the client responsible for securely storing the *JWT* and ensuring it is sent in the `Authorization` header of each request. This allows the server to verify the user's identity and permissions by fetching the user ID from the token and reduce the database queries

[27] A claim is a piece of information asserted about a subject and represented as a name/value pair.

needed to check the user's session (Peyrott, Sebastian E et al., 2018). Sending the token in the `Authentication` header prevents *Cross-Origin Resource Sharing (CORS)* issues, as it does not rely on cookies, which are subject to *CORS* restrictions (Auth0, 2018).

JWTs are composed of three parts: a header, a payload, and a signature/encrypted data (rows 1, 2 and 3 in Listing 6, respectively). The header typically contains the type of token, the algorithms used and whether the *JWT* is signed or encrypted. The payload contains the claims, which can include standard claims like *sub* (subject), *iss* (issuer), and *exp* (expiration time), as well as custom claims defined by the application. Just like the header, the payload is a *JSON* object. The signature is created according to the algorithm specified in the header, typically using a secret key or a private key. The signature is used to verify the message wasn't changed along the way and if signed with a private key, it also verifies the sender's identity.

Since the *JWT* generated by the solution, only contains the user ID in the *sub* claim ("subject") there is no need to encrypt the token, as it does not contain sensitive information. The *JWT* is signed using a secret key stored in the server's environment variables, which is used to verify the token's integrity. The decoded *JWT* is shown in Listing 7, where the header contains the *alg* claim informing the signing algorithm was *HMAC* using *SHA-256* and the *typ* claim clarifying that it refers to a *JWT*. The payload contains the *exp* claim that indicates the expiration time of the token in seconds since the epoch (January 1, 1970), which is set to `"2025-08-05 17:14:13 GMT+0100"` in this example. The *sub* claim contains the user ID, which is set to `"2"` in this case.

```
1  // Header
2  {
3    "alg": "HS256",
4    "typ": "JWT"
5  }
6
7  // Payload
8  {
9    "exp": 1754410453, // Expiration time in seconds: 2025-08-05 17:14:13 GMT+0100
10   "sub": "2"         // User ID
11 }
```

Listing 7. Sample Decoded JWT

3.6.3. Middleware

In web development, *middlewares*, also known as *interceptors* or *filters*, are methods that encapsulate functionality common to multiple requests or independent of specific request logic. These methods can be chained together, allowing for sequential execution, and can be hooked into either the pre-processing or post-processing phases of a request. This allows them to modify the request before it reaches the main handler or to alter the response

before it is returned to the client (Eric Anderson, 2024; Johan Brandhorst-Satzkorn, 2024). Such characteristics make middlewares particularly well-suited for:

- Authentication
- Authorization
- CORS handling
- Logging
- Caching

Based on these use cases, the solution implements three middlewares during the pre-processing phase. The first handles *CORS*, as detailed in 3.6.4. The second validates the authenticity and validity of the *JWT*, securing web endpoints, also known as routes, that require authentication. The third protects administrative routes by verifying whether the user has administrative privileges. If any middleware validation fails, the execution chain is halted, and an error response is returned, preventing the request from reaching the main handler.

The *JWT* and admin middlewares are implemented using the `gRPC-Gateway`'s `Middleware` type[28]. Unlike standard middleware implementations that allow specifying which middleware applies to individual routes, `gRPC-Gateway` middlewares are applied globally to all routes. To work around this limitation, the middlewares implement an *Access Control List (ACL)* logic: for each incoming request, they check whether the route is listed as unauthenticated (for the *JWT* middleware) or as admin-protected (for the admin middleware).

3.6.4. Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers to restrict web applications running on one origin (domain, protocol, and port) from interacting with resources hosted on a different origin. This policy, known as the *Same-Origin Policy (SOP)*, is a fundamental mechanism for maintaining web security, but it also limits the ability of applications to make cross-origin requests.

CORS provides a controlled way to relax this restriction by allowing servers to specify which origins are permitted to access their resources. This is achieved through the inclusion of specific *HTTP* headers in the server's response, such as:

- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Credentials`

When a browser detects a cross-origin request, it first checks whether the target server permits such an operation. Certain types of requests, known as simple requests, are automatically allowed without requiring a preflight check. A request qualifies as simple if it meets all of the following conditions:

[28] `https://pkg.go.dev/github.com/grpc-ecosystem/grpc-gateway/v2/runtime#Middleware`

- The *HTTP* method is one of:
  - `GET`
  - `HEAD`
  - `POST`
- The request headers are limited to:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type`
- If the `Content-Type` header is used, its value must be one of:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`
- The request does not include custom headers or send credentials such as cookies or authorization tokens.

For all other cases, classified as non-simple requests, the browser first issues a preflight request using the `OPTIONS` method to verify that the server explicitly allows the intended operation before proceeding with the actual request (Mozilla Corp, 2025).

In this solution, CORS is handled using a middleware[29] that inspects incoming requests and appends the appropriate *CORS* headers to the response. This middleware is designed to allow requests from trusted origins while denying unauthorized cross-origin traffic. In the development environment, the `Access-Control-Allow-Origin` header is set to `"*"` to allow requests from all origins, as depicted in Figure 7. This is particularly useful when the frontend is served from a different port than the backend. In the production environment, however, the header is set to the value of the `BASE_URL` environment variable to restrict access to same-origin requests only.

[29] `https://github.com/rs/cors`

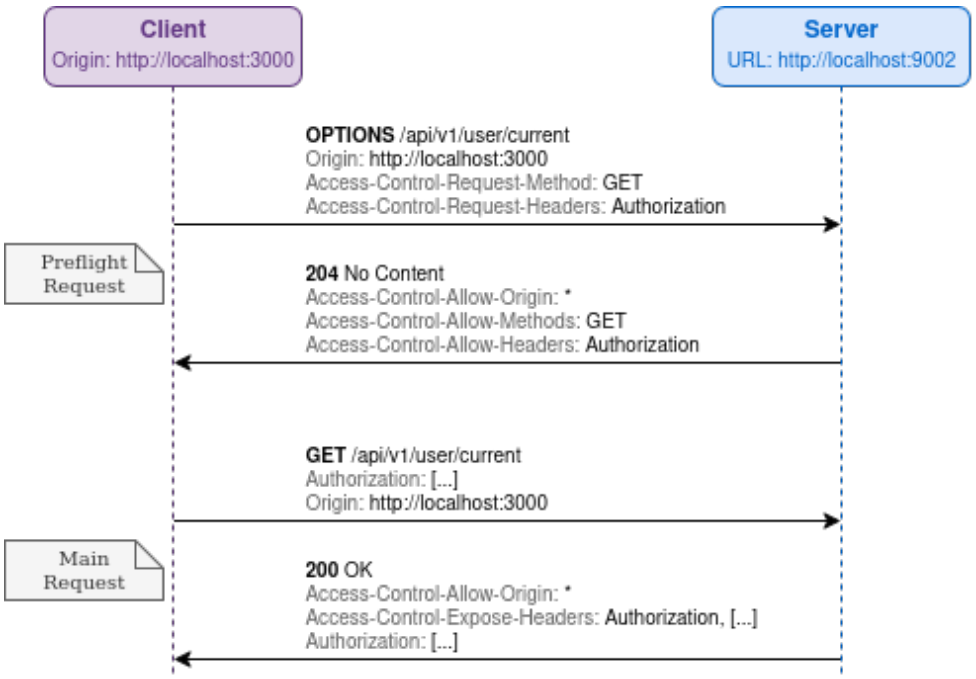


FIGURE 7. Sequence diagram of a CORS request

3.7. Plugins

The solution currently supports plugins that can extend both the backend (server) and the frontend (client), with each plugin capable of affecting either one or both components. Plugins are distributed as folders that contain a manifest in *YAML* along with the corresponding backend and/or frontend code. During server initialization, the solution scans the designated plugins directory for folders containing a manifest file named `.cassette.yaml`, which provides the necessary metadata for registration and loading. An example of a plugin manifest is shown at Listing 8.

```yaml
name: opml
display_name: OPML
description: Enables exporting and importing subscriptions using OPML files.
author: Shaxine
version: 0.2
backend:
  wasm_path: plugin.wasm
frontend:
  folder_path: "frontend/OpmlExtension"
```

LISTING 8. Sample plugin manifest file

3.7.1. Backend

While researching for options on how to implement a plugin system compatible with the *Go* programming language, three different approaches were found:

- go-plugin[30]

  Developed and maintained by HashiCorp, the creators of Terraform[31], `go-plugin` is a Go plugin system over *RPC* through a local network, where the main application is the server and the plugin is the client. It also supports *gRPC*, allowing for a plugin to be written in other major programming languages. In order to consume the plugin the main application executes the compiled version of the plugin in a subprocess, preventing a crash from the plugin to propagate to the main application. While it supports multiple languages, the plugins must be compiled in a format compatible with the main application and host system.
- Yaegi[32]

  Yaegi which stands for *Yaegi is Another Elegant Go Interpreter* by the creators of Traefik[33], as the name implies, is a *Go* interpreter, which means that it can evaluate and execute *Go* scripts at runtime, without the need to first compile to an executable format, allowing the scripts to work everywhere *Go* runs and making it possible to execute scripts dynamically by embedding yaegi into a *Go* application and calling them when needed. It has limited support to external dependencies (modules) outside *Go standard libraries*.
- Extism[34]

  Extism by Dylibso[35] is a cross-platform plugin system powered by *Wasm*. Extism provides *Host Software Development Kits (SDKs)* to manage and run plugins for *Go* and other 16 programming languages. Plugins can be written in *Go* and 7 more languages with the support of what Extism calls *Plugin Development Kits (PDKs)* to ease the plugins' development and shared memory management and access. The plugin code is then compiled to *Wasm* and distributed as an `.wasm` binary artifact. Allowing plugin authors to write in their chosen language while the plugin runs wherever the main application runs.

Among the available options, Extism proved to be the most promising, as it provides greater flexibility for both the main application and its plugins. Through its development kits, plugin execution remains agnostic to the host platform, ensuring portability and simplifying deployment across different environments. Furthermore, by supporting multiple programming languages for plugin development, Extism lowers the barrier to adoption,

[30] `https://github.com/hashicorp/go-plugin`
[31] `https://www.terraform.io/`
[32] `https://github.com/traefik/yaegi`
[33] `https://traefik.io/`
[34] `https://github.com/extism/extism`
[35] `https://dylibso.com/`

enabling developers to use familiar languages and ecosystems while still benefiting from a unified runtime.

As a result, Extism was selected, and as a *PoC*, a plugin was developed to enable the export and import of podcast subscriptions in the *OPML* format. *Outline Processor Markup Language (OPML)* is an *XML*-based standard widely used by podcast applications to exchange subscription lists, making it an ideal choice for ensuring interoperability across different clients.

From the host's perspective, exporting a user's subscribed podcasts requires the plugin to register a web endpoint accessible to the frontend and to retrieve the user's subscription data. Conversely, importing podcast subscriptions from an *OPML* file requires the plugin to access the podcast catalog in order to search for existing entries, as well as to create new podcasts, episodes, and subscriptions when necessary.

For the solution to support plugins through Extism, the *Go Host SDK*[36] was employed. This *SDK* is responsible for loading plugins in the *Wasm* format from the plugins directory, exposing the host-side functions required by the plugins (referred by Extism as *Host functions*), and invoking the functions made available by the plugin itself, which Extism refers to as *Export functions*. For example, the host may call an initialization function if one is defined, as well as other export functions exposed by the plugins during execution.

Regarding the plugin implementation, the backend was also written in *Go* for the sake of simplicity. Consequently, the *Go* PDK[37] is required to expose the plugin's *Export Functions* to the host and to import *Host Functions* into the plugin.

Since Extism is language agnostic, the host and *Wasm* components rely on different *Application Binary Interfaces (ABIs)* for variable assignment and function invocation. In order to enable data exchange between the host and the plugin functions, Extism encodes the required values as bytes, writes them to memory, and passes the corresponding memory address to the other side, which then reads and decodes the data prior to use. Because the host and the plugin manage memory differently, Extism provides an isolated memory space under its own management, ensuring safe and consistent communication between both environments. Figure 8 illustrates how Extism encodes, stores, and transfers data between host and plugin memory spaces

[36] `https://github.com/extism/go-sdk`
[37] `https://github.com/extism/go-pdk`

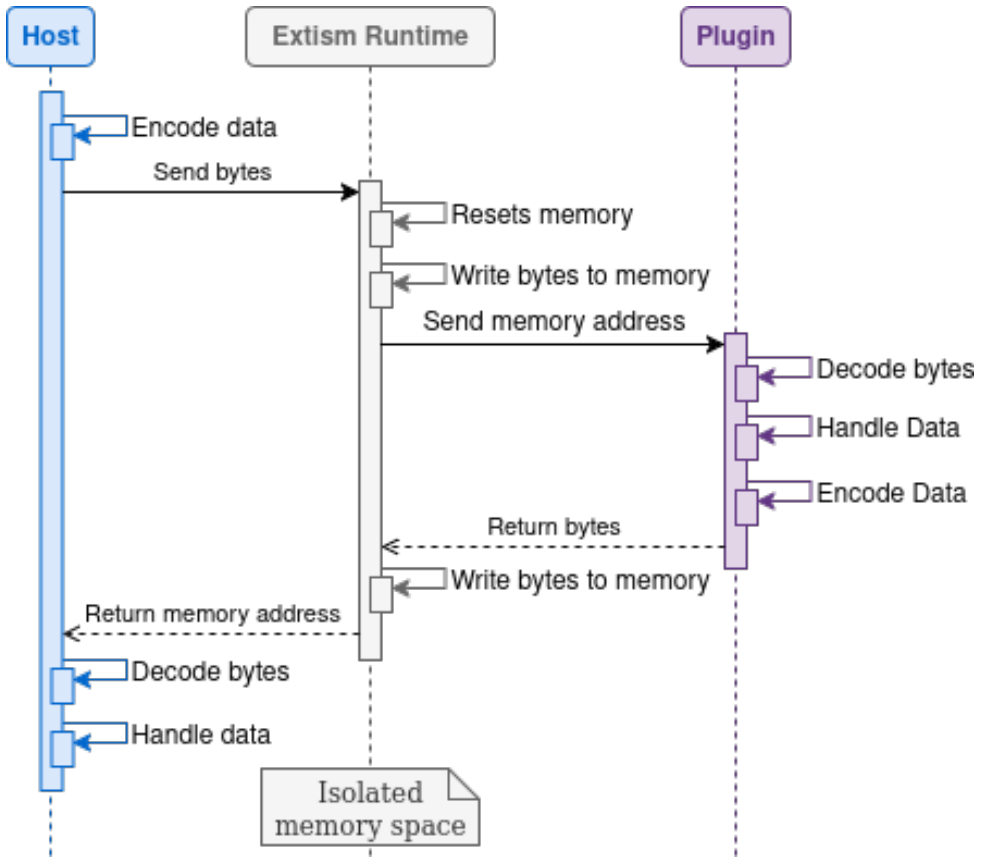


FIGURE 8. Extism memory management sequence diagram

3.7.2. Frontend

On the frontend, the plugin system is implemented using `vue-extension`[38], a library designed to extend *Vue.js* applications with modular and dynamically loadable components and web routes, also know as pages. This approach enables third-party developers to enhance the user interface without modifying the core application code. This design promotes flexibility and maintainability, as the base application remains lightweight while still offering a mechanism for continuous feature expansion through community-driven plugins.

The `vue-extension` library enables the declaration of `extensionpoint` components, each identified by a `hook` name. These components mark locations within the interface where plugins can later inject their own elements. In practice, maintainers place `extensionpoint` components directly in the *Vue.js* templates, thereby exposing well-defined integration points. Listing 9 illustrates how such an extension point is declared in the code.

[38] `https://github.com/nerdocs/vue-extensions`

```
<template>
  <v-container>
    <v-card height="100%" width="100%" flat class="d-flex flex-column">
      <v-card-title class="d-flex align-center">
        <v-icon icon="mdi-podcast"/>
        Subscribed Podcasts
        <v-btn prepend-icon="mdi-plus" rounded color="primary"
        ↪  variant="tonal">Add</v-btn>
        <v-menu>
            <v-list>
                <extensionpoint hook="user-podcasts.header.menu"/>
            </v-list>
        </v-menu>
        <v-text-field label="Search"/>
      </v-card-title>
    </v-card>
  </v-container>
</template>
```

LISTING 9. Sample extension point component

From the plugin's perspective, interaction with an `extensionpoint` involves registering one or more components to be rendered at the corresponding `hook`. To achieve this, the plugin exports a `default` object that defines a `hooks` property, which contains an array mapping hook identifiers to the components intended for injection, as shown in Listing 10. During application initialization, the `vue-extension` library processes these declarations and automatically renders the plugin components at their designated extension points.

```
import ExportOpmlButton from "./components/ExportOpmlButton.vue";
import ImportOpmlButton from "./components/ImportOpmlButton.vue";

export default {
  hooks: {
    "user-podcasts.header.menu": [{component: ExportOpmlButton}, {component:
    ↪  ImportOpmlButton}],
  }
}
```

LISTING 10. Sample plugin hooking

3.8. Scheduler

A scheduler is a system component responsible for executing predefined tasks at specific intervals or under certain conditions, without requiring direct user interaction. It provides a mechanism to automate repetitive operations, ensuring that important processes are carried out consistently and on time.

In the context of this solution, the scheduler is used to manage background jobs that keep the podcast library up to date and reliable. These jobs include periodically fetching new podcast episodes, downloading them as they become available, and retrieving metadata such as episode durations when this information is missing from the feed. By automating these operations, the scheduler reduces manual effort and ensures the library remains complete and consistent.

For scheduling functionality, the solution relies on the `gocron`[39] module for *Go*, which provides a flexible and reliable framework for task automation. `gocron` was chosen for its support for various job types (e.g., periodic, daily and cron), job concurrency handling, simplicity of use, and active community support, which makes the definition of recurring jobs both intuitive and expressive. Within this solution, three scheduling modes are supported and can be configured as described in section 3.3. These modes are:

- **Periodic**: Executes background jobs at fixed time intervals (e.g., every 1 hour). This mode is suitable for continuously checking for updates such as new podcast episodes or metadata refreshes.
- **Daily**: Executes background jobs once per day at a specified time. This mode is useful when regular updates are sufficient but resource usage should be minimized, for example in low-power environments.
- **Disabled**: Deactivates the scheduler entirely. In this mode, background jobs are not executed automatically and must be triggered manually, which can be useful for testing or highly constrained deployments.

3.9. Socket.IO

In order to keep track of episode listening progress in real time, the solution requires a communication mechanism between the client and the server. As a user listens to an episode, the client must continuously notify the server of the current playback position so that the progress can be stored and synchronized. The server, in turn, responds with updated information, such as whether the episode should be marked as finished, since the logic governing these decisions is maintained centrally on the server side. This bidirectional exchange highlights the need for a reliable and efficient real-time communication channel. For this purpose, the solution adopts `Socket.IO`[40].

`Socket.IO` is a widely used library that simplifies the implementation of real-time, bidirectional communication between clients and servers. Built on top of the WebSocket protocol, it provides an abstraction layer that not only manages the underlying connection

[39] `https://github.com/go-co-op/gocron`
[40] `https://socket.io/`

but also extends it with additional features designed to improve reliability and ease of development. `Socket.IO` supports event-based communication, where both the client and server can emit and listen to custom events, making the interaction model more intuitive and flexible than raw WebSocket message handling.

One of the key advantages of `Socket.IO` over regular WebSockets lies in its ability to handle network interruptions gracefully. Whereas a standard WebSocket connection will simply drop in the event of a disconnection, `Socket.IO` automatically attempts to reconnect and resynchronize state, reducing the need for manual recovery logic. Additionally, it provides built-in fallbacks to other transport mechanisms (such as *HTTP* long polling) when *WebSockets* are not available, ensuring compatibility across a wider range of browsers and network conditions. These features, combined with native support for broadcasting messages to multiple clients and fine-grained namespace and room management, make `Socket.IO` a more robust and developer-friendly solution for real-time web applications.

For its implementation, the server integrates the `zishang520/socket.io`[41] module for Go, chosen for its active maintenance and support of the latest `Socket.IO` specification (version 4). When a client establishes a connection, the server employs a `Socket.IO` middleware to verify and validate the *JWT* used for authentication; if the token is invalid, the client is immediately disconnected. Once authenticated, the server listens for the `setEpisodeProgress` event, which is triggered by the client to update the user's listening progress for a given episode. The outcome of this operation is then communicated back to the client through a `Socket.IO` acknowledgement, ensuring that the update is confirmed and synchronized reliably.

On the client side, the implementation relies on the official *JavaScript* `Socket.IO` client library[42]. The client establishes a connection with the server whenever the user authenticates (e.g., after logging in) or when the application is reloaded (e.g., following a page refresh). Once connected, as the user listens to an episode, the client emits `setEpisodeProgress` events containing the percentage of the episode that has been completed. To avoid overwhelming the server with frequent updates, event emission is regulated by a throttling mechanism, ensuring that no new event is sent if a previous one occurred within the last 30 seconds. This threshold was chosen as a compromise between responsiveness and efficiency: it is short enough to provide meaningful synchronization across devices, while long enough to reduce unnecessary server load and database writes.

An alternative approach, such as debouncing, was considered but ultimately not adopted. Debouncing would delay updates until playback paused or stopped, which risks losing progress information if the application closes unexpectedly. Throttling was therefore deemed the most appropriate strategy, as it provides timely synchronization while keeping the implementation simple and reliable.

[41] `https://github.com/zishang520/socket.io`
[42] `https://github.com/socketio/socket.io`

`Socket.IO` lays the foundation for a future notification system, that can be used to notify the users when new episodes become available.

3.10. Frontend

The frontend represents the user-facing layer of the solution, responsible for delivering an intuitive and responsive interface through which users interact with the system. It communicates with the backend via the *API* detailed in section 3.5 to retrieve, display, and update data in real time, ensuring a seamless user experience. The implementation is built with *Vue.js*[43], a progressive *JavaScript* framework chosen for its reactive data binding, component-based architecture, and gentle learning curve, which streamline development and improve maintainability. This choice was further supported by the author's previous experience with the framework, its comprehensive documentation, and the strength of its open-source community. The frontend is served directly by the backend, making accessibility across devices essential; therefore, the interface is designed to be fully mobile-friendly to accommodate users accessing the service through smartphones and tablets. To ensure a consistent and modern design, the project integrates *Vuetify*[44], a *Material Design* component library for Vue. Together, these technologies provide a cohesive, performant, and accessible frontend suited for diverse usage contexts.

The frontend architecture follows a modular and organized structure designed to promote scalability, maintainability, and clarity. At its core, the application is divided into four primary layers: routes, pages, components, and store. The `routes` define the navigation structure of the application, mapping *URL* paths to their corresponding views and enabling seamless transitions between sections such as the podcasts library, episodes view, and settings. Each route typically renders a `page`, which represents a high-level view responsible for orchestrating data fetching, layout, and interactions specific to a given context. Pages are composed of reusable `components`, smaller and self-contained elements such as buttons, cards, and player controls that encapsulate logic, presentation, and behavior, ensuring consistency and reusability throughout the interface. The diagram shown in Figure 9 illustrates the relationships between these layers, highlighting how components are nested within pages, which are in turn linked to routes.

[43] `https://vuejs.org/`
[44] `https://vuetifyjs.com/`

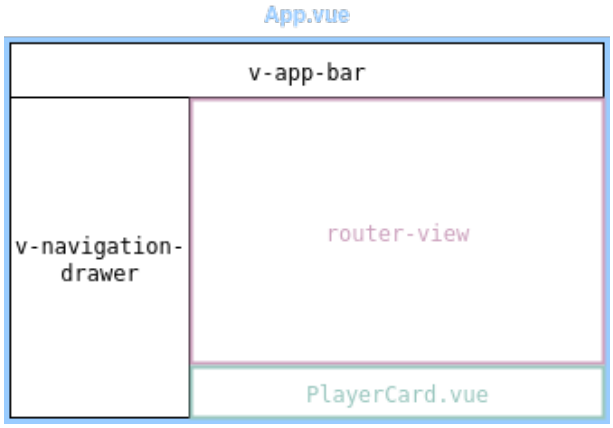


FIGURE 9. Layout structure of the Vue frontend

To manage the application's global state, the solution employs `Pinia`[45], *Vue*'s official state management library. `Pinia` serves as the single source of truth for shared data, such as, user information, playback progress, and podcast metadata. Allowing components to remain decoupled while maintaining synchronization across the interface. It provides a simple yet powerful *API* that supports modular stores, type safety, and persistence, improving both developer experience and application performance. By structuring the frontend in this way, the solution achieves a clear separation of concerns, simplifies maintenance, and facilitates the addition of new features or views without compromising system integrity.

3.11. Docker

Docker[46] is an open-source platform designed to automate the deployment, scaling, and management of applications through lightweight, portable containers. Unlike traditional *Virtual Machines (VMs)*, containers package an application together with its dependencies, libraries, and runtime environment, while sharing the host operating system's *kernel*. A comparison between both architectures can be seen in Figure 10. This approach reduces overhead and improves efficiency, making it easier to build, ship, and run applications consistently across different environments. Common use cases for *Docker* include simplifying software installation, ensuring reproducibility in development, isolating services in microservice architectures, and deploying self-hosted solutions on servers or personal devices. In the context of this project, *Docker* provides a convenient way to distribute the application and its dependencies (database and *NodeJS*) in an agnostic way, enabling users to quickly deploy a fully functional instance without complex setup procedures, regardless of their environment.

[45] `https://pinia.vuejs.org/`
[46] `https://www.docker.com`

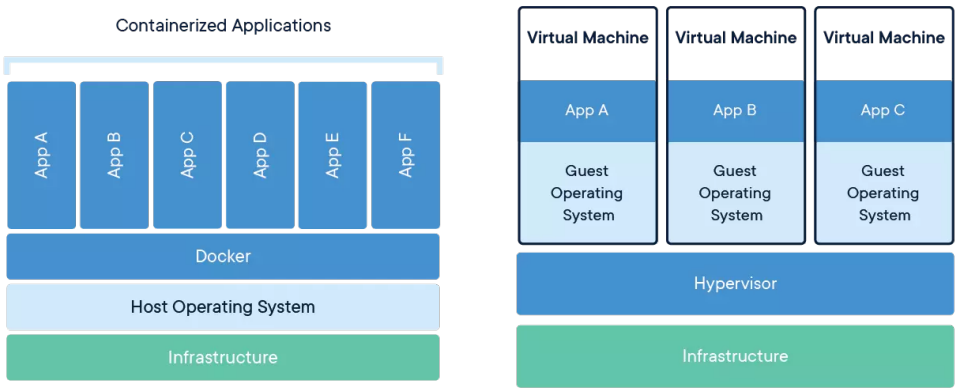


FIGURE 10. Docker vs Virtual Machines

Source: Docker Inc.

A core concept of *Docker* is the distinction between *images*, *containers*, and *volumes*. A *Docker image* is a read-only template that defines the application, its dependencies, and its environment configuration. From an *image*, *Docker* can instantiate one or more *containers*, which are the running instances of the application, each isolated but lightweight. *Containers* themselves are ephemeral by design: once removed, their internal state is lost. To address persistence needs, *Docker* provides *volumes*, which are persistent storage managed independently of *containers*. *Volumes* allow data (such as configuration files, media libraries, or databases) to persist across *container* restarts and updates, ensuring stability and continuity in long-running services.

Beyond these core concepts, *Docker* also offers an ecosystem of tools that simplify distribution and orchestration. Images can be published and retrieved from *Docker Hub*[47], a cloud-based registry containing both pre-built and custom *container images*, facilitating reuse and sharing across teams and communities. For more complex scenarios involving multiple interconnected *containers* (for example, a web server and database), *Docker Compose*[48] enables developers to define and manage multi-*container* applications through a single configuration file, improving reproducibility and maintainability of deployments.

The solution is containerized through a multi-stage *Dockerfile* shown in Listing 11 that builds both the backend and the frontend components into a single lightweight *image*. The first stage, named `backend`, uses the official `golang:1.24-alpine` image to compile the backend source code. The project files are copied into the container, and the *Go* compiler is invoked with `CGO_ENABLED=0` to produce a statically linked binary placed in the `dist` directory. This ensures that the backend service can run without external dependencies, making it highly portable.

The second stage, named `frontend`, is based on `node:22-alpine`, the official *NodeJS Docker image*. In this stage, additional tools such as `tzdata` are installed to configure time zone support, and environment variables are defined for application runtime, including

[47] `https://hub.docker.com/`
[48] `https://docs.docker.com/compose/`

the server port, base directory, and data directory. The compiled backend binary is copied from the previous stage, and port `9002` is exposed for client connections. To enable persistent data storage, a `/data` directory is created, assigned to the `node` user, and declared as a *Docker volume*. Finally, the working directory is switched to the `dist` folder, and the *container*'s entrypoint is set to execute the `cassette` binary.

This multi-stage build approach reduces the final *image* size by separating the compilation environment from the runtime environment, while ensuring that both backend and frontend components are packaged consistently in a single *container*. This design simplifies deployment, improves security by limiting unnecessary dependencies, and guarantees reproducibility across different environments.

```
# Build backend
FROM golang:1.24-alpine AS backend

WORKDIR /usr/src/cassette

COPY . .

RUN CGO_ENABLED=0 go build -o ./dist/cassette ./cmd/cassette/main.go


# Build frontend
FROM node:22-alpine AS frontend

RUN apk add --no-cache tzdata
ENV TZ="UTC"

# Use development node environment to allow dev dependencies installation for the
↪  frontend build
ENV NODE_ENV development

ENV SERVER_PORT 9002
ENV BASE_DIR /usr/src/cassette
ENV DATA_DIR /data

WORKDIR /usr/src/cassette

COPY --from=backend --chown=node:node /usr/src/cassette /usr/src/cassette

EXPOSE 9002

RUN mkdir -p /data
RUN chown -R node:node /data
VOLUME /data

```

```
34 WORKDIR /usr/src/cassette/dist
35 ENTRYPOINT ["./cassette"]
```

LISTING 11. Content of the solution's Dockerfile file

To simplify deployment and orchestration, the solution provides a sample `docker-compose.yml` file, shown at Listing 12, that defines and manages the required services as a cohesive stack. The configuration specifies two main services: `cassette` and `postgres`. The `cassette` service is built directly from the local *Dockerfile*, exposing port `9002` for client connections, and running under a non-root user for improved security. It depends on the `postgres` service, ensuring the database is available before initialization. Environment variables configure the application's runtime behavior, such as time zone, database connection *URL*, scheduler settings, and *JWT* signing key. A *volume* is mounted to persist application data between container restarts.

The `postgres` service uses the official `postgres:16-alpine` image, exposing port `5432` and configured with default user credentials (which should be swapped) and database name through environment variables. Persistent storage for the database is achieved by mapping a local directory to `/var/lib/postgresql/data` inside the *container*. By combining these services, *Docker Compose* ensures seamless integration between the application and its database, while maintaining isolation, portability, and persistence.

```
1  version: '3.6'
2
3  services:
4    cassette:
5      container_name: cassette
6      build: .
7      user: 1000:1000 # UID:GID
8      depends_on:
9        - postgres
10     ports:
11       - 9002:9002
12     environment:
13       TZ: Europe/Lisbon
14       DB_URL: postgres://root:root@postgres:5432/cassette?sslmode=disable
15       SCHEDULER_MODE: periodic
16       SCHEDULER_TIME: 02:00
17       JWT_SIGNING_KEY: jwt_signing_key
18     volumes:
19       - ./dist/docker/cassette/data:/data
20
21   postgres:
22     container_name: postgres
23     image: postgres:16-alpine
24     restart: unless-stopped
```

```
25      user: 1000:1000 # UID:GID
26      ports:
27        - 5432:5432
28      environment:
29        - POSTGRES_USER=root
30        - POSTGRES_PASSWORD=root
31        - POSTGRES_DB=default
32      volumes:
33        - ./dist/docker/postgres/data:/var/lib/postgresql/data
```

LISTING 12. Content of the solution's sample docker-compose file

3.12. Licensing

Licensing is a fundamental aspect of open-source software, as it defines the legal framework under which the software can be used, modified, and distributed. The *Open Source Initiative (OSI)*[49] and the *Free Software Foundation (FSF)*[50] play central roles in this ecosystem, each promoting software freedom through well-defined licensing standards and advocacy. While the *OSI* focuses on ensuring that licenses comply with the Open Source Definition[51], the *FSF* maintains its own classification of free software licenses that guarantee users the essential freedoms to run, study, modify, and share software. To promote consistency and interoperability across projects, the *Software Package Data Exchange (SPDX)*[52] maintains a standardized list of license identifiers recognized throughout the open-source ecosystem. By attaching an *OSI*-approved, *FSF*-recognized, or *SPDX*-registered license to a project, developers clarify the rights granted to users and contributors, ensuring transparency and protecting both parties from legal uncertainty. Licenses not only establish permissions (such as free redistribution and access to source code) but also outline obligations, such as attribution or sharing derivative works under the same terms. In this way, licensing safeguards the principles of open source while enabling collaboration, innovation, and the sustainable growth of software ecosystems.

Open-source licenses can generally be divided into three main categories: *permissive*, *copyleft*, and *weak copyleft*. Permissive licenses, such as the *MIT License* or the *Apache License 2.0*, grant users broad freedoms to use, modify, and redistribute the software, with minimal obligations beyond attribution. This flexibility makes them popular for projects that aim for widespread adoption, including integration into proprietary systems. Copyleft licenses, such as the *GNU General Public License (GPL)*, impose stronger requirements: derivative works must be distributed under the same license terms. This ensures that improvements and modifications remain open-source, preserving software freedom across subsequent versions (Free Software Foundation, Inc., 2022a). Weak copyleft licenses, such as the *GNU Lesser General Public License (LGPL)* or the *Mozilla Public License (MPL)*,

[49] https://opensource.org/
[50] https://www.fsf.org/
[51] https://opensource.org/osd
[52] https://spdx.org/licenses/

take a middle-ground approach by requiring modifications to the licensed components themselves to remain open-source, while allowing them to be combined with proprietary code. This balance makes weak copyleft licenses particularly suitable for libraries or components intended to be reused in broader software ecosystems (Sawers, 2025).

To assist in selecting an appropriate license, the websites *Choose an open source license*[53] maintained by *GitHub* and *How to Choose a License for Your Own Work*[54] supported by the *FSF* were consulted. Since the present solution seeks to preserve its open-source nature and foster community contribution, a *copyleft* license was identified as the most appropriate choice. Given that the solution functions as a web service accessible over a network, the *GNU Affero General Public License (AGPL)* was selected. The *AGPL* extends the principles of the *GPL* by requiring that any modified version deployed as a network service (such as a web application) must also make its source code available to users who interact with it (Free Software Foundation, Inc., 2022b). Although the *Open Software License (OSL)* was also considered, the *GNU Project* advises against its use, as recent versions demand explicit assent from users before applying the license (Free Software Foundation, Inc., 2025). By closing the loopholes of traditional copyleft licenses, the *AGPL* ensures that software freedoms are maintained even in cloud-based and networked environments, making it particularly well suited for this solution.

3.13. Contributions

In the spirit of the open-source community, during the development of this solution, contributions were made to several third-party projects that were utilized. These contributions included bug fixes, feature enhancements, and documentation improvements, all aimed at enhancing the functionality and usability of the libraries and tools that supported the project. By actively participating in these communities, not only was the quality of the solution improved, but it also fostered collaboration and knowledge sharing among developers. This reciprocal relationship underscores the importance of contributing back to the open-source ecosystem, ensuring that others can benefit from the improvements made and encouraging a culture of continuous improvement and innovation.

Platforms like *GitHub* and *GitLab*[55] provide *Issues* and *Pull Requests* (or *Merge Requests* in *GitLab*) that play a crucial role in this collaborative process. When bugs or areas for improvement are identified in third-party projects, *Issues* are created to document and discuss these problems with the wider community. *Pull Requests* or *Merge Requests* are then submitted to propose fixes or enhancements, allowing maintainers and contributors to review and integrate changes. This workflow not only streamlines the resolution of bugs and implementation of new features, but also encourages transparent communication and shared responsibility among developers. By engaging with *Issues* and *Pull Requests* or

[53] `https://choosealicense.com/`
[54] `https://www.gnu.org/licenses/license-recommendations.en.html`
[55] `https://about.gitlab.com/`

Merge Requests, contributors help maintain and improve the quality of open-source libraries, further strengthening the ecosystem. A summary of the ten contributions made across five distinct repositories is provided in Table 3.

Date	**Repository**	**Contribution Type**	**URL**
2024-01-21	chazeon/olgitbridge	Issue	#1
2024-02-04	chazeon/olgitbridge	Pull Request	#2
2024-03-31	ThomasLeconte/vuetify3-dialog	Pull Request	#15
2024-04-24	ThomasLeconte/vuetify3-dialog	Issue	#16
2024-04-28	ThomasLeconte/vuetify3-dialog	Pull Request	#18
2024-06-28	nerdocs/vue-extensions	Issue	#42
2024-06-28	nerdocs/vue-extensions	Pull Request	#43
2024-07-07	nerdocs/vue-extensions	Pull Request	#44
2024-09-01	ThomasLeconte/vuetify3-dialog	Pull Request	#25
2025-03-07	vuetifyjs/vuetify	Pull Request	#21078
2025-03-09	grpc-ecosystem/grpc-gateway	Pull Request	#5339

TABLE 3. Contributions made to third-party open-source projects

As shown in Table 3, the contributions started with the openning of an *Issue* in the `chazeon/olgitbridge` repository, which is a tool used to bridge the online *LaTex* editor *Overleaf* with the *Git* version control system. The *Issue* reported a problem encountered when using the bridge to push local changes to *Overleaf*, which was subsequently addressed by submitting a *Pull Request* that provided a fix. This contribution not only resolved the specific *Issue* but also improved the overall reliability of the tool for other users.

Contributions were also made to the `ThomasLeconte/vuetify3-dialog` repository, a library that extends the *Vuetify* framework to enable dynamic creation of dialogs and toasts (also known as Snackbars) components. An *Issue* was first reported to inform the maintainers of a breaking change introduced by recent updates to the *Vuetify* framework, which prevented the library from functioning correctly. This notification allowed the maintainers to promptly address the problem and release a fix. Followed by contributions in the form of three *Pull Requests*: the first to enhance the library by adding support for custom icons in dialogs, the second to fix default settings taking precedence over user-defined options and the third to fix the placement and stacking logic of the Snackbar component. These contributions not only resolved existing *Issues* but also added valuable features that improved the library's usability and flexibility.

During the development of the frontend plugin system and integration of the `nerdocs/vue-extensions` library, an *Issue* was discovered where the library failed to load because the main module path was incorrectly specified in its `package.json` file, preventing successful loading. To resolve this, a *Pull Request* was submitted to correct the path and enable proper usage of the library. Additionally, another *Pull Request* improved

the library's robustness by preventing errors when the plugin hook is unused, thereby enhancing the overall user experience.

In order to address a misleading statement in the documentation of the *Vuetify framework* regarding whether `v-checkbox-btn` respects the `v-data-table` header alignment, a *Pull Request* was submitted to `vuetifyjs/vuetify`. This clarified the documentation, ensuring users have accurate information about the component's behavior.

Finally, a *Pull Request* was submitted to the `grpc-ecosystem/grpc-gateway` repository (mentioned in subsection 3.5.3) to fix a previously reported bug where the user-defined service name was not correctly used when generating the *OpenAPI* specification for *Swagger UI*. This contribution improved the reliability and usability of the `gRPC-Gateway` library, benefiting developers who rely on it to build *gRPC*-based services with *RESTful* interfaces.

[This page has been intentionally left blank]

CHAPTER 4

Conclusions

The development of the solution successfully fulfilled the objectives defined at the start of the project, addressing the limitations identified in existing podcast clients while leveraging the strengths of the open-source community. The resulting system provides a free, self-hosted podcast library management platform that emphasizes user control, privacy, and long-term adaptability. All core features established during the planning phase were implemented, including support for multiple user accounts, podcast subscriptions, automatic episode downloads, and synchronization of playback progress across devices. The solution also delivers an ad-free listening experience accessible directly through the browser, ensuring broad usability without requiring additional software. Furthermore, its extensible architecture, incorporating plugin systems for both backend and frontend components, enables customization and future expansion. Overall, the project demonstrates the viability of an open-source alternative to proprietary podcast management platforms.

While the developed solution meets its initial objectives, several limitations remain that present valuable opportunities for future improvement. One notable enhancement would be the integration of a podcast search feature directly within the application, enabling users to discover and subscribe to new shows without relying on external directories. Additionally, the current plugin system requires a restart after installation, a limitation that could be addressed by introducing dynamic loading to improve usability. Extending compatibility with mobile clients by integrating with apps that support the *GPodder* sync *API* or supporting offline listening, would further expand accessibility. Other areas for enhancement include internationalization to support multiple languages and interface customization through themes, improving inclusivity and personalization for a broader audience.

Beyond these core improvements, several advanced features could be inspired by modern media players and streaming platforms. These include smart playback utilities such as automatic skipping of intros or silences, listening analytics to provide usage insights, user notifications, and a configurable sleep timer for timed playback control. Implementing support for federated communication between instances would also promote decentralization and content sharing within the open-source ecosystem. Finally, enabling media casting through the web application, such as via *Chromecast*, would provide a seamless and device-agnostic listening experience. Collectively, these extensions would elevate the platform from a robust self-hosted system to a feature-rich, community-driven alternative to commercial podcast services.

[This page has been intentionally left blank]

References

AUTH0, 2018. *JSON Web Token Introduction - Jwt.Io* [online]. 2018-06-14 [visited on 2025-07-29]. Available from: https://jwt.io/introduction.

AUTH0, 2021. *Hashing in Action: Understanding Bcrypt* [online]. 2021-02-25 [visited on 2025-07-29]. Available from: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/.

BOTTOMLEY, Andrew J., 2015. Podcasting: A Decade in the Life of a "New" Audio Medium: Introduction. *Journal of Radio & Audio Media* [online]. Vol. 22, no. 2, pp. 164–169 [visited on 2025-09-19]. ISSN 1937-6529. Available from DOI: 10.1080/19376529.2015.1082880.

BOWERS, Andy, 2005. The Year of the Podcast. *Slate* [online] [visited on 2025-09-21]. ISSN 1091-2339. Available from: https://slate.com/news-and-politics/2005/12/the-year-of-the-podcast.html.

EDISON RESEARCH, 2023a. *The Infinite Dial 2023 from Edison Research with Amazon Music, Wondery, and ART19* [online]. 2023-03-03 [visited on 2024-02-21]. Available from: https://www.edisonresearch.com/infinite-dial-2023-from-edison-research-with-amazon-music-wondery-and-art19/.

EDISON RESEARCH, 2023b. *The Podcast Consumer 2023: An Infinite Dial Report* [online]. 2023-03-09 [visited on 2024-02-18]. Available from: https://www.edisonresearch.com/the-podcast-consumer-2023-an-infinite-dial-report/.

ERIC ANDERSON, 2024. *Interceptors* [online]. 2024-02-29 [visited on 2025-08-05]. Available from: https://grpc.io/docs/guides/interceptors/.

EVANS, Chris, 2008. The Effectiveness of M-Learning in the Form of Podcast Revision Lectures in Higher Education. *Computers & Education* [online]. Vol. 50, no. 2, pp. 491–498 [visited on 2025-09-07]. ISSN 0360-1315. Available from DOI: 10.1016/j.compedu.2007.09.016.

FERNANDEZ, Vicenc; SIMO, Pep; SALLAN, Jose M., 2009. Podcasting: A New Technological Tool to Facilitate Good Practice in Higher Education. *Computers & Education* [online]. Vol. 53, no. 2, pp. 385–392 [visited on 2025-09-07]. ISSN 0360-1315. Available from DOI: 10.1016/j.compedu.2009.02.014.

FREE SOFTWARE FOUNDATION, INC., 2022a. *What Is Copyleft? - GNU Project - Free Software Foundation* [online]. 2022-01-02 [visited on 2025-10-06]. Available from: https://www.gnu.org/licenses/copyleft.en.html.

FREE SOFTWARE FOUNDATION, INC., 2022b. *Why the GNU Affero GPL - GNU Project - Free Software Foundation* [online]. 2022-01-02 [visited on 2025-10-06]. Available from: https://www.gnu.org/licenses/why-affero-gpl.en.html.

FREE SOFTWARE FOUNDATION, INC., 2025. *Various Licenses and Comments about Them - GNU Project - Free Software Foundation* [online]. 2025-08-13 [visited on 2025-10-06]. Available from: https://www.gnu.org/licenses/license-list.html#OSL.

FRIZZELL, Nell, 2016. 'I Felt like Morse Tapping His First Code' – the Man Who Invented the Podcast. *The Guardian: Television & radio* [online] [visited on 2025-09-19]. ISSN 0261-3077. Available from: https://www.theguardian.com/tv-and-radio/2016/nov/03/christopher-lydon-podcast-inventor-open-source-mp3-files-interview.

GITHUB, INC., 2020. *Saving Repositories with Stars* [online] [visited on 2024-03-09]. Available from: https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars.

GOOGLE LLC, 2022. *Encoding* [online] [visited on 2025-07-20]. Available from: https://protobuf.dev/programming-guides/encoding/.

GRÖBER, Lea; MROWCZYNSKI, Rafael; VIJAY, Nimisha; MULLER, Daphne A.; DABROWSKI, Adrian; KROMBHOLZ, Katharina, 2023. To Cloud or Not to Cloud: A Qualitative Study on Self-Hosters' Motivation, Operation, and Security Mindset. In: [online], pp. 2491–2508 [visited on 2025-12-21]. ISBN 978-1-939133-37-3. Available from: https://www.usenix.org/conference/usenixsecurity23/presentation/grober.

HAMMERSLEY, Ben, 2004. Audible Revolution. *The Guardian: Media* [online] [visited on 2025-09-16]. ISSN 0261-3077. Available from: https://www.theguardian.com/media/2004/feb/12/broadcasting.digitalmedia.

HOME ASSISTANT, 2025. *Creating Your First Integration | Home Assistant Developer Docs* [online]. 2025-03-20 [visited on 2025-10-27]. Available from: https://developers.home-assistant.io/docs/creating_component_index.

JOHAN BRANDHORST-SATZKORN, 2024. *Runtime Package - Github.Com/Grpc-Ecosystem/Grpc-Gateway/v2/Runtime - Go Packages* [online]. 2024-05-23 [visited on 2025-08-05]. Available from: https://pkg.go.dev/github.com/grpc-ecosystem/grpc-gateway/v2/runtime#Middleware.

JONES, Michael B.; BRADLEY, John; SAKIMURA, Nat, 2015. *JSON Web Token (JWT)* [online]. 2015-05 [visited on 2025-07-29]. Request for Comments, RFC 7519. Internet Engineering Task Force. Available from DOI: 10.17487/RFC7519.

JULIA WILSON; OLGA DIACHKOVA; REBECCA FLOYD, 2025. *2025 Docker State of App Dev: Key Insights Revealed* [online]. 2025-07-10 [visited on 2025-12-21]. Available from: https://www.docker.com/blog/2025-docker-state-of-app-dev/.

KUCHTA, Martin, 2001. Audio on the Internet: History and Evolution of Podcasts. In: *ResearchGate* [online] [visited on 2025-09-14]. Available from: https://www.

researchgate.net/publication/358022724_Audio_on_the_Internet_History_and_Evolution_of_Podcasts.

MICROSOFT, 2025. *Extension API* [online]. 2025-09-10 [visited on 2025-10-27]. Available from: https://code.visualstudio.com/api/index.

MIKOWSKI, Michael; POWELL, Josh, 2013. *Single Page Web Applications: JavaScript End-to-End*. Simon and Schuster. ISBN 978-1-63835-134-4. Available from Google Books: eDszEAAAQBAJ.

MORRIS, Jeremy Wade; PATTERSON, Eleanor, 2015. Podcasting and Its Apps: Software, Sound, and the Interfaces of Digital Audio. *Journal of Radio & Audio Media* [online]. Vol. 22, no. 2, pp. 220–230 [visited on 2024-02-18]. ISSN 1937-6529. Available from DOI: 10.1080/19376529.2015.1083374.

MOZILLA CORP, 2025. *Cross-Origin Resource Sharing (CORS) - HTTP | MDN* [online]. 2025-07-07 [visited on 2025-08-05]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS.

NEWMAN, Julliana; LIEW, Andrew; BOWLES, Jon; SOADY, Kelly; INGLIS, Steven, 2021. Podcasts for the Delivery of Medical Education and Remote Learning. *Journal of Medical Internet Research* [online]. Vol. 23, no. 8, e29168 [visited on 2025-09-07]. Available from DOI: 10.2196/29168.

OXFORD, 2024. *Oxford Word of the Year: Defining the Past 20 Years* [online]. 2024-10-01 [visited on 2025-09-21]. Available from: https://corp.oup.com/feature/oxford-word-of-the-year-defining-the-past-20-years/.

PEYROTT, SEBASTIAN E; AUTH0, 2018. *The JWT Handbook*.

PLEX INC., 2022. *Podcasts and Web Shows Are Going Away on April 15, 2022 - Announcements* [online]. 2022-04-08 [visited on 2024-03-03]. Available from: https://forums.plex.tv/t/podcasts-and-web-shows-are-going-away-on-april-15-2022/787096.

RSS ADVISORY BOARD, 2009. *RSS 2.0 Specification (Current)* [online]. 2009-03-30 [visited on 2025-09-22]. Available from: https://www.rssboard.org/rss-specification.

SAWERS, Paul, 2025. *Open Source Licenses: Everything You Need to Know* [online]. 2025-01-12 [visited on 2025-10-02]. Available from: https://techcrunch.com/2025/01/12/open-source-licenses-everything-you-need-to-know/.

SHOPIFY, 2025. *WebAssembly for Functions* [online] [visited on 2025-10-27]. Available from: https://shopify.dev/docs/apps/build/functions/programming-languages/webassembly-for-functions.

TITTERINGTON, Alanna, 2025. *WordPress: Vulnerabilities in Plugins and Themes* [online]. 2025-08-29 [visited on 2025-10-27]. Available from: https://www.kaspersky.com/blog/vulnerable-wordpress-plugins-and-themes/54228/.

USZKAY, Duncan, 2020. *How Shopify Uses WebAssembly Outside of the Browser* [online]. 2020-12-18 [visited on 2025-10-27]. Available from: https://shopify.engineering/shopify-webassembly.

VERGHOST, 2022. *VLC Lua Docs* [online] [visited on 2025-10-27]. Available from: https://vlc.verg.ca/.

VLC, 2025. *VLC: VLC* [online] [visited on 2025-10-27]. Available from: https://videolan.videolan.me/vlc/index.html.

WINER, Dave, 2000a. *DaveNet : Virtual Bandwidth* [online]. 2000-10-30 [visited on 2025-09-21]. Available from: http://scripting.com/davenet/2000/10/31/virtualBandwidth.html.

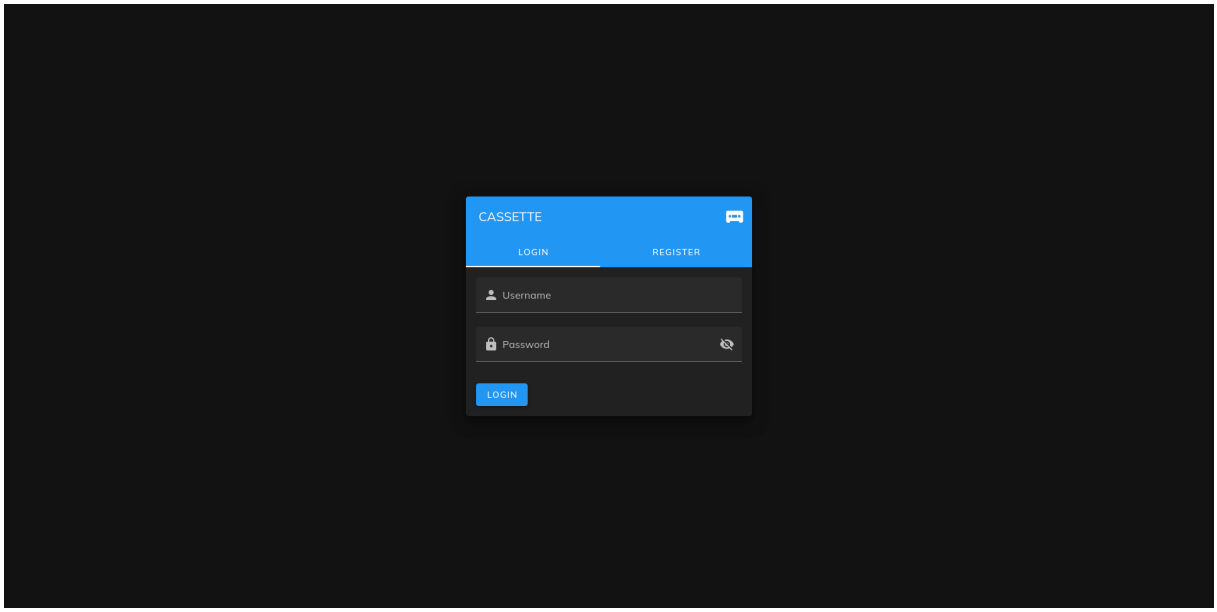
WINER, Dave, 2000b. *RSS 0.92* [online]. 2000-12-25 [visited on 2025-09-21]. Available from: http://backend.userland.com/discuss/msgReader$110.

WINER, Dave, 2003. *RSS 2.0 Specification (RSS 2.0 at Harvard Law)* [online]. 2003-07-15 [visited on 2025-09-21]. Available from: https://cyber.harvard.edu/rss/rss.html.

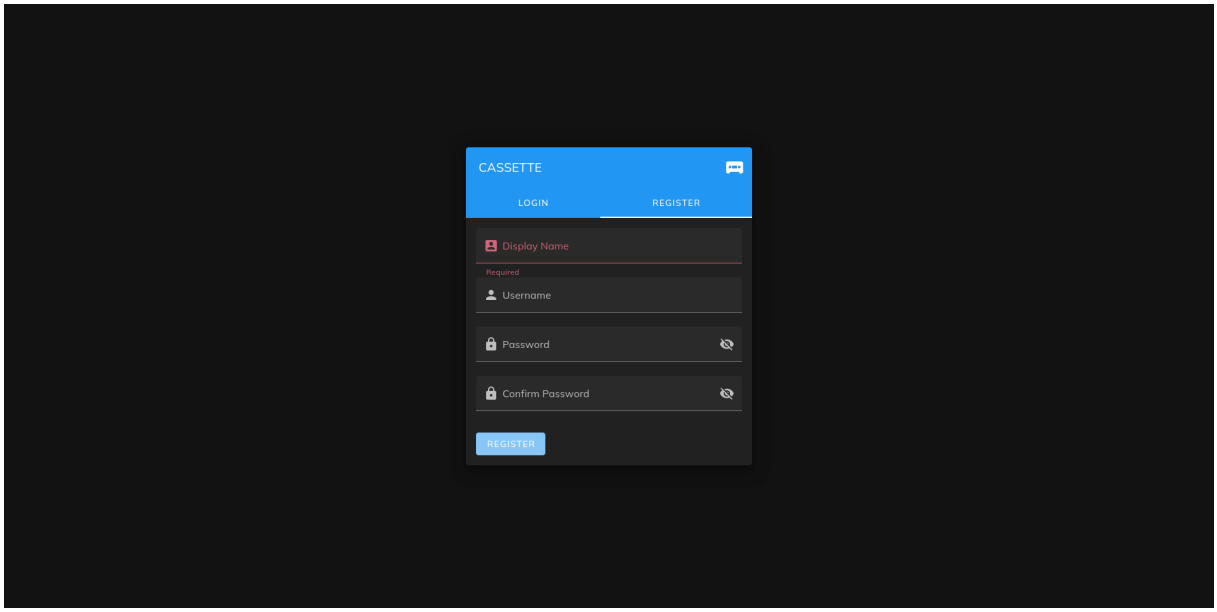
WORDPRESS, 2023. *Introduction to Plugin Development – Plugin Handbook | Developer.WordPress.Org* [online]. 2023-12-14 [visited on 2025-10-27]. Available from: https://developer.wordpress.org/.

Appendices

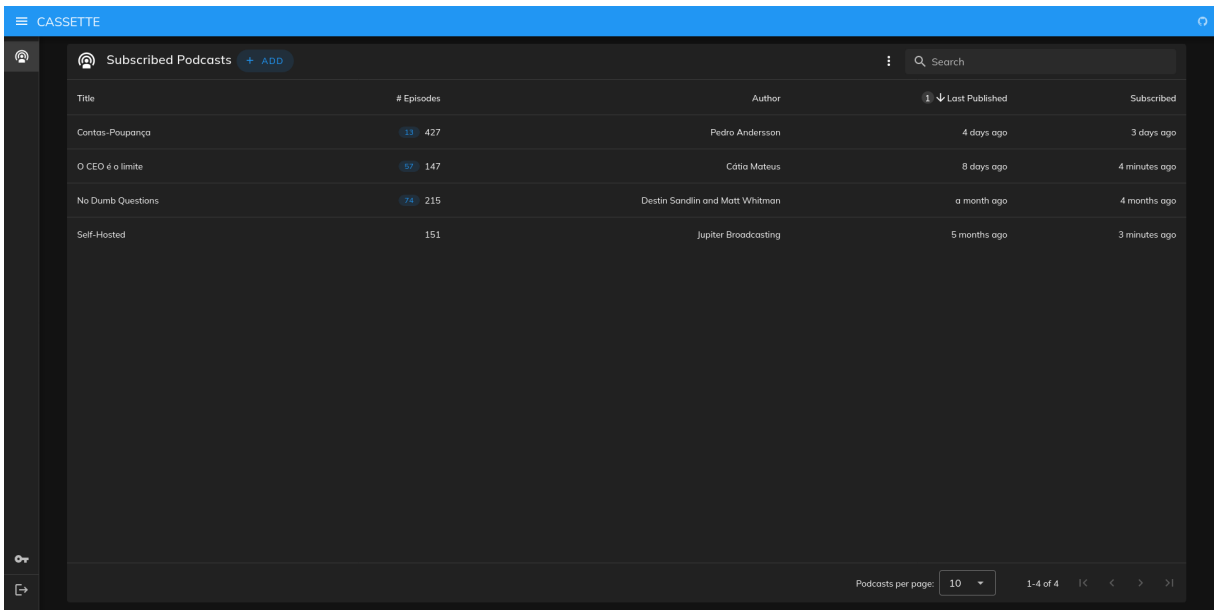
Screenshots of the Software



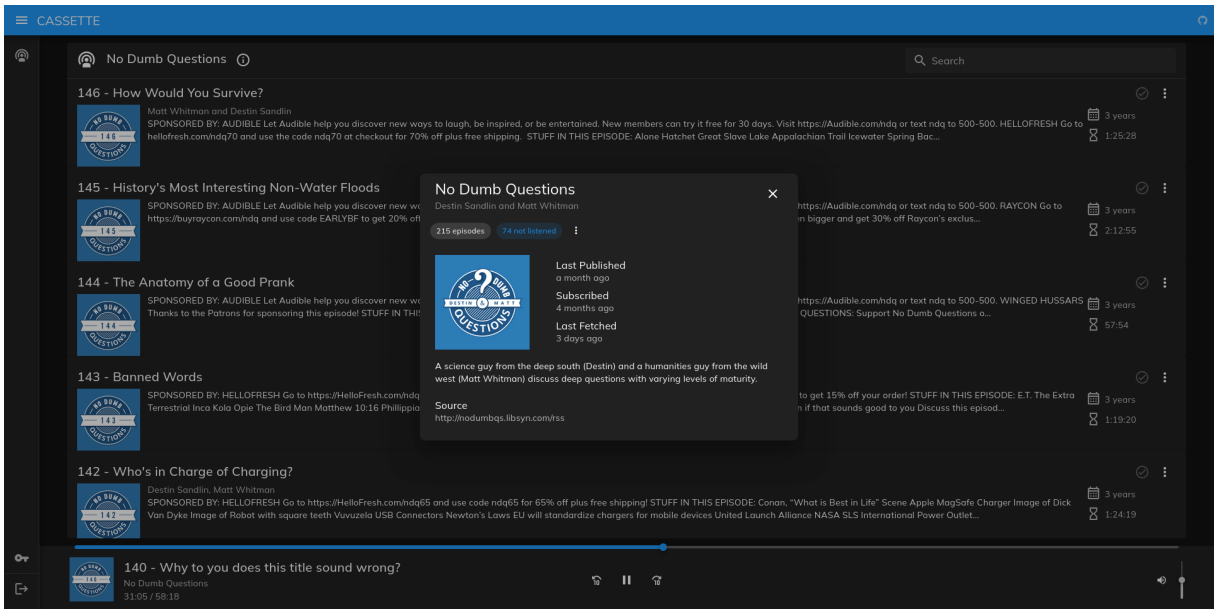
Screenshot of login page



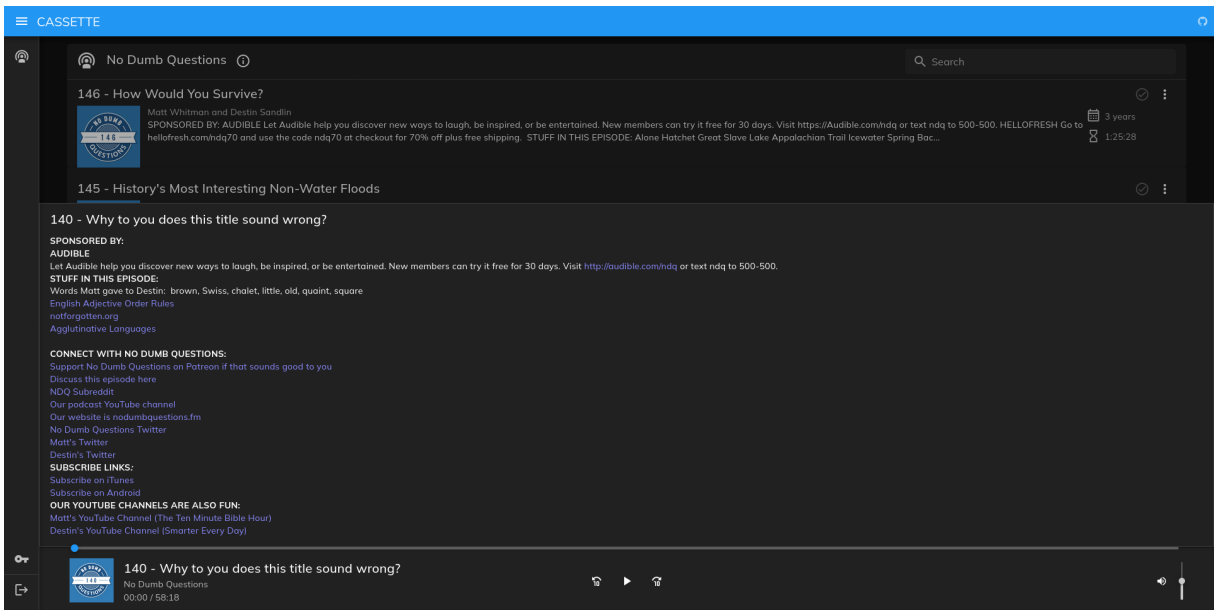
Screenshot of register page



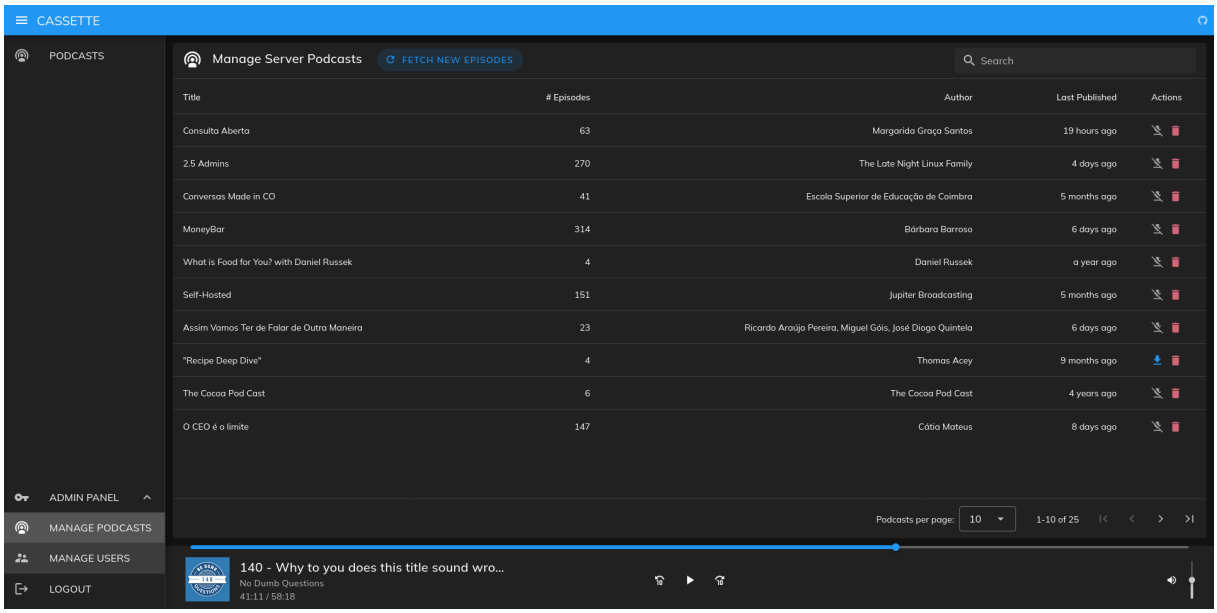
Screenshot of subscribed podcasts page



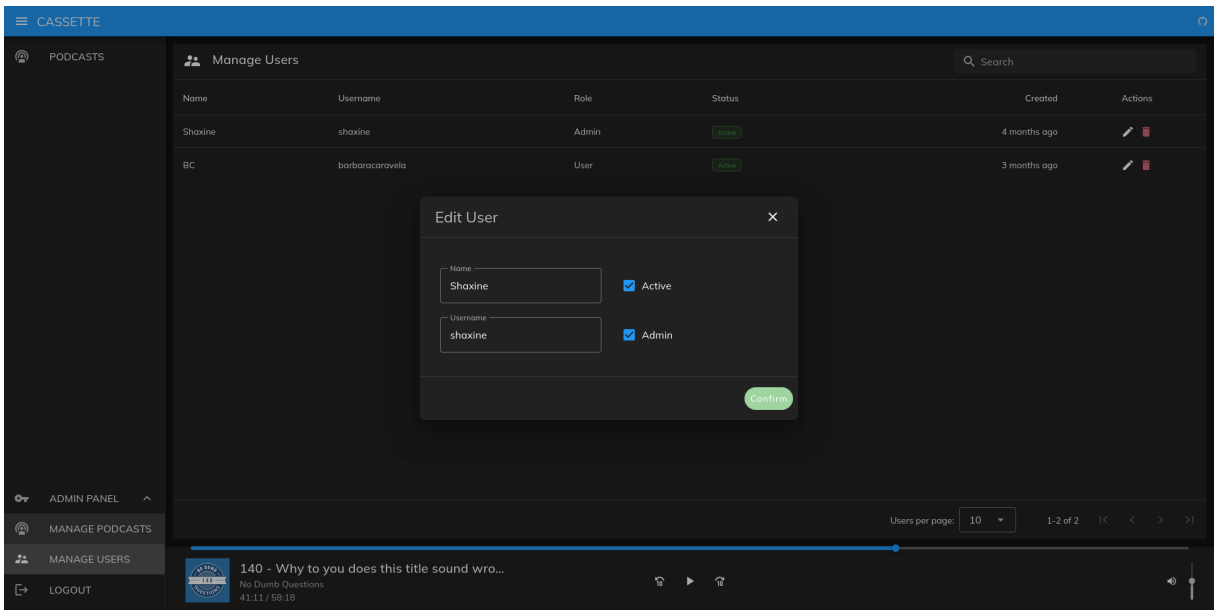
Screenshot of podcast details page



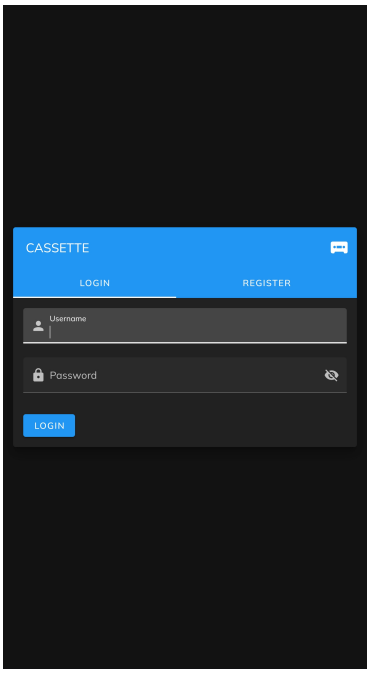
Screenshot of episode details in podcast details page



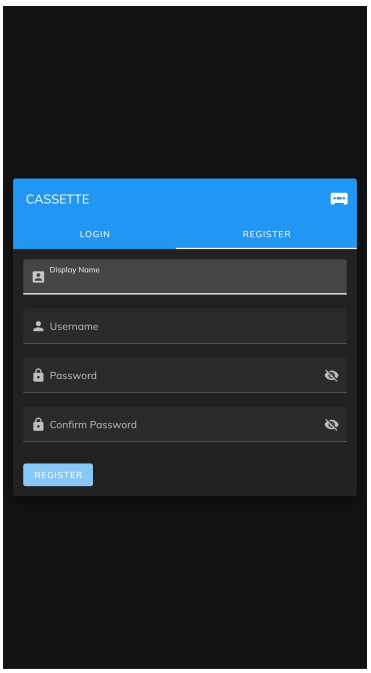
Screenshot of administator podcasts page



Screenshot of administator users page

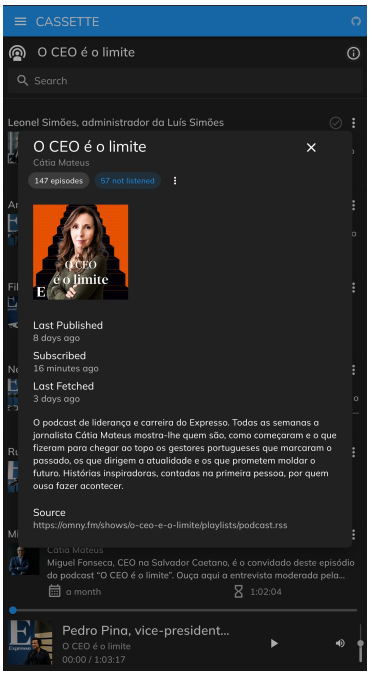


Screenshot of login page on a mobile device

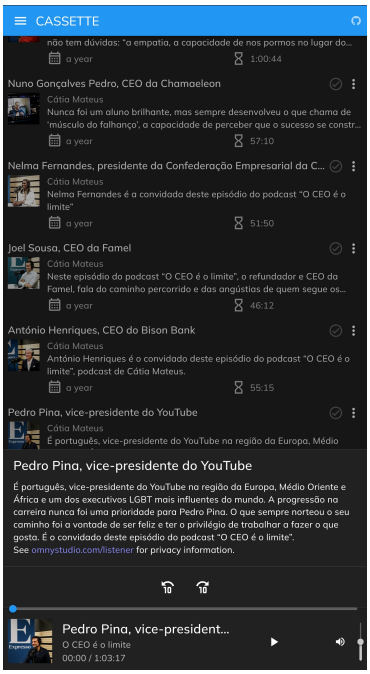


Screenshot of register page on a mobile device

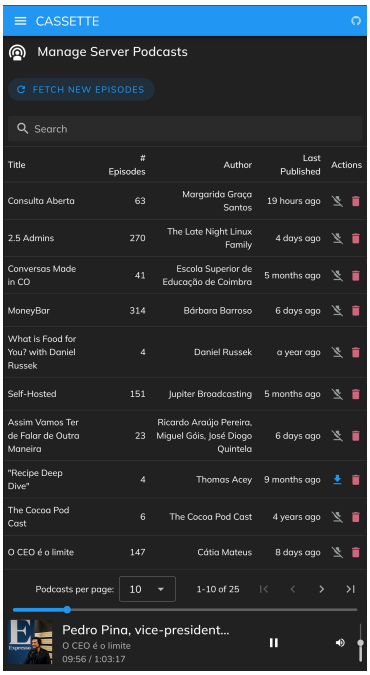
Screenshot of subscribed podcasts page on a mobile device



Screenshot of podcast details page on a mobile device



Screenshot of episode details in podcast details page on a mobile device



Screenshot of administator podcasts page on a mobile device