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 15, 2020
Tutor: Tatjana Zrimec

Contents

1 Overview

1.1 Introduction

With over 100 million monthly listeners[6] 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[6] 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

The highlevel system architecture can be seen in Figure 1. The end users, podcast listeners and content creators, connect to the presentation layer which is powered by a ReactJS application. ReactJS was selected as the framework for the frontend application primaly due to it being a mature and world leading framework[4] as well as due to previous experience with the framework.

Flask, a python based micro-framework[2], was used for the web-server, due ease of use which allowed for rapid development. The React application communicates with Flask web-server through a GraphQL API: a scalable alternative to the popular REST API[3]. The Flask web-server also contains a recommendation service which drives recommendation functionality described in Sections 2.1.3 and 2.2. The Flask web-server uploads static files (episode audio and podcast covers) to a static file storage server while the urls for these static files along with all other data is stored in MongoDB. MongoDB, a NoSQL database, was selected to store metadata due to its scalability[5]. MongoDB could not be hosted on CSE so was hosted on AWS EC2 instead, see 1 for reason. As a result the Flask web-server needed to be on the the remote server as well to improve performance as described in 2.3.1. Algolia, a search engine service[1], was used to maximise search performance and reduce development time.

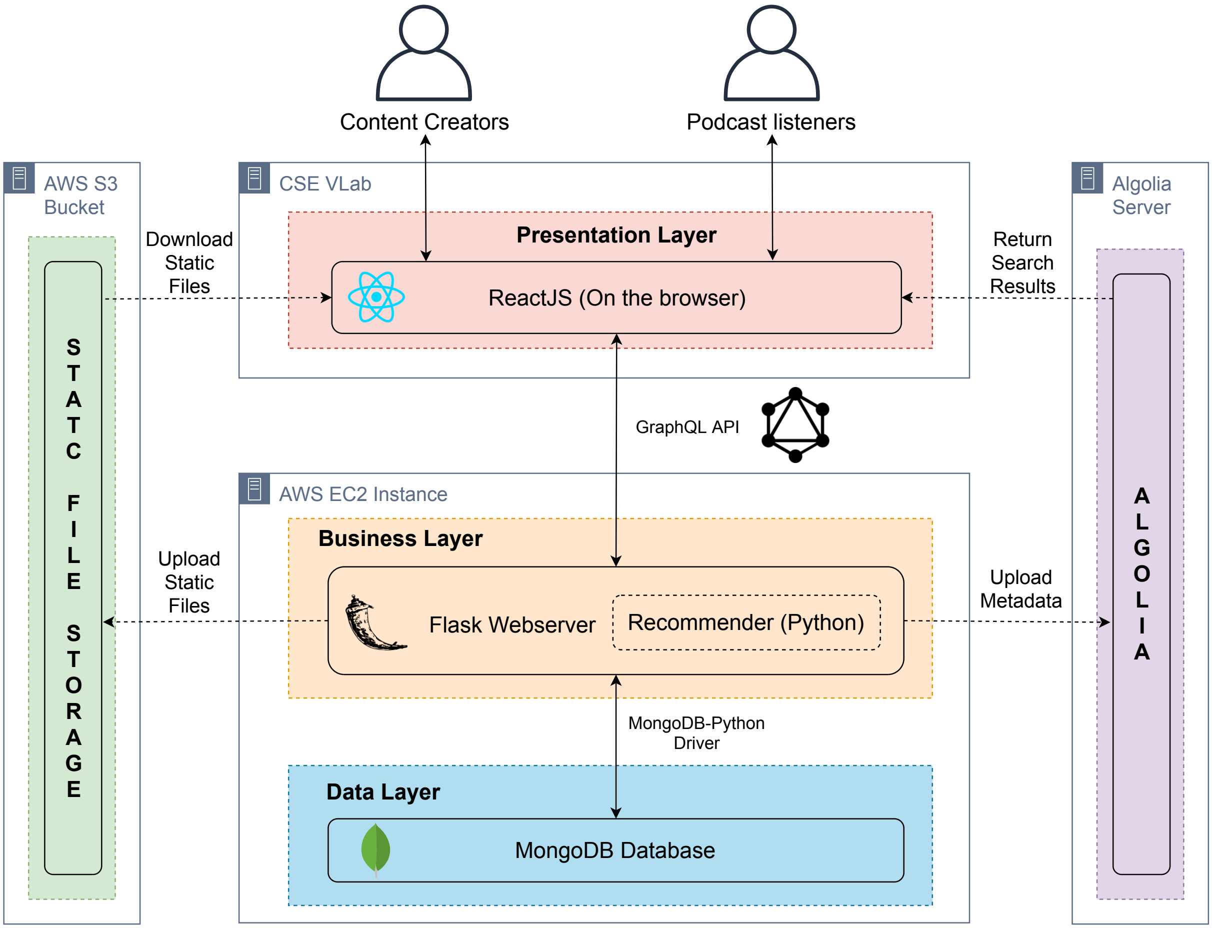

Figure 1: UltraCast System Architecture

1.3.1 Third Party Components

The third party components which had a notable impact on the functionality of UltraCast are shown in Table 1. This is not an exhaustive list of all the libraries and system libraries in UltraCast but rather a list of components which were notable and impacted functionality.

Table 1: Third Party Components

Name	Component Type	Reason For Use
AWS S3	Cloud: Static File Storage	Exceeded CSE server storage limits. >80GB of test data
AWS EC2	Cloud: Remote Server	MongoDB not supported on Debian 6 (Linux environment on the VLab machine)
Algolia	SaaS: Search Engine	Avoid building search engine from scratch
React	Frontend Framework	Bootstrap and provide basic functionality
Flask	Micro Framework	Bootstrap and provide basic functionality
Graphene	Python GraphQL Library	Standard library
Mongoengine	Python object data mapper for MongoDB	Standard library
Pandas	Python data Library	Standard library
NumPy	Python maths Library	Standard library

2 Functionalities and Implementation Challenges

2.1 Functionalities

The subsections below describe the functionalities of UltraCast broken down into 5 major functionality groupings. Each functionality is directly connected to a User Story, defined in the Propsal, as well as a project objective if it addresses it. The colour schema defined in Table 2 was used for functionalities in Tables 3 to 7.

Table 2: Functionality Table Colour Schema

Colour	Meaning
🟨	Functionality relates to project objective
⬜	Functionality not specified in project objective
🟦	As above, but functionality is novel

2.1.1 Viewing and Searching

The viewing and searching functionalities are described in Table 3 below.

Table 3: Viewing and Searching Functionality Mapping

Story ID	Functionality	Project Objective
UL-2	Use keywords to search for podcasts, return list of podcasts (See UL-4 for format)	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
UL-3	View the total number of subscribers for each podcast returned from a search	
UL-4	View the title, description, author details and list of episodes for a podcast	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
UL-14	Login as specific user	-
UL-24	View a title, length, upload date for episodes	-
UL-29	Save search as a ”Stream”	-
UL-41	Signup as user	-

2.1.2 Playing Podcast Episode

The playing podcast episode functionalities are described in Table 4 below.

Table 4: Playing Episode Functionality Mapping

Story ID	Functionality	Project Objective
UL-5	Play episodes	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”
UL-6	Once episode starts being played it is marked as played	
UL-18	Pause episode that is playing	-
UL-19	Adjust playback volume	-
UL-20	Skip to next episode, previous episode and start of current episode	-
UL-21	Jump to a point in an episode	-
UL-22	Adjust playback speed	-
UL-23	Auto-play episodes in a podcast (after added to playlist)	-
UL-26	”Bookmark” a point in an episode with a title and description	-

2.1.3 Recommendation and Following

The recommendation and following functionalities are described in Table 5 below.

Table 5: Recommendation and Following Functionality Mapping

Story ID	Functionality	Project Objective
UL-10 UL-11	View episode history Episode history is sorted by most recent to least recent	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
UL-12 UL-13	Podcast recommendations are based on: Existing subscriptions recently played episodes and past searches A "recommended" panel shows recommended podcasts	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
UL-18	Follow users, view their listen history	-

2.1.4 Creator Mode

The creator functionalities are described in Table 6 below.

Table 6: Creator Mode Functionality Mapping

Story ID	Functionality	Project Objective
UL-15 UL-16	Create podcasts and episodes Delete podcasts and episodes	-
UL-17	Update podcasts and episodes	-
UL-27	Access to podcast and episode viewer metrics	-

2.1.5 Subscribe

The subscribe functionalities are described in Table 7 below.

Table 7: Subscribe Functionality Mapping

Story ID	Functionality	Project Objective
UL-7 UL-8	Subscribe to podcasts Unsubscribe to podcasts	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

UL-9	User receives notification for each new episode in a podcast they are subscribed to	Listeners must be notified by the platform when a new episode for a show they are subscribed appear
UL-30	The latest episode for each subscribed podcast is linked in the "Subscriptions" page	Listeners must be able to see the latest episode available for each show that they subscribed to in a "Podcast Subscription Preview" panel

2.2 System

2.3 Implementation Challenges

2.3.1 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 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

[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

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.

Database Integrity

Populating the Site To build a meaningful recommendation system, the website must have a reasonable amount of podcasts already uploaded to it. Since UltraCast has not been released, there are no users to generate this data. To allow for experimentation with different approaches to recommending podcasts to users, a podcast dataset was scraped. It was difficult to find a suitable dataset that contained the required category, sub-category and keyword tags for podcasts that did not impose commercial obligations on UltraCast (due to terms of use of the dataset). A dataset which is an aggregation of public domain podcasts was found and scraped, providing over 200 podcasts and 2000 podcast episodes for the site.

3 User Manual

3.1 Software Setup Instructions

For the simple case where no API keys need to be changed, setting up and running UltraCast is as simple as running:

```
# cd to the root directory of the git repo
./start.sh
```

This script will:

- Create a python venv for the backend
- Install all required python packages in the venv
- Install all npm packages that are required for the frontend
- Launch the backend webserver
- Launch the frontend
- Open ultracast in your browser

3.2 Configuration

The following external services are used and their IP addresses and/or API keys will need to be set in configuration files:

- Algolia
- MongoDB Instance (hosted in a cloud container e.g. Amazon EC2)
- S3 Bucket
- Backend GraphQL endpoint (if not hosted on local machine)

Some of these variables need to be set for the frontend and some for the backend.

3.2.1 Backend Configuration

The backend is configured by using python files which set various configuration variables. These include options including:

- The IP address of the MonogDB instance
- The MonogDB database
- Flask secret keys (for encryption)
- Algolia API key and user

A full list of the variables that can be set is in `backend/config/default_settings.py`. Any variables that are not set are defaulted to the value in `backend/config/default_settings.py`. You can override these settings by writing a new python file and setting the environment variable `ULTRACAST_BACKEND_SETTINGS` to be the real path of this file. For example if the settings file is at `~/ultracast_settings.py`, you could do:

```
export ULTRACAST_BACKEND_SETTINGS=$(realpath ~/ultracast_settings.py)
bash backend/start.sh
```

3.2.2 Frontend Configuration

The frontend can be configured by editing the file `frontend/src/api/config.js` Here you can set options including:

- The backend GraphQL endpoint to use
- Algolia API key and user

3.3 Site Usage and Functionality Guide

References

[1] Algolia. https://www.algolia.com/doc/. Accessed: 2020-11-14.

[2] Flask: web development, one drop at a time. https://flask.palletsprojects.com/en/1.1.x/. Accessed: 2020-10-03.

[3] Graphql. https://graphql.org/. Accessed: 2020-10-03.

[4] React: A javascript library for building user interfaces. https://reactjs.org/. Accessed: 2020-10-03.

[5] MongoDB. The most popular database for modern apps — MongoDB, 2020. http://mongodb.com, Last accessed on 2020-10-03.

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