UNSW
SYDNEY

School of Computer Science and Engineering
Faculty of Engineering
UNSW Sydney

UltraCast

BY

Daniel Latimer	Connor O'Shea	Kevin Chan	Oliver Richards	Peter Kerr
z5115175	z5115177	z5113136	z5157383	z5115807

Submitted: November 12, 2020
Tutor: Tatjana Zrimec

Contents

1 Overview

1.1 Introduction

With over 100 million monthly listeners[1] and a steadily increasing user base, there is no doubt that podcasts are a greatly enriching source of information and entertainment for a large variety of individuals.

Although they are highly valuable, it can be difficult to find podcasts that are of interest to a particular user amidst the 1 million[1] that are already available. Thus, podcast streaming services (such as UltraCast) have been created, to provide a centralised place for exploring and discovering new podcasts that are valuable to the listener.

However, all of the web based podcast streaming services available lack many important features, and their interfaces leave much to be desired. For example, there is no streaming service that allows the user to bookmark certain parts of a podcast, nor take notes at certain timestamps. It is even difficult to find a service that allows the listener to change the playback speed of the podcast.

UltraCast combines all of the most important features together into a single package with a web-based podcast streaming service.

UltraCast differentiates itself from competitors by allowing users to:

- Follow friends to see what they have been listening to
- Create *Streams* of podcasts to find interesting podcasts
- Create bookmarks inside podcast episodes
- Monitor episode and podcast play metrics

1.2 Project Requirements

The minimun project requirements from the specifications are:

- Listeners must be able to search for podcasts that interest them by keywords, resulting in a list of matching podcast titles, where the total number of subscriptions on the UltraCast platform (function described later) for each podcast is shown next to the title
- Listeners must be able to select a podcast show from returned search results to view its full details, including its title, description, any author details that exist, as well as a list of episodes for the show
- Listeners must be able to play a selected episode within a podcast show, and once that episode starts being played, the listener must be able to also clearly see this episode marked as "Played"
- Listeners must be able to subscribe or unsubscribe from a podcast show Listeners must be able to see the latest episode available for each show that they subscribed to in a "Podcast Subscription Preview" panel
- Listeners must be notified by the platform when a new episode for a show they are subscribed appears

- Listeners must be able to see a history of the podcast episodes that they have played, sorted in order from most recently played to least recently played
- UltraCast must be able to recommend new podcast shows to a listener based on at least information about the podcast shows they are subscribed to, podcast episodes they have recently played, and their past podcast searches

The following additional requirements have also been implemented:

- Listeners should be able to follow their *friends* and view the podcasts that their *friends* have recently listened to
- Listeners should be able to create *Streams* based off search queries that they can use to find interesting podcasts
- Listeners should be able to add bookmarks with a name and description to podcast episodes as they listen to them
- Content creators should be able to create and upload podcasts and podcast episodes
- Content creators should be able to monitor analytics of their uploaded podcasts related to their listeners

1.3 System Architecture

2 Functionalities and Implementation Challenges

2.1 Functionalities

2.2 Implementation Challenges

2.2.1 Backend Stack

The backend of UltraCast employs an unusual technology stack, with MongoDB as a persistence layer, flask as a webserver framework and graphql (via graphene and graphene-mongo libraries) as an API layer. This created difficulties in implementing common web-app functionalities due to (1) a lack of documentation on the libraries being used and (2) no online examples implementing these functionalities with this stack.

User Authentication Implementing user authentication for the backend was a non-trivial task because the Graphene and Graphene-Mongo libraries which are used for the API layer do not natively support this functionality. A major challenge in applying general purpose authentication libraries, for example flask-jwt[1], is that only one route is used for all API calls. Some of these API calls need to be authenticated e.g. deleting a podcast where others should not be e.g. signing up to the site. The Flask-GraphQL-Auth library[2] provides the required authentication methods, however, it is not actively maintained. After much research, user authentication was implemented using the flask-jwt-extended library[3]. This library allows authentication to be required on a per-function level, rather

[1] Available at https://github.com/mattupstate/flask-jwt
[2] Available at https://github.com/NovemberOscar/Flask-GraphQL-Auth
[3] Available at https://github.com/vimalloc/flask-jwt-extended

than for an entire route. Hence, certain mutations and queries can be protected with user authentication where required. The frontend calls a signin mutation which returns a Json Web Token (JWT). This mutation does not require authentication. The frontend then stores this JWT as a cookie and sends it in the header of any future GraphQL API requests.

Resolving Nested Queries While testing the frontend, it was discovered that some backend GraphQL queries were taking upwards of one minute to return. The site was still responsive, however it took a long time for recommended podcasts to be displayed. Further investigation revealed that where nested references were used in the database models, and the GraphQL query involved dereferencing these references, the Graphene-Mongo library would perform one database operation per parent node. These database operations are performed sequentially. Since the MongoDB instance is hosted in the cloud, each database operation takes some number of milliseconds due to network latency. When a large number of parent nodes were fetched, this resulted in very slow queries. It was not feasible to modify the Graphene-Mongo libary to issue less database operations. Hence, the decision was made to move the GraphQL API webserver to the same cloud container as the MongoDB instance. This improved the time for some queries from over fourty seconds to less than a second.

3 User Manual

3.1 Software Setup Instructions

3.2 Site Usage and Functionality Guide

References

[1] Gavin Whitner. Podcast statistics (2020). https://musicoomph.com/podcast-statistics/, Sep 2020. Accessed: 2020-10-02.