Adblocker für lineares Radio

Ein Interdisziplinäres Projekt der

FH-Aachen, Wintersemester 2023/24

Inhaltsverzeichnis

Aufgabenstellung

In unserem interdisziplinären Projekt “Adblocker für lineares Radio” des Wintersemesters 2023/24 haben wir die Aufgabe bekommen, eine Android-App zu programmieren, mit der man Radio hören kann.

Das Besondere: Sobald Werbung auf dem Radio ist, das man hört, soll automatisch auf ein Radio geschaltet werden, das gerade keine Werbung abspielt.

Die Mindestanforderungen (MVP) und weitere Wünsche lauteten dabei wie folgt:

- Entwicklung einer Android-App, bestenfalls sogar für iOS, Windows und als Webanwendung
- Manuell vorprogrammierte Zeitpläne, wann zu welchem Radio umgeschaltet werden muss. Bestenfalls eine automatische Erkennung der Werbung und Umschaltung zu einem passenden Radio
- Angabe der Vorlieben, welche Radios man am liebsten hören würde. Vorlieben könnten zum Beispiel “Musik” oder “Nachrichten”. Bestenfalls würde die App solche Vorlieben automatisch lernen können
- Automatisches Umschalten, wenn Werbung kommt. Bestenfalls vorher aufgenommene Fragmente (zum Beispiel ein Song) als Überbrückung einspielen, statt umzuschalten.

Das Endergebnis

Einführung

Wir haben eine Android-App entwickelt, mit der man Radios über das Internet hören kann. Im Falle, dass auf einem Radiosender die Werbung anfängt, wird automatisch auf ein vorher festgelegtes Radio gewechselt.

Die folgende Kurzübersicht soll einen Überblick darüber geben, was unsere App für Features anbietet:

- Auflistung aller von uns unterstützten Radios mit deren aktuell abgespielten Songs und der Angabe, ob dort im Moment Werbung gespielt wird
- Favorisierung der einzelnen Radios (favorisierte Radios bekommen von uns die Bezeichnung “Fluchtradio”)
- Priorisierung der Favoriten auf die im Falle von Werbung automatisch in der angegebenen Reihenfolge umgeschaltet wird
- Eine große Ansicht des aktuell gespielten Radios mit einer GUI, wie man sie aus Musik Apps wie Spotify kennt
- Einstellungsmöglichkeiten bieten an, die Farben der Anwendung auf dunkel oder hell zu stellen. Zudem gibt es eine Möglichkeit, dass Werbung optional aktiviert werden kann
- Keine Anmeldung erforderlich
- Es werden so wenig Daten wie nötig an den Server gesendet. Diese werden vom Server so bald wie möglich wieder gelöscht

Hier ist eine Liste aller von uns unterstützten Radios:

- Antenne AC (privat, NRW Lokalradio)
- Radio RST (privat, NRW Lokalradio)
- 100,5 Das Hitradio (privat)
- BAYERN 1 (öffentlich-rechtlich)
- BremenNext (öffentlich-rechtlich)

- 1Live (öffentlich-rechtlich)
- WDR2 (öffentlich-rechtlich)

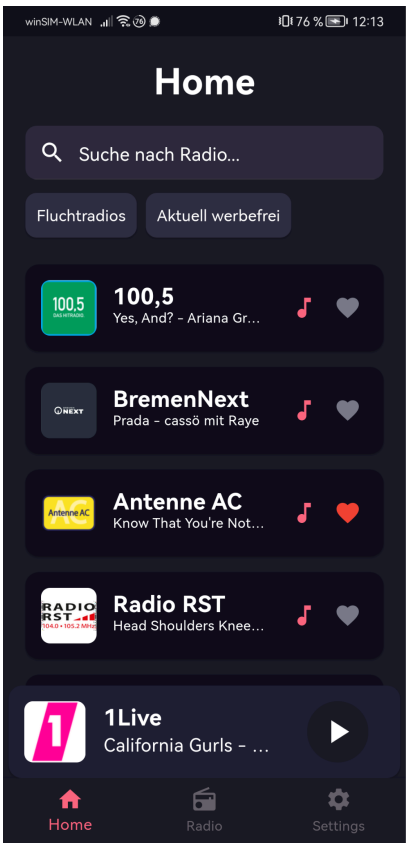
Homescreen

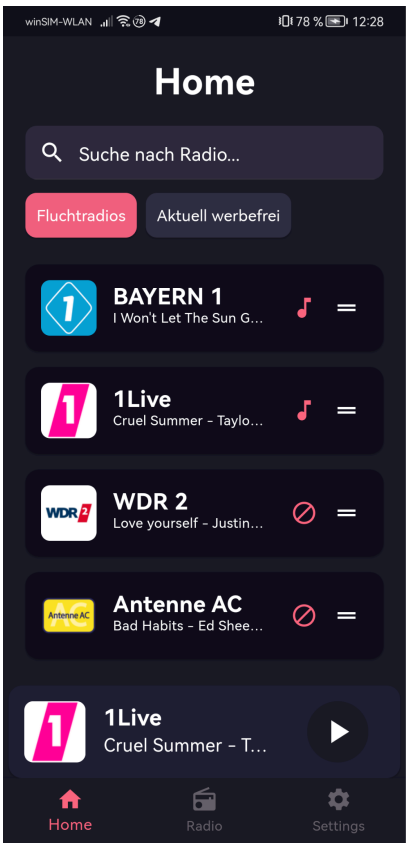
Hier werden alle verfügbaren Radios aufgelistet. Die Liste der Radios enthält Live Daten und wird laufend aktualisiert. Jeder Eintrag hat einen Indikator (Icon), ob dort gerade Musik oder Werbung läuft. Falls dort Musik abgespielt wird, wird außerdem der Songtitel angezeigt.

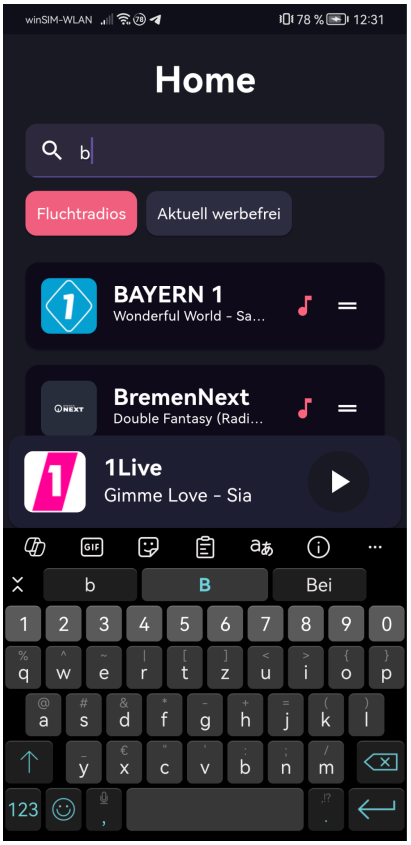
Zum Abspielen eines Radios muss dieses angetippt werden. Falls dieses gerade Werbung abspielt, so wird auf ein anderes Radio ausgewichen. Es wird jedoch vermerkt, dass das ausgewählte Radio eine Präferenz ist. Das abgespielte Radio wird am unteren Bildschirmrand in einer Kachel dargestellt. Zudem kann in der Kachel über den Play-/Pause-Button der Audiostream pausiert und wieder aufgenommen werden.

Radios können durch Berühren des Herzsymbols favorisiert werden. Favorisierte Radios werden dann zu einem “Fluchtradio”. Um sich alle Fluchtradios anzusehen, kann man dazu den Filter “Fluchtradio” aktivieren. Dort ist es nun möglich, eine Priorität festzulegen. Falls auf ein Radio ausgewichen werden muss, wird versucht, ein Radio auszuwählen, was möglichst weit oben steht. Falls keines der Fluchtradios Musik abspielt, wird vom Server ein zufälliges ausgewählt.

Damit die Radios, auf denen gerade Werbung gespielt wird, ausgeblendet werden, kann man den Filter “Aktuell werbefrei” auswählen. Zusätzlich kann über eine textuelle Suche gezielt nach Radios gesucht werden. Alle Filtermöglichkeiten können überdies beliebig miteinander kombiniert werden.


winSIM-WLAN
76 % 12:13
Home
Suche nach Radio...
Fluchtradios
Aktuell werbefrei
100,5
Yes, And? - Ariana Gr...
BremenNext
Prada - cassö mit Raye
Antenne AC
Know That You're Not...
Radio RST
Head Shoulders Knee...
1Live
California Gurls - ...
Home
Radio
Settings


winSIM-WLAN
78 % 12:28
Home
Suche nach Radio...
Fluchtradios
Aktuell werbefrei
BAYERN 1
I Won't Let The Sun G...
1Live
Cruel Summer - Taylo...
WDR 2
Love yourself - Justin...
Antenne AC
Bad Habits - Ed Shee...
1Live
Cruel Summer - T...
Home
Radio
Settings

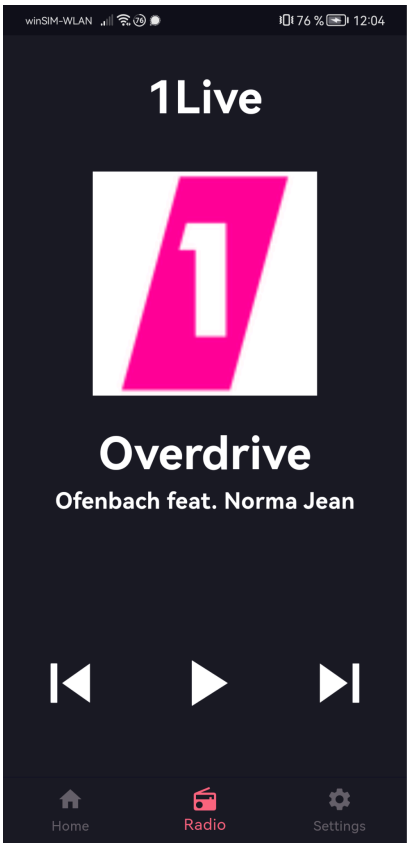

winSIM-WLAN
78 % 12:31
Home
b
Fluchtradios
Aktuell werbefrei
BAYERN 1
Wonderful World - Sa...
BremenNext
Double Fantasy (Radi...
1Live
Gimme Love - Sia
b
B
Bei

Radioscreen

Hier wird das aktuell gespielte Radio angezeigt. Die Menü-Icons sollten von Musikstreaming-Diensten bekannt sein.

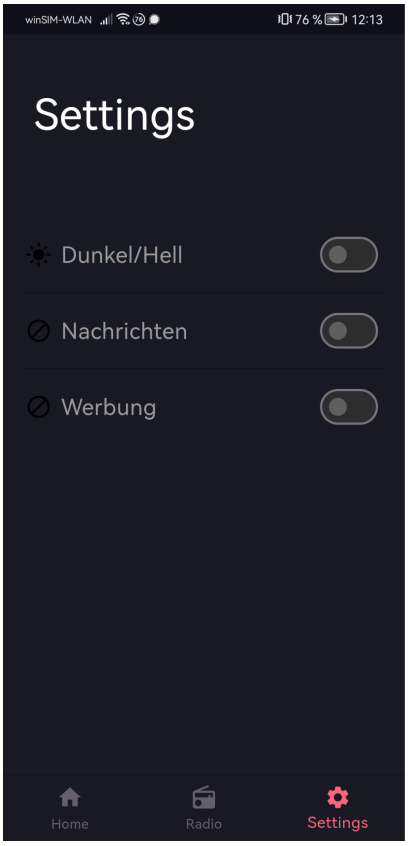
Die Buttons mit den Pfeilen nach links und rechts wählen das nächste bzw. vorherige Radio aus. Dabei wird durch die Liste aktuell verfügbarer Radios iteriert, die von dem Server bereitgestellt wird.

Der Benutzer bekommt die Informationen, welches Radio läuft, sowie Artist und Interpret, wenn gerade Musik läuft. Sollte gerade keine Musik gespielt werden, bekommt er Informationen über das, was alternativ gerade läuft. Sollten die Informationen platztechnisch nicht darstellbar sein, dann laufen sie von rechts nach links durch den Bildschirm, wie dies auch von anderen Musikstreaming-Diensten bekannt ist.



Settingsscreen

Diese Seite gibt die Option, dass man das Farbschema der Anwendung auf dunkel oder hell anpassen kann. SCREENSHOT AKTUALISIEREN


winSIM-WLAN
76 %
12:13
Settings
Dunkel/Hell
Nachrichten
Werbung
Home
Radio
Settings

Die Umsetzung

Unser Team besteht aus den Mitgliedern Ahmad Fadi Aljabri, Nicolas Harrje, Nils Stegemann, Chris Henkes, Kaan Yazici, Maximilian Breuer und Gerrit Weiermann.

Im ersten Schritt informierten wir uns über andere Projekte mit ähnlicher Aufgabenstellung. Bei unserer Recherche hat uns das Projekt “Adblock Radio” am meisten helfen können. Unsere Rechercheergebnisse, sowie Überlegungen haben wir näher im Anhang “Unsere Recherche” abgelegt. Im Folgenden werden wir in den meisten Fällen nur unsere Ergebnisse vorstellen. Wie wir zu unseren Entscheidungen gekommen sind, kann man dann in der Recherche genauer nachlesen.

Um uns besser zu organisieren, teilten wir uns in die zwei Untergruppen Frontend und Backend auf.

Frontend Gruppe

Das Frontend-Team beschäftigte sich mit allem rund um die Android-App. Angefangen mit dem Design und der User-Experience, bis hin zur funktionsfähigen App.

Die Mitglieder der Gruppe sind Ahmad Fadi Aljabri, Nicolas Harrje und Nils Stegemann.

Planung

Für die Implementierung der App einigten wir uns recht schnell auf Flutter als unterliegendes Framework. Dies war naheliegend, da einerseits mehrere aus der Gruppe Erfahrung damit hatten, es uns andererseits aber auch ermöglichte, die App für alle anderen Plattformen wie iOS, Windows oder als Webseite zu veröffentlichen. In der gesamten Entwicklung lag der Fokus jedoch trotzdem hauptsächlich auf der Android-Entwicklung.

Zunächst haben wir uns überlegt, über welche Funktionen die App verfügen soll. Dabei haben wir uns auf folgende geeinigt:

- Radio abspielen
- Übersicht über alle verfügbaren Radios
- Textbasierte Suche nach Radios
- Festlegen von favorisierten Radios
- Filtern nach aktuell werbefrei und Favoriten

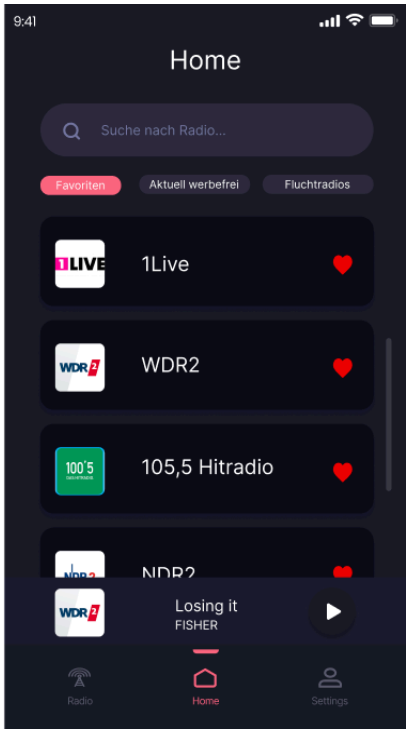
- Anti-Werbebutton (Sollte fehlerhafterweise Werbung laufen, kann dies mittels des Buttons gemeldet werden und zur Verbesserung des Backends beitragen)
- Light/Dark-Mode
- Große Darstellung des aktuellen gespielten Radio

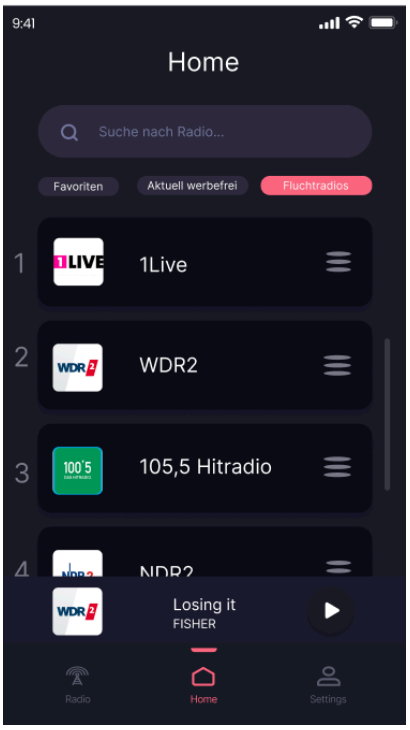
Anschließend haben wir uns überlegt, wie diese Funktionen sinnvoll in eine grafische Benutzeroberfläche überführt werden können. Dabei sind wir zu dem Ergebnis gekommen, dass eine Aufteilung in die drei Sichten Home, Radio und Settings sinnig ist und dass die Navigation über eine Navigationsleiste am unteren Bildschirmrand eine intuitive Art der Bedienung darstellen würde. Hierbei haben wir uns auch von anderen Musikstreaming-Diensten inspirieren lassen. Jedem Teammitglied wurde dann die Verantwortung für einen Screen übertragen. Auch wurde beschlossen, dass alle bei Bedarf am Homescreen mitarbeiten, da dieser am umfangreichsten war.

Mockups

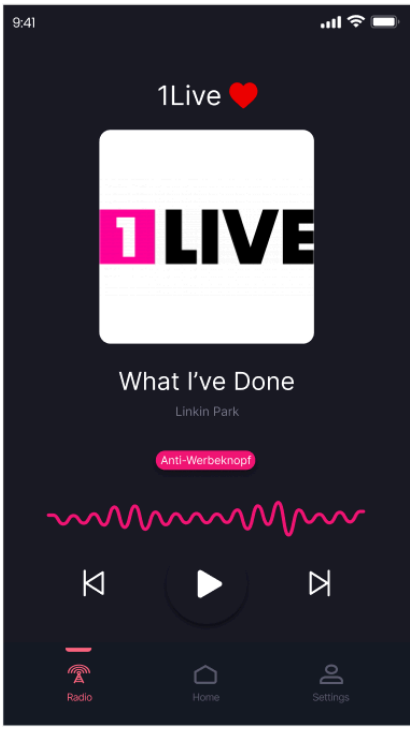
Die farbliche Gestaltung sowie die Strukturierung der einzelnen Screens sind inspiriert von einem Template. Für die Ausarbeitung und Gestaltung der Mockups haben wir die Software Figma verwendet.

Homescreen

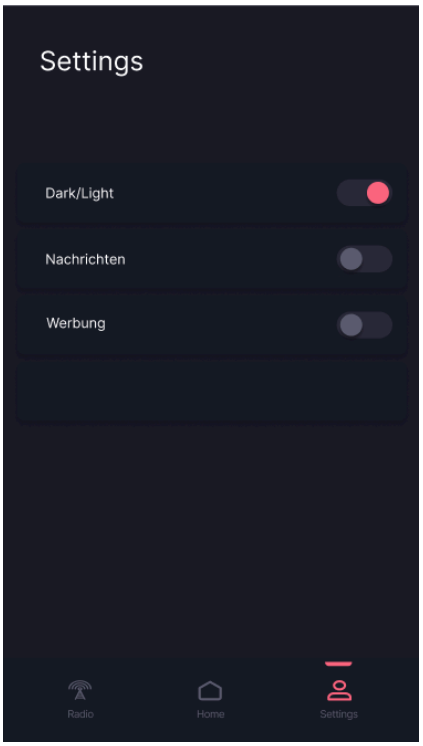




Radioscreen



Settingsscreen

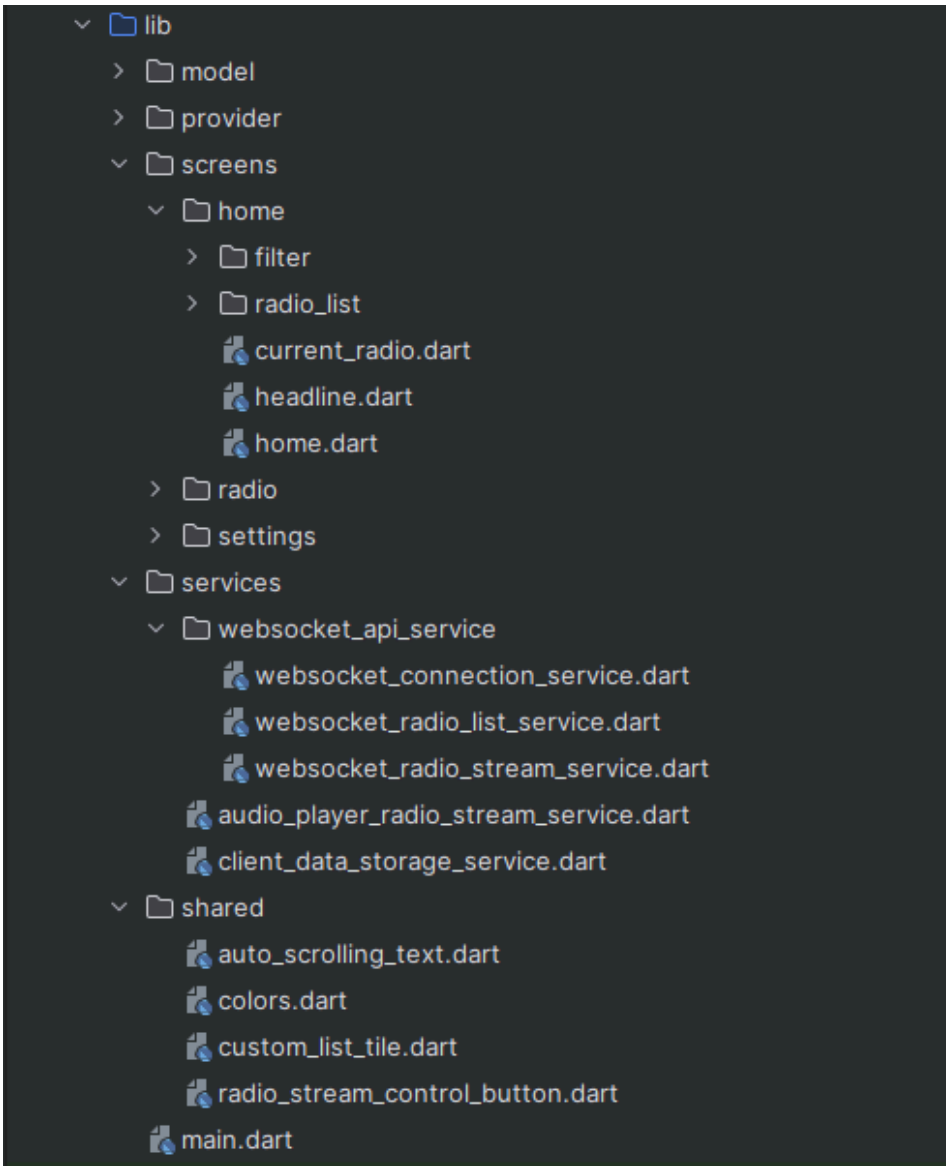


Programmierung

Nachfolgend soll der Ablauf der Entwicklungsarbeit dargestellt werden. Im Laufe dieser hat sich herauskristallisiert, dass einige Funktionen, wie z.B. der Anti-Werbe-Button weniger wichtig sind. Wir haben aber auch festgestellt, dass eine Priorisierung der Radiosender, sowie einige Veränderungen des UX das Benutzererlebnis verbessern würden. Daher sind diese Veränderungen nicht in den Mockups enthalten, jedoch in der folgenden Ausführung.

Erstellung allgemeiner App Struktur

Zu Beginn der Programmierung wurde von uns eine grobe Ordnerstruktur erstellt. Hierbei wurden jeweils Ordner für die einzelnen Screens angelegt. Im weiteren Verlauf des Entwicklungsprozesses wurden diese um weitere Ordner ergänzt. Die finale Ordnerstruktur ist nachfolgend abgebildet.



Temporäre Daten

Da zu Anfang des Entwicklungsprozesses noch keine Schnittstelle zum Server bestand, füllten wir alle relevanten Stellen mit Dummie-Daten. So konnten wir testen, wie die Darstellung der ersten Entwicklungen aussah.

Filteroptionen

Zur besseren Orientierung in der Radioliste haben wir Filtermöglichkeiten hinzugefügt. Als Filtermöglichkeiten steht eine textuelle Suche, die Filterung nach Fluchtradios (Favoriten) und die Anzeige von aktuell werbefreien Radiosendern.

Für die technische Umsetzung verwenden wir eine Liste, welche alle angewendeten Filterqueries speichert. Wenn kein Filter aktiv ist, bleibt die Liste leer. Erst, wenn ein Filter aktiviert wird, wird dieser der Liste hinzugefügt. Dies ermöglicht auch die Kombination verschiedener Filteroptionen. Sobald ein Filter nicht mehr aktiv ist, wird er wieder aus der Liste der Filterqueries entfernt.

Bevor die Radioliste dargestellt wird, durchläuft sie alle Filter aus der Filterqueries Liste. Infolgedessen werden nur die Radios angezeigt, welche alle Filterqueries erfüllen.

Persistierung

Damit die Favoriten mit ihren Prioritäten und das zuletzt gehörte Radio auch nach dem Neustarten der App gespeichert bleiben, haben wir eine Persistierung dieser Daten beim Client hinzugefügt.

Für die Implementierung der Persistierung haben wir die Library SharedPreferences verwendet. Die Library ermöglicht es, Daten lokal auf dem Gerät zu speichern und abzurufen. Die Verwaltung wird über eine gesonderte Klasse realisiert.

Beim Start der App werden die Favoriten und das zuletzt gehörte Radio aus den gespeicherten Präferenzen geladen. Durch einen Klick auf das Herzsymbol eines Radioeintrags wird die ID des Radiosenders entweder zu den Favoriten hinzugefügt oder bei schon vorhandenen Favoritenstatus wieder entfernt. Diese Änderungen werden anschließend in den Präferenzen gespeichert. Jedes Mal, wenn ein Radio abgespielt wird, wird es automatisch als das neue, zuletzt gehörte Radio gespeichert. Diese Information wird ebenfalls in den Präferenzen aktualisiert.

Immer wenn der Homescreen geladen wird, werden die Prioritäten der Radios aus den Präferenzen geladen. Die IDs der Radiosender werden als Liste persistiert, wobei der Index der Liste die Priorität des jeweiligen Radiosenders darstellt. Wenn Änderungen an der

Priorisierung vorgenommen werden, werden diese den Präferenzen übernommen und abgespeichert.

Audiowiedergabe

Für die Steuerung des Audioplayers wurde eine Klasse für die Audiowiedergabe als Singleton implementiert. So wird Dateiübergreifend die gleiche Instanz verwendet. Infolgedessen steht in der gesamten App der gleiche Audiostream verfügung . Mittels dieser Klasse wurde ein AudioButton-Widget umgesetzt, welches für das Starten und Stoppen des Audiostreams zuständig ist. Dieser Button findet Verwendung in den beiden Screens Home und Radio. Somit ist es möglich, dass beide Buttons den gleichen Stream steuern und immer den gleichen Zustand haben.

Anbindung der Websocket API

Damit die beiden nachfolgenden Klassen eine Verbindung zum Server herstellen können, wurde die Klasse WebSocketConnectionService entwickelt, die diese Verbindungen erzeugt und bereitstellt.

Damit der HomeScreen die Liste der Radio-Sender darstellen kann, muss diese vom Server empfangen werden. Dazu wird beim Start der App die Methode requestRadioList der Klasse WebSocketRadioListService aufgerufen. Diese überprüft, ob eine Verbindung zum Server besteht, initialisiert im Zweifel eine, und stellt eine Anfrage an den Server. Über einen StreamProvider wird dann die Funktion getRadioList der selbigen Klasse aufgerufen. Diese empfängt die Antwort vom Server und verarbeitet diese so, dass sie einen kontinuierlichen Strom von Listen zurückgibt, die alle verfügbaren Radios mit ihren jeweiligen Attributen enthalten. So werden auch alle Updates bezüglich der Anfrage direkt an den Provider weitergegeben. Der Provider stellt diesen Strom nun global in der App zur weiteren Verwendung zur Verfügung.

Die Bereitstellung eines spielbaren Radios erfolgt in der Klasse WebSocketRadioStreamService. Sie funktioniert ähnlich wie die oben erklärte Bereitstellung der Radio-Liste. Jedoch wird eine andere Anfrage an den Server gesendet. Diese beinhaltet das angefragte Radio sowie eine Liste der favorisierten Radios. Die Antwort des Servers enthält nun das spielbare Radio (Link zum Stream und weitere Metadaten). Die Funktion getStreamableRadio empfängt diese Nachricht und alle Updates und stellt diese einem StreamProvider zur Verfügung. Dieser wiederum stellt den Stream global zur Verfügung.

Die Klassen WebSocketRadioListService und WebsocketRadioStreamService bauen beide eine seperate Verbindung zum Server auf.

Verbesserung der UX

Um dem Benutzer eine intuitive Bedienung der App zu gewährleisten, haben wir uns im Laufe des Projekts dazu entschieden, einige Funktionalitäten anders umzusetzen, als sie zu Beginn des Projekts geplant waren.

Ursprünglich sollte der Audiostream nur dann starten, wenn der Play-/Pause-Button geklickt wurde. Es stellte sich jedoch als benutzerfreundlich heraus, wenn zusätzlich beim Anwählen eines Radios der Stream sofort startet.

Da die App vorsieht, dass Radios, auf denen Werbung läuft, nicht gestreamt werden können, kann es vorkommen, dass das gewünschte Radio nicht abgespielt wird. Stattdessen wird ein anderer Radiostream gestartet. Diese Funktionalität kann für den Nutzer als fehlerhaftes und kontraintuitives Verhalten wahrgenommen werden. Um dem entgegenzuwirken, haben wir uns dazu entschieden, dem Nutzer mehr Feedback zu geben. Wenn der Nutzer ein Radio auswählt, auf dem gerade Werbung läuft, erscheint ein Dialogfenster auf dem Bildschirm mit einem entsprechenden Hinweis. Außerdem haben wir jeden Radioeintrag mit einem zusätzlichen Icon versehen, welches den Status des Radios angezeigt. Ist das Icon eine Musiknote, so wird gerade Musik gespielt. Ist das Icon ein Blocksymbol, so wird gerade Werbung gespielt.

Des Weiteren haben wir eine Benachrichtigung hinzugefügt, die immer dann erscheint, wenn die App die Verbindung zum Internet verloren hat.

Prioritäten-Vergabe

Die Prioritäten-Vergabe regelt, auf welches Fluchtradio gewechselt wird, wenn auf dem gewählten Sender gerade Werbung läuft. Die Prioritäten der einzelnen Radios können verändert werden, wenn der Filter Fluchtradios ausgewählt ist. Dies ist so geregelt, da davon ausgegangen wird, dass der Benutzer möchte, dass in jedem Fall auf ein Fluchtradio gewechselt wird, sollte gerade Werbung laufen. Die Reihenfolge der Listenelemente spiegelt die Priorisierung wider. Dabei stellt das oberste Element das höchst priorisierte dar. Die Priorität ist nach unten hin absteigend geordnet. Durch drag-and-drop mittels des Symbols mit den zwei weißen Strichen kann die Reihenfolge (Priorität) verändert werden.

Um diese Funktionalität zu ermöglichen, wird, sobald der Fluchtradios-Filter gesetzt ist, eine reorderableListView zurückgegeben. Sobald die Reihenfolge dieser verändert wird, wird sie persistiert. Jedes Mal, wenn der Homescreen neu geladen wird, (nach neu Anordnen der Liste, Auswahl eines Filters oder Auswahl des Homescreens) wird die persistierte Priorisierung geladen und die verfügbaren Radio-Sender vom Server angefragt.

Anschließend wird die Radio-Liste nach Priorität sortiert und anschließend grafisch dargestellt.

Backend Gruppe

Das Backend beschäftigt sich um die Analyse der einzelnen Radios, des Einpflegens von Radios und der Bereitstellung der Metadaten, in denen enthalten ist, ob gerade Werbung läuft, sowie Informationen über den aktuell gespielten Song.

Mitglieder der Gruppe sind Chris Henkes, Kaan Yazici, Maximilian Breuer und Gerrit Weiermann.

Die Backend Gruppe erarbeitete sich eine in Python geschriebene Serversoftware, die in einem Docker Container läuft.

Die App wird eine persistente WebSocket-Verbindung zum Server aufbauen und kann zwei Aktionen ausführen:

- Eine Liste aller Radios anfordern:
  Die einzelnen Einträge enthalten Daten wie Logo, Streaming URL, Songname und die Information, ob gerade Werbung läuft.
  Verändern sich diese Werte, weil bspw. ein Radio gerade auf Werbung umgesprungen ist, wird die aktualisierte Liste sofort an den Client übermittelt.
- Dem Server mitteilen, welches Radio gerne gehört werden will:
  Anbei übermittelt man ebenfalls seine Fluchtradios in der Reihenfolge, wie man sie priorisieren möchte.
  Der Server gibt daraufhin zurück, welches Radio am geeignetsten sein wird (das erstbeste Radio ohne Werbung).
  Der Server merkt sich, welches Radio von dem Nutzer gehört wird und benachrichtigt die App sofort, sobald er bemerkt, dass auf diesem Radio gerade Werbung läuft und bietet ein alternatives Radio an.

Eine vollständige Dokumentation der Schnittstelle findet man im Anhang “Server Schnittstellendokumentation”

Die App wird das Radio nicht direkt von unserem Server streamen, sondern von dem eigentlichen Anbieter. Das reduziert nicht nur unsere Serverlast, sondern verhindert auch urheberrechtliche Probleme. Stattdessen gibt der Server nur Anweisungen, wie sich die App verhalten soll.

Der Server, den wir in der Zeitspanne unserer Projektarbeit nutzten, wurde von dem externen Anbieter Netcup gehostet (RS 1000 G9.5 a1 12M, mit vier dedizierten Kernen und 8 GB RAM).

Mit den vier Kernen schaffte es der Server, sieben bis acht Radios gleichzeitig zu analysieren. In der Regel wurden nur 1GB von den gesamten 8GB Arbeitsspeicher benötigt. Möchte man die Leistung des Servers skalieren, müsste also vor allem auf die Anzahl der Kerne achten.

Werbeerkennung durch Zeit

Im ersten Schritt wollten wir das MVP umsetzen: Zeitbasierte Werbeerkennung.

Wir recherchierten zuerst, wann Radios in der Regel ihre Werbung schalten und haben dabei auch erfahren, dass bspw. öffentlich-rechtliche Radios nur zwischen 6 und 22 Uhr Werbung schalten dürfen.

Im nächsten Schritt hörten wir dann für eine lange Zeit alle von uns ausgewählten Radios und schrieben die Zeiten mit, wenn Werbung auftrat. Wir bemerkten schnell, dass diese Zeiten nie punktgenau waren und zumindest um wenige Minuten variierten.

Die gemittelten Zeiten haben wir dann in die Datenbank geschrieben und mit einem Skript den jeweiligen Status der Radios geändert.

Die Änderung dem Client mitzuteilen war am Anfang etwas schwierig umzusetzen, da wir die API ursprünglich so gestalteten, dass auch Suchanfragen möglich waren. Man musste erst einmal herausfinden, welche Clients das aktualisierte Radio überhaupt auf der Oberfläche darstellen wollen. Die SQL-Abfrage dafür war relativ komplex.

Fingerprinting

Natürlich wollten wir die Werbeerkennung verbessern und entschieden uns dafür, dass wir die Werbejingles erkennen, die Radios immer vor einer Werbung abspielen.

Wir setzten uns also daran, die Radiostreams aufzunehmen und die Werbejingles herauszukristallisieren.

Andererseits mussten wir uns auch für eine Python-Bibliothek entscheiden. Mehr dazu findet man unter “Unsere Recherche”.

Die Implementierung brachte mehrere Herausforderungen mit. Diese werden genauer unter “Herausforderungen” dargestellt.

Metadaten

Damit die App auch die aktuell gespielten Songs darstellen kann, mussten wir uns darum kümmern, die Informationen “Songname” und “Interpret” zu bekommen.

Nach näherer Betrachtung von der Webseite auf radio.de fanden wir heraus, dass es dort eine inoffizielle API gibt, die wir dann auch angebunden haben.

Leider bietet diese API nicht für alle Radios diese Metadaten an, weshalb wir diesen Schritt ebenfalls für die API von NRW Lokalradios wiederholt.

Erfüllung der Mindestanforderungen

Software

- ☑ MVP: Android-App
- ☐ Optional: iOS-App
- ☑ Optional: Desktopanwendung (Windows)
- ☐ Optional: Webanwendung

AdBlock

- ☑ MVP: Zeitgesteuerte Umschaltung zwischen vorgegebenen Sendern
- ☑ Optional: Auswahl eines Senders und automatische Einblendung von Inhalten eines anderen Senders

Auswahlmöglichkeiten

- ☑ MVP: Einstellen von Radio Vorlieben wie zum Beispiel “Musik”, “Nachrichten” und “Lokales”
- ☐ Optional: Automatische Erkennung der Vorlieben des Nutzers

Methodik

- ☑ MVP: Direktes Umschalten zwischen den Stationen
- ☐ Optional: Einstanzen von Fragmenten bei Bedarf mit zeitlicher Verschiebung

Hardware

- ☑ MVP: Bereitstellung eines Internetradios als Software (Android-App)
- ☐ Optional: Smartphone-App und Integration in ein physisches Radio

Weitere erreichte Ziele

- Hoher Grad an Privatsphäre
- Keine Anmeldung erforderlich
- Für das Hinzufügen oder der Aktualisierung von Radios muss keine neue Version der App herausgebracht werden
- Für die Implementierung von neuen Erkennungsmechanismen wird kein App Update benötigt
- Schönes App-Design
- Near-Realtime Daten zu den einzelnen Radios
- Einfaches Aufsetzen des Servers möglich (mit Docker)

Ausblick

Businessmodell

Da der Server nicht kostenlos ist, muss man sich langfristig über eine Finanzierung Gedanken machen. Hier sind mehrere Ansätze, die wir uns überlegt haben:

Analyse der Konkurrenz

Der Server kann die Abläufe der Radios loggen und die klassifizierte Werbung einer Firma zuordnen. Die Konkurrenz der Firmen, die Werbung ausstrahlen lässt, ist möglicherweise daran interessiert, wann und wie oft von welcher Firma Werbung ausgestrahlt wird.

Kostenpflichtige App

Die App könnte man im Google Play Store und App Store für einen angemessenen Preis veröffentlichen. Will man werbefreies Radio hören, muss man also einmalig diese App kaufen.

Abomodell

Statt nur einmalig für die App zu bezahlen könnte man auch ein Abomodell einführen, dass man bspw. monatlich für die App zahlen muss. Hier sollte man aber ein Alleinstellungsmerkmal ausarbeiten, um sich besser von Streaming Diensten wie Spotify abgrenzen zu können.

Daten der Benutzer sammeln und verkaufen

Das Benutzerverhalten über die Fragestellungen:

- Welche Radios werden gerne gehört?
- Zu welchen Tageszeiten wird die App benutzt?
- Wie lange wird am Stück Radio gehört?
- Bei welchen Liedern wird das Radio vom Nutzer gewechselt?
- Wo befindet sich der Nutzer? (Standortdaten)

können gesammelt und an Dritte verkauft werden. Man sollte hierfür aber eine Marktanalyse durchführen, da Nutzer von Adblockern meist dafür sensibilisiert sind, ihre Daten bestmöglich zu schützen.

Werbebanner in der App anzeigen

Die App könnte einen Werbebanner anzeigen oder sogar immer wieder Popups anzeigen. Jedoch ist die App eigentlich genau aus der Idee heraus entstanden, Werbung zu blockieren, sodass dieser Lösungsweg zweifelhaft erscheint.

Skalierbarkeit

Backend

Im Bezug zur Anzahl der Nutzer, die gleichzeitig auf der App unterwegs sind

Da die Streams selbst nicht vom Server übertragen werden, sondern nur Websocket Verbindungen für Metadaten aufgebaut werden, ist der Overhead hier relativ schlank. Man müsste schauen, wie viele Verbindungen der Server gleichzeitig aufbauen kann, bis er damit in die Knie gezwungen wird.

Maximal wird man hier sicherlich auf die Anzahl der möglichen Ports (65536 - 1024 reserviert) beschränkt, wahrscheinlich aber bereits auf eine kleinere Zahl.

Um mehr Anfragen bearbeiten zu können, müsste man ein Load Balancing betreiben.

Es lohnt sich außerdem zu schauen, wie viele Verbindungen Flask (das Webframework) gleichzeitig handhaben kann. Wir haben im Moment die Nicht-Production Umgebung von Flask eingesetzt. Zudem läuft der Server im Moment auf nur einem Thread. Hier ist wohhl das größte Verbesserungspotenzial.

Derzeit werden alle aktuellen Verbindungen in einem Dictionary gespeichert, sodass man ein Mapping hat, die die connection_id aus der Datenbank auf das Websocket Objekt des Servers abbildet, sodass alle Funktionen mit den Clients über die connection_id kommunizieren können.

Ein Dictionary sollte in der Regel viele tausende Einträge vorhalten können, sodass auch die Obergrenze von 65536 Ports - 1024 Ports nicht problematisch werden sollte.

Im Bezug zur Anzahl der angebotenen Radioanzahl

Eine weitere, größere Auslastung stellt die Analyse dar. Es werden viele verschiedene Radios durch den Server gestreamt. Jedes Radio durchläuft eine Analyse mit Fingerprinting und ggf. auch eines KI-Modells. Dies ist sicherlich eines der größten Aufwände, den der Server zu stemmen hat.

Im Vergleich kann man sich jedoch folgendes überlegen:
Man kann die Anzahl der analysierten Radios selbst bestimmen und somit auf die Leistung des Servers abstimmen. Die Anzahl der Nutzer (siehe oben) lässt sich weniger gut steuern. Daher sollte besonders auf den ersten Punkt ein Augenmerk liegen, um den Dienst ausreichend vor Ausfällen zu schützen.

Im Moment kann die API keine Filterung der Radios vornehmen. Die Filterung wird durch den Client vorgenommen (siehe unten). Da die Anzahl der Statusupdates der Radios stark zunimmt, je größer die Anzahl der Radios ist, die man beim Client auf dem neuesten Stand halten muss, könnte alles außer Kontrolle geraten, wobei der Client mit Updates quasi zugespammt wird.

Ab so einem Punkt müsste man sich über Filteroptionen Gedanken machen (siehe unten).

Frontend

Aktuell wird die gesamte Liste der Radios an den Client übertragen. Dies mag bei einer Liste mit Anzahl < 100 noch zuverlässig funktionieren, jedoch ist dieser Ansatz nicht skalierbar. Besser wäre es, eine Pagination hinzuzufügen, sodass die Einträge in kleinen Blöcken angefragt werden können. Im Sinne von: “Gib mir Einträge 0 bis 50”, man scrollt weiter: “Gib mir Einträge 50 bis 100”.

Augenblicklich wird die Filterung vom Client übernommen. Beim Steigern der Radioanzahl könnte dies zu Performance-Problemen führen. Der Server hat intern bereits eine Struktur zur Filterung, die erweitert und als API zur Verfügung gestellt werden müsste. Dies müsste das Frontend sowie das Backend noch nachträglich hinzufügen.

Die App baut derzeit zwei Verbindungen zum Server auf. Dies beansprucht unnötig viele Ressourcen vom Server. Besser wäre es, sämtliche Kommunikation über eine Verbindung laufen zu lassen.

Testing

Die Korrektheit der aktuellen Version für das Fingerprinting, sowie das Erhalten der Metadaten wurde durch Real Time Testing verifiziert. Um eine bessere Robustheit des Programms sicherzustellen, wäre es sinnvoll, ein Testing Framework zu verwenden, bei dem man mit Unit-Tests arbeitet, um bspw. die Funktionen der API im Backend gründlich zu testen, ohne dass das Frontend Fehler bekommt, für die sie gar nicht verantwortlich sind.

Tests, die ständig überprüfen, ob die Metadaten und Werbestatus aktualisiert werden. Zuletzt gab es immer wieder das Problem, dass die Metadaten irgendwann aufgehört haben, sich zu aktualisieren.

Abhängigkeiten

API

Wir nutzen für unsere Metadaten die folgenden APIs:

https://prod.radio-api.net/stations/now-playing?stationIds=<station_id1>,<station_id2>,...

https://api-prod.nrwlokalradios.com/playlist/current?station=<station_id>

Diese nutzen wir ungefragt und diese wurden außerdem nicht nach außen hin für die Öffentlichkeit dokumentiert. Es kann also sein, dass sich die APIs jederzeit einfach ändern können.

URLs

Wir haben einmalig die Stream URLs der von uns angebotenen Radios herausgesucht. Ändern diese sich bei dem Anbieter, funktioniert das Radio bei uns nicht mehr. Dadurch müssen bei der Weiterverwendung des Projektes die Radio URLs je nachdem gepflegt werden. Bei den URLs ist zu beachten, dass der Content-Type durchgängig “audio/mpeg” ist, da unsere Fingerprint-Bibliothek nur mpeg unterstützt.

Synchronisation

Server - Client

Nach aktuellem Stand laufen Server und Clients asynchron, was je nach Ping der Clients auf Dauer problematisch werden könnte. Wenn die Clients zu weit hinten oder vorne hängen, wird Werbung, je nachdem, zu spät oder zu früh erkannt.

Fingerprinting

Jingles

Jingles sind kurze einprägsame Melodien, die bei Radiosendern kurz vor Nachrichten oder Werbungen geschaltet werden. Die Jingles für das Fingerprinting sind ja nach Alter des Projekt möglicherweise nicht mehr aktuell und müssten durch die neuen ersetzt werden. Je nach Radio existieren Saisonale Jingles wie z.B Weihnachtsjingles, die angepasst werden müssten.

Zeitbasiertes Fingerprinting

Stand jetzt wird dauerhaft in allen Radio gefingerprintet, was viel Serverleistung zieht.
Ein Ansatz, um das zu umgehen, wäre zeitbasiertes Fingerprinting, also Fingerprinting in bestimmten Zeitabschnitten. Dadurch jedoch könnten dann Jingles verpasst werden, da nicht alle Radios strikte Werbezeiten haben.

Fingerprinting Queue

Die Aufnahme Schnipsel der Radios werden alle in einer Queue getan, die nach und nach abgearbeitet wird. Es könnte jedoch passieren, dass sich das schneller füllt als bearbeitet, was zu verzögerten Fingerprints führt. Man müsste eine Warnung einbauen, die signalisiert, wenn dies geschieht und die Queue leert.

Weitere Herausforderungen

MySQL und PostgreSQL Datenbank

Wir sind mit einer PostgreSQL Datenbank gestartet. Als wir dann mit den Fingerprints mittels der Library Dejavu anfangen wollten, ist uns aufgefallen, dass es für die Version ab Python 3.x keine PostgreSQL Anbindung mehr gibt. Wir haben viel Zeit darauf angewendet

entweder alles auf MySQL zu machen oder einen neuen PostgreSQL Driver zu programmieren. Beide Ansätze sind leider ins Leere gelaufen, weshalb wir als Notlösung nun beide Datenbanken parallel laufen lassen.

Ansatz: PostgreSQL auf MySQL migrieren

~~Wenn wir den Treiber für MySQL nutzen, gibt es Fehler bezüglich Race Conditions, dass Commits in falscher Reihenfolge ausgeführt werden. Man müsste dort schauen, wie man für jeden Thread eine eigene Transaktionen erstellt. Dies sollte eigentlich sowieso ein Problem auch bei PostgreSQL sein, jedoch wird dort kein solcher Fehler gemeldet.~~

Update: Jede Verbindung hat nun ihre eigene Datenbank-Verbindung und somit eigene Transaktionen. Es gibt teilweise PostgreSQL spezifische Aufrufe, die man migrieren müsste. Aufgrund der niedrigen Priorität, haben wir das aber nicht mehr umgesetzt.

Ansatz: Eigenen Treiber schreiben

Der Treiber der Library müsste von Psycopg2 auf Psycopg3 migriert werden. Dieser hat sich in seiner Funktionalität aber stark verändert. Nach stundenlangem Ausprobieren haben wir uns dann dazu entschieden, auch diesen Ansatz zu verwerfen.

Es macht sicherlich Sinn, sich auf eine Datenbank zu einigen und wir finden diese Lösung bedauerlich. Die Behebung dieses unschönen Details sollte man bei vorhandener Zeit erneut aufgreifen, da die Umsetzung ziemlich sicher umsetzbar sein wird.

Erweiterungen

Hier wird kurz darüber gesprochen, welche Features noch wünschenswert gewesen wären.

Meldefunktion

Um bei Nichterkennung bzw. Falscherkennung von Werbejingles dem Client eine Möglichkeit zu bieten, dem Server einen Bugreport zu senden, wäre es interessant, eine Schaltfläche in der Radioansicht einzubauen (Anti-Werbe-Button). Diese Schaltfläche müsste nur von dem Nutzer angeklickt werden. Dem Server würde somit alle notwendigen Daten zugesendet werden, um Informationen über den Bug zu bekommen.

Für die Erweiterung dieser Meldefunktion müsste das Backend in der API eine Schnittstelle erstellen, mit der die Bugreports erhalten und verarbeitet werden können.

Allgemeine Bugreports

Dem User eine Möglichkeit bereitstellen, Fehler innerhalb der App dem Server zu jedem Zeitpunkt senden zu können, wäre eine wichtige zukünftige Funktion. Durch solch eine Erweiterung, ist dem Entwicklerteam geholfen, schneller Bugs zu erkennen und ebenfalls zu beheben. Vor allem ist dies eine Funktion, die dem Frontend hilft, weil clientseitige Bugs dann dem Server vorliegen und diese auch später noch erkannt werden können.

Automatische Logs an den Server senden

Treten in der App Warnings oder Errors auf, sollten diese an den Server automatisch übermittelt werden, damit auch ohne Zutun des Nutzers eine Fehlerbehandlung durchgeführt werden kann.
Hier sollte man sich aber mit der datenschutzrechtlichen Frage auseinandersetzen, ob dies so umsetzbar ist und ob man möglicherweise bestimmte Informationen vorher “schwärzen” muss.

An- und Ausschalten von Nachrichten/Werbung

In dem Settingsscreen sind bereits zwei Schalter für diese Funktionalität vorhanden. Der Benutzer könnte so entscheiden, ob er Nachrichten/Werbung hören möchte, oder nicht. Denkbar wäre auch, dies noch weiter zu verfeinern, so dass der Benutzer entscheiden kann, auf welchen Radiosendern er Nachrichten/Werbung hören möchte.

Grafische Darstellung einer Tonspur im Radioscreen

In den Mockups des Radioscreens ist bereits eine pinke Animation einer Tonspur angedeutet. Diese dürfte das Erscheinungsbild des aktuellen Radioscreens aufhübschen.

Kommentarsektion im Radioscreen

Sollte das Businesmodell der Anzeigenschaltung (Werbebanner) gewählt werden, könnte eine Kommentarsektion in dem Radioscreen die Aktivität der Benutzer erhöhen. Dies würde zu mehr Bildschirmzeit auf unserer App führen und somit die Werbeeinnahmen erhöhen. Dabei würde dem Benutzer ermöglicht werden, den Radiostream zu kommentieren. Diese Kommentare könnten dann anderen Nutzern angezeigt werden, welche wiederum darauf reagieren könnten.

Speichern von Songs in Musik-Streamingdiensten

Für viele Radiohörer dürfte folgende Situtation bekannt sein: Man hört einen unbekannten Song im Radio und möchte diesen zu der Bibliothek seines genutzten Musik-Streamingdienstes hinzufügen. Um dies bequemer zu gestalten, könnte eine Anbindung zu bekannten Musik-Streamingdiensten implementiert werden. So könnte der Song dann zu den Favoriten oder zu einer Playlist hinzugefügt werden. Denkbar wäre auch, das Generieren einer Tages-Playlist eines Senders. Diese kann anschließend modifiziert und zum gewählten Streamingdienst exportiert werden.

Autoscrolling Animation auf dem Homescreen

Wenn der Text breiter ist als der verfügbare Platz, wäre es sinnvoller, den Text mit Hilfe einer Animation darzustellen, sodass er komplett gelesen werden kann. Sinnvoll wäre diese Animation z.B. bei langen Interpreten oder Songnamen.

Die Animation lässt den Text von rechts nach links durchlaufen. Eine solche Animation wurde schon implementiert und ist auch im Radioscreen im Einsatz. Auch auf dem Homescreen wurde sie schon einmal angewendet, führte jedoch zu Fehlern, deren Lösung noch offen ist. Daher wurde die Animation vorerst aus dem Homescreen entfernt. In einer zukünftigen Version könnten die Fehler behoben und die Animation wieder verwendet werden.

Anmerkungen

Das Feld ad_until eines Radios gibt bei Radios mit Endjingle an, dass es nach Start der Werbung erst nach sechs Minuten wieder zurückspringt. Dies ist natürlich irreführend, da es bei anderen Radios so ist, dass es die genaue Zeit angibt, wann es von Werbung wieder zurückspringen wird.

Bei Radios mit Endjingles handelt es sich um die maximal Dauer, falls dann doch kein Endjingle kommt. Bei solchen Radios ist das Feld ad_duration = 0.

Herausforderungen

Frontend

Kommunikation mit dem Server

Die wahrscheinlich größte Herausforderung stellte für uns die Programmierung der Serverschnittstelle dar. Es galt zunächst große Wissenslücken zu schließen, eine geeignete Bibliothek zu finden und diese zu verstehen. Während der Implementierung traten ständig Exceptions auf, bei denen es uns zuerst nicht gelungen ist, sie abzufangen. Dies hat uns großes Kopfzerbrechen bereitet.

Des Weiteren ist uns erst relativ spät klar geworden, dass für die Bereitstellung der Liste und des spielbaren Radios jeweils eine Verbindung zum Server aufgebaut werden muss. Lange Zeit erfolgte beides über die gleiche Verbindung, was ständig zu Fehlern führte. Die Ursache des Problems zu finden, hat lange Zeit in Anspruch genommen.

Datenbereitstellung

Damit die vom Server empfangenen Daten global in der App zur Verfügung gestellt werden konnten, mussten sogenannte Provider verwendet werden. Anfangs entschieden wir uns für changeNotifierProvider, welche auf ein Signal warten, dass die Daten sich verändert haben. Jedoch konnten wir ein solches Signal nicht bereitstellen. Daher mussten wir einen geeigneteren Provider finden. Dies sollte der streamProvider sein, da wir einen ständigen Strom an Daten vom Server empfangen.

Serverausfall

Die Entwicklungen im Frontend wurden ständig mit einem Emulator, der ein Smartphone emuliert getestet. Hierbei waren wir auf eine Verbindung zum Server angewiesen. Traten nun Fehler auf, suchten wir den Fehler lange Zeit bei uns, bis uns klar wurde, dass er beim Server liegen müsste. Dies beanspruchte unnötig viel Zeit und Mühe.

Persistierung

Während der Datenpersistierung mithilfe von Providern stießen wir auf einige Schwierigkeiten bei der Implementierung des Sharedpreferences-Plugins. Durch Teamkommunikation konnten wir eine Lösung finden. Allerdings hat dieser Prozess mehr Zeit in Anspruch genommen als ursprünglich geplant, weil am Anfang nur ein Teammitglied an dem Problem gearbeitet hat.

Play-/Pause-Button

Zu Anfang der Entwicklung wurden für den Home- und den Radioscreen unterschiedliche Play Button als Platzhalter verwendet. Später musste dann jedoch derselbe Button verwendet werden, wie im Kapitel Die Umsetzung bereits erläutert. Dies brachte Probleme, da verschiedene Skalierungen benötigt wurden und es mehrere Möglichkeiten gab, diese Skalierung umzusetzen. Mehrere Personen haben dann an verschiedenen Stellen die Skalierung verändert. Das Problem konnte behoben werden, hätte jedoch durch bessere Kommunikation / Dokumentation vermieden werden können.

Prioritätenvergabe

Beim Implementieren der reorderableListView fehlten Anfangs die Icons in den Listenelementen, mit denen die Anordnung der Elemente verändert werden konnte. Dadurch war die Liste nicht funktional. Nach längerer Recherche wurde eine Lösung gefunden. Später stellte sich heraus, dass die Lösung auch in der Dokumentation zu der reorderableListView enthalten war. Aufmerksames Lesen der Dokumentation hätte das Problem also verhindern können.

Kommunikation

Insgesamt hätte eine bessere Kommunikation innerhalb unseres Teams, sowie zwischen unseren beiden Teams einige Probleme verhindern und zur schnelleren Lösung anderer Probleme beitragen können.

Auto-Scrolling Animation

Damit die Animation immer nur dann ausgeführt wird, wenn der Text breiter ist als der verfügbare Platz, haben wir einen Erkennungsalgorithmus hinzugefügt. Dieser Algorithmus war schwierig zu konfigurieren, sodass es mühsam war, die richtigen Parametereinstellungen zu finden. Infolgedessen kam es anfangs zu dem Problem, dass die Animation nicht gestartet ist, obwohl der Platz nicht ausreichend war.

Lightmode

Der Lightmode war schwierig zu implementieren, da er nicht zu Anfang eingeplant war. So wurde es nicht konsequent beachtet, die Farben über globale Variablen festzulegen. Dies musste für den Lightmode nachgeholt werden. Vor allem aber das Finden einer geeigneten Vorgehensweise zur Ermöglichung des Lightmodes bereitete Schwierigkeiten.

- Benachrichtigen weshalb gewechselt wird schwieriger als gedacht ->Erste Verbesserung: Icon für Radio mit/ohne Werbung
- Abspielen von Audio kann zu Exception führen
    - Wahrscheinlich, wenn keine Internetverbindung vorhanden

Backend

Fingerprinting

Die Wahl der Fingerprint Bibliotheken war sehr begrenzt und keine konnte einen Livestream unterstützen. Wir mussten dann eine Funktion schreiben, die den Livestream in Schnipseln fingerprinted, was viele Probleme verursacht hat:

Abbruch der Verbindung

Manche Radiosender haben die Verbindung nach ungefähr sechs Stunden zu uns abgebrochen. Seitdem wir von dem Problem wissen, starten wir den Stream nun stattdessen nach 5 Stunden und 50 Minuten neu. Diese Zeit ist in der .env-Datei konfigurierbar.

Fehlerhafte Fingerprints

Ungefähr alle zwei Stunden gibt es einen Fehler durch ffmpeg, dass die zu analysierende Audiodatei einen Fehler hat.
Da dies aber ein relativ seltener Fehler ist (alle zwei Stunden über alle sieben Radios hinweg), ist das größte Problem daran, dass diese Fehler in unserer Log-Datei dominieren. Der Fehler kommt wahrscheinlich daher, dass es hin und wieder zu Übertragungsfehlern kommt.

Richtige Einstellung der Confidence

Damit ein Audioschnipsel als Werbejingle erkannt wird, muss man eine Toleranz (“Confidence”). Am Anfang war die bei uns zu niedrig, später aber konnten wir einen guten Wert finden.

Doppelte Fingerprints

Da die Audioschnipsel sich jeweils überschneiden müssen, kann es passieren, dass zwei Mal schnell nacheinander ein Werbejingle erkannt wird. Da manche Radiosender auch einen Endjingle haben (der sich so anhört wie der Startjingle), toggeln wir den Status zwischen Werbung und Musik. Der Status wurde in solchen Fällen also schnell hin und her getoggelt. Als Lösung entschieden wir uns dafür eine Sperrzeit einzubauen.

Wann ist die Werbung zuende?

Nicht alle Radios haben einen Jingle, der das Ende der Werbung angibt. Daher hat jedes Radio ein Feld “ad_duration”, in dem festgelegt wird, für wie viele Minuten der Status bei “Werbung” verbleiben soll.

Saisonale Jingles

In den Radios RadioRST und AntenneAC gab es einen weihnachtlichen Jingle. Diesen Jingle mussten wir zusätzlich aufnehmen und zu den Fingerprints hinzufügen.

Performance Probleme

Die Bibliothek Dejavu, die wir zum Fingerprinting benutzen, benötigt ungeahnt viel Rechenleistung. Daher haben wir leider nur sieben Radios in unsere Liste aufgenommen.

Datenbanken

Wir haben uns für die PostgreSQL-Datenbank entschieden, da wir durch diverse andere Module dessen Vorteile gelernt haben.

Die Dejavu-Library benötigt leider eine MySQL-Datenbank. Zwar haben wir versucht einen Treiber für die Dejavu-Library zu schreiben, damit diese auf PostgreSQL laufen kann, und auch andersherum, unsere Codebase auf MySQL abzuändern, jedoch haben diese Bemühungen viel Zeit gekostet, ohne zu einem zufriedenstellenden Ergebnis zu kommen. Daher leben wir nun mit dem Kompromiss, zwei Datenbanken zu nutzen.

Metadaten

Wir besorgen uns die Metadaten (aktueller Song und Interpret des Radios) von der inoffiziellen API von radio.de. Leider bieten diese keine Daten zu den sogenannten NRW Lokalradios.

Wir haben uns dann darum gekümmert, eine inoffizielle API von den NRW Lokalradios direkt anzubinden.

Einzelaufwände

Frontend

Ahmad Fadi Aljabri

- Erstellung des Mockups für den Setting Screen
- Persistierung
    - Favoriten
    - zuletzt gehörtes Radio
    - Settings
    - Dark/Light mode
- Benachrichtigung wenn ein Radio angeklickt wird, wo Werbung läuft
    - Programmierung eines Dialogs
- Programmierung des Setting Screens
- Light/Dark Mode implementieren
- Benachrichtigung wenn die Verbindung zum Internet verloren geht
    - Snackbar-Benachrichtigung konfigurieren
- Recherche zum Verbindungsaufbau zum Server
- Dokumentation

Nicolas Harrje

- Erstellung des Mockups für den Radioscreen
- Programmierung der Navigationsleiste
- Programmierung des Radioscreens
    - Funktionalität der Steuerungsbuttons

        - Darstellung der Metadaten
- Anbindung des Servers
    - Verbindungsaufbau zum Server
    - Daten für Listendarstellung im Homescreen abfragen
    - Implementierung des Providers für die Listendarstellung
- Priorisierung
    - Initialisierung der Prioritäten
    - Persistierung der Prioritäten
    - Liste, deren einzelne Elemente per Drag-and-drop neu angeordnet werden können, um die Prioritäten neu anzuordnen
        - Listendarstellung abhängig vom Fluchtradios-Button (wenn ausgewählt, kann Liste neu angeordnet werden)
- Dokumentation

Nils Stegemann

- Erstellung des Mockups für den Homescreen
- Programmierung des Homescreens
    - Filteroptionen
        - Textuelle Suche
        - Fluchtradios
        - Aktuell Werbefrei
    - Liste aller verfügbaren Radios
        - Indikator Werbung / keine Werbung
        - Vergabe der Favoriten
        - Darstellung der Metadaten

    - Darstellung des gespielten Radios mit Play Button zum starten und stoppen des Audiostreams.
- Anbindung des Servers
    - Verbindungsaufbau zum Server
    - Kommunikation mit dem Backend, wenn der Client ein Radio streamen möchte.
- Programmierung der Audiowiedergabe
    - Wechsel der stream Url
    - Erstellung des Play Buttons zum starten und stoppen des Audiostreams.
- Programmierung einer Auto-Scrolling Animation, wenn der Text breiter als der verfügbare Platz ist.
- Persistierung der Favoriten und des zuletzt gehörten Radios
- Dokumentation

Backend

Chris Henkes

- Zeitbasierte Umschaltung der Radios zwischen Werbung und Musik
- Verbesserung der Performance der zeitbasierten Umschaltung
- Thread basierte Notifications an Clients senden
    - für das aktuelle aktive Radio
    - für alle in der App vorhandenen Radios
- Aktuelle Metadaten(Song und Interpret) der Radios mit Hilfe der Radio.de API erhalten und an Client weiterleiten
- Umschaltung zu Werbung der Radios anhand der jeweiligen Werbejingles
    - Bei Radios mit Werbe end Jingles, mit diesem zurück zum Radio schalten

    - Bei Radios ohne Werbe end Jingles, mit einer festgelegten ‘Werbezeit’ zurück schalten

Kaan Yazici

- Für Tests: Erstellung eines Konsolenprogramms, das über ähnliche, aber vereinfachte Funktionen wie die vom Frontend entwickelte App verfügt
- Korrekte Fehlerbehandlung in der Server Schnittstelle (Version 1)
- Analyse der Radios
    - Antenne AC, Radio RST, BAYERN 1, 100.5 Das Hitradio, 1Live, WDR2, BigFM
    - Aufnahme der Radios
    - Extraktion der Zeiten, in denen Werbung abgespielt wird
    - Extraktion der Zeiten, in denen Nachrichten abgespielt werden
    - Siehe Anlage “Recherche”
- Recherche zu einer geeigneten Python Bibliothek, mit der wir Acoustic Fingerprintings von Jingles erstellen und diese Jingles möglichst präzise aus dem Livestream erkennen können
- Extraktion und Nachbearbeitung der einzelnen Werbejingles, damit diese mit der Fingerprinting Technologie erkannt werden können
- Mitschneiden der Radiostreams und Anwendung des Fingerprintings (Version 1 und 2)
- Generelle Fehlerbehebung beim Fingerprinting

Maximilian Breuer

- Recherche zur Datenbank
- Recherche zu einem geeigneten Object Relational Mapping Framework (ORM)
- Design der Datenbank
- Umsetzung der Datenbank im ausgewählten ORM in Python

- Um die zwei Datenbanken MySQL und PostgreSQL auf eine Datenbank zu verschmelzen
    - Recherche zur Migration der PostgreSQL Datenbank auf MySQL
    - Umsetzung der Migration von PostgreSQL auf MySQL (leider gescheitert)
    - Recherche den MySQL Treiber der Fingerprinting Bibliothek auf einen PostgreSQL Treiber umschreiben
    - Umsetzung der Migration von MySQL auf PostgreSQL (leider gescheitert)
- Erstellung von vielen Hilfsfunktionen, die Lese-, Schreib- und Löschaktionen auf der PostgreSQL Datenbank vornehmen, damit Statements wiederverwendbar werden und das nötige Hintergrundwissen für das dahinter liegende ORM nur selten nötig war
- Dokumentation des Backend Codes mittels PythonDoc
- Recherche und Einsatz von Multithreaded Logging Funktionen, um Statusupdates, Informationen zur Server-Client-Kommunikation und Fehler auf der Konsole und in Dateien niederzuschreiben
- Logging der vom Server erkannten Werbejingles und der Radio Metadaten
- Fehlerbehebung für Datenbankfunktionen, Connection aufräumen und kontinuierliche Erweiterung des Loggings

Gerrit Weiermann

- Design und Umsetzung der Server-Schnittstelle
- Einrichtung des extern gehosteten Servers
- Verbesserung und Vereinfachung der Fehlerbehandlung in der Server-Schnittstelle
- Mitschneiden der Radiostreams und Anwendung des Fingerprintings (Version 3)
- Verbesserung der Performance des Fingerprintings durch Multiprocessing statt Multithreading (da GIL in Python sonst keine echte Parallelisierung zulässt)
- Recherche und Umsetzung des Umzugs auf Docker

- Anlegen einer globalen Konfigurationsdatei .env zur Steuerung aller Konfigurationen, die zuvor als hardgecodete Werte im Source Code versteckt waren
- Betreuung der Dokumentation
    - Planung der Dokumentenstruktur
    - Verantwortlich für die meisten Inhalte der Dokumente, insbesondere die Abschnitte:
        - Das Endergebnis (für Backend)
        - Die Umsetzung (für Backend)
        - Erfüllung der Mindestanforderungen
        - Weitere erreichte Ziele
        - Design der Datenbank
        - Server-Schnittstellendokumentation
        - Unsere Recherche (teilweise)
        - Umsetzung der Werbeerkennung
        - Verwendete Technologien > Backend
        - Getting Started: Backend

Anhänge

Im Folgenden sind weitere Dokumente angehangen:

Konventionen in der Frontend-Entwicklung

Schreibweise für Bezeichner

Variablen	camelCase
Klassen	PascalCase
Ordner	snake_case
Dateien	snake_case

Dart-Doc Dokumentation

- Doc-Kommentare nutzen: ///
- Libraries kommentieren
- Erste Zeile ist Zusammenfassung in einem Satz
- Leerzeichen nach erster Zeile
- Dinge, die offensichtlich sind, nicht kommentieren
- Funktionen oder Methoden aus dritter Person beschreiben (Starts…, Returns…)
- Nicht-Booleans mit einem Nomen beschreiben
- Booleans mit "Whether …” beschreiben
- Nur eine Dokumentation für Getter und Setter
- Klassen mit Nomen beschreiben
- Codebeispiele in Backticks ‘’’ setzen
- Bezeichner (Variablen, Klassen, Methoden, Funktionen, Exceptions etc.) in []
- Markdown kann zum Formatieren verwendet werden

Git

Für jedes Feature wird ein neuer Branch erstellt. Soblad das Feature erfolgreich getestet wurde, kann es in den Hauptbranch *next* gemerged werden.

Design der Datenbank

Abgesehen von der Fingerprinting Library, die für sich alleine in MySQL arbeitet, läuft alles in einer PostgreSQL Datenbank. Im Folgenden werden die einzelnen Tabellen genauer erklärt.

Tabelle: radios

Diese Tabelle repräsentiert jeweils ein von uns unterstütztes Radio.

Attribut	**Beschreibung**
id	Primärschlüssel
name	Name des Radios
status_id	Referenz auf die Statustabelle (ob Werbung, Musik, etc. läuft)
currently_playing	Falls gerade ein Song auf dem Radio läuft: Der Song Name
current_interpret	Falls gerade ein Song auf dem Radio läuft: Der Interpret
stream_url	Unter welcher URL der Client das Radio hören kann
logo_url	Unter welcher URL das Logo des Radios zu finden ist, das der Client anzeigen kann
ad_duration	Wenn es beim Radio einen Ende Jingle gibt: Wert 0 Sonst: Geschätzte Dauer, bis Werbung endet
ad_until	Falls keine Werbung läuft: null Sonst: Timestamp, wann die Werbung geschätzterweise endet
station_id	Die station_id für die Metadaten API (radio.de)
metadata_api	Eine Referenz für die Interna des Servers, welche API genutzt werden muss. Es gibt Radios, bei denen die radio.de API nicht funktioniert. **Bisher gibt es aber noch keine Umsetzung für andere APIs.**

Tabelle: radio_states

Zustände, die ein Radio haben kann. Beispiele: “Werbung”, “Musik”, “Nachrichten”. Für mehr Informationen unter der Server-Schnittstellendokumentation → Datenstrukturen → Status schauen.

Attribut	**Beschreibung**
id	Primärschlüssel

label	Name des Zustands

Tabelle: radio_ad_time (deprecated)

Diese Tabelle ist eine Altlast, da wir von Zeitschaltung auf Fingerprinting umgestiegen sind. Sie gibt die Uhrzeit an, wann Werbung auf einem bestimmten Radio geschaltet wird.

Da Radios je nach Uhrzeit mal halbstündlich, zur vollen Stunde oder gar nicht Werbung schalten, wird eine Zeitspanne (ad_transmission_*) angegeben, in der die Regel gilt.

Attribut	**Beschreibung**
id	Primärschlüssel
radio_id	Referenz auf das Radio, dessen Werbezeit hier angegeben wird
ad_start_time	Ab welcher Minute die Werbung startet (Datentyp: int)
ad_end_time	Mit welcher Minute die Werbung endet (Datentyp: int)
ad_transmission_start	Ab welcher Stunde die Regel gilt (Datentyp: int)
ad_transmission_end	Bis welche Stunde die Regel gilt (Datentyp: int)

Tabelle: connections

Eine Connection ist eine Abbildung eines Clients. Da der Client eine Suche starten kann und die Ergebnisliste vom Server laufend aktualisiert wird, muss der Server die Suchparameter hier abspeichern.

Da die eigentliche Suche abgeschafft wurde und immer eine ungefilterte Radioliste an den Client übergeben wird, sind die Attribute in der Tabelle hier nicht mehr in Benutzung (mit “deprecated” markiert)

Die Anzahl der Updates, die der Server dem Client selbstständig sendet, ist begrenzt. Die Anzahl noch folgender Updates ist unter “search_remaining_update” verzeichnet.

Attribut	**Beschreibung**
id	Primärschlüssel
search_query	Suchbegriff der aktuell laufenden Suche (deprecated)
search_without_ads	Filterung der Radioliste auf Radios, auf denen gerade keine Werbung läuft (deprecated)

current_radio_id	Welches Radio gerade vom Client gehört wird (kann null sein)
search_remaining_update	Wie viele Updates vom Server noch selbstständig gesendet werden.
preference_music	Ob der Client Musik hören möchte (wird im Moment vom Server ignoriert)
preference_talk	Ob der Client Gespräche bspw. zwischen Moderatoren hören möchte (wird im Moment vom Server ignoriert)
preference_news	Ob der Client Nachrichten hören möchte (wird im Moment vom Server ignoriert?)
preference_ad	Ob der Client Werbung hören möchte

Tabelle: connection_search_favorites (deprecated)

Many-to-many Mapping Tabelle, um zu einem Client die favorisierten Radios abzuspeichern, damit diese in der Suchfunktion gefiltert werden können.

Da die Funktionalität zur Filterung von Radios wieder entfernt wurde, ist diese Tabelle nur eine Altlast.

Attribut	**Beschreibung**
id	Primärschlüssel
radio_id	Referenz zum Radio, das favorisiert wurde
connection_id	Referenz zum Client, der diesen Favoriten wählte

Tabelle: connection_preferred_radios

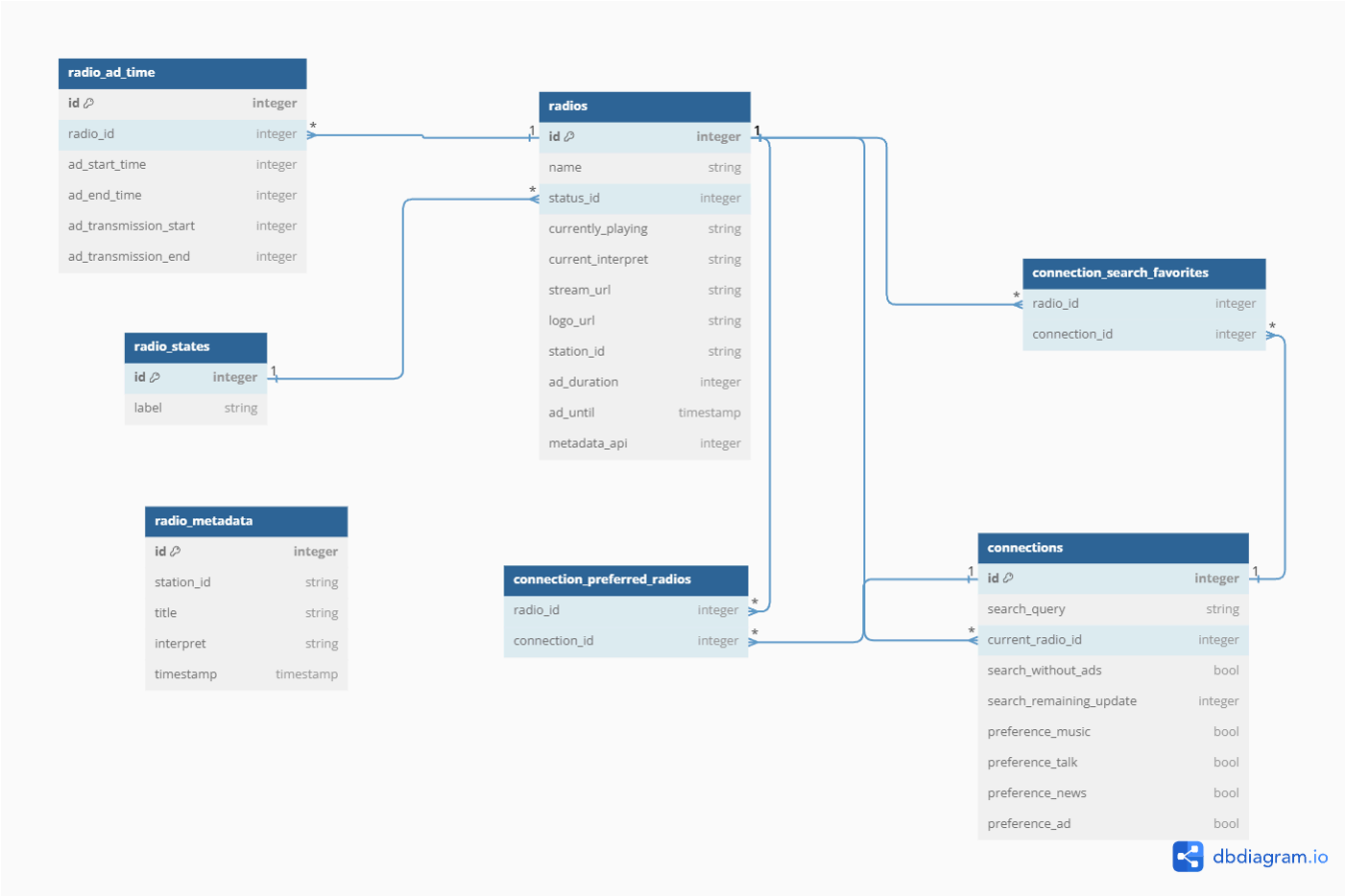
Wenn der Client gerade Radio hört, gibt er gleichzeitig an, welche Radios er priorisiert, um im Falle von Werbung auf alternative Radios auszuweichen.

Attribut	**Beschreibung**
id	Primärschlüssel
radio_id	Referenz zum Radio, das als Alternativradio festgelegt wurde
connection_id	Referenz zum Client, der die Präferenz wählte
priority	Priorität des Ausweichradios (je kleiner, desto höher priorisiert)

Tabelle: radio_metadata (deprecated)

Besteht nur für Loggingzwecke, wobei wir zum Loggen nun auf Logdateien umgestiegen sind.

Attribut	Beschreibung
id	Primärschlüssel
station_id	Parameter für die genutzte Metadaten API (radio.de), repräsentiert ein Radio
title	Repräsentiert den Songnamen
interpret	Repräsentiert den Interpreten
timestamp	Zu welchem Zeitpunkt die Metadaten von der API gelesen wurden



Server-Schnittstellendokumentation

Die Verbindung läuft über Websocket. Der Client Verbindet sich mit dem Endpunkt: ws://<host>:<port>/api

Danach sind ohne Authentifizierung sofort alle Optionen möglich. Während die Verbindung aufgebaut ist, speichert der Server notwendige Sitzungsdaten. Sobald der Client die Verbindung abbricht, werden diese aber wieder gelöscht.

Der Nachrichtenaustausch funktioniert über JSON-kodierte Objekte. Wie diese aufgebaut sein müssen, steht unter “Datenstrukturen”.

Auflistung der Radios

Um eine Liste aller Radios mit allen Informationen abzurufen, muss ein in JSON-kodierter SearchRequest gesendet werden (nähere Details unten). Der Server sendet daraufhin ein SearchUpdate.

Sobald sich in der darauffolgenden Zeit ein Detail in einem Radio aktualisiert (bspw. neuer Songname oder Status ist zu Werbung gewechselt), sendet der Server unaufgefordert die aktualisierte Liste an den Client.

Wie oft diese unaufgeforderten Updates gesendet werden, wird im SearchRequest im Attribut “requested_updates” konfiguriert. In jedem Update wird dem Client mitgeteilt, wie viele Updates noch folgen werden: “remaining_updates”.

Mit einem erneuten SearchRequest kann man wieder neue Updates anfordern: “requested_updates” überschreibt “remaining_updates”.

Starten eines Radiostreams

Möchte der Client ein Radio hören, so muss der Client dies dem Server mitteilen mit einem StreamRequest mitteilen. Der Server gibt ein StreamUpdateEvent, falls sich die Metadaten vom aktuell gehörtem Radio ändern (dann ist das Attribut “switch_to” mit dem selben Radio versehen) oder falls das aktuell gehörte Radio seinen Status gewechselt hat, was nicht mehr der PreferredExperience entspricht (bspw. Werbung). Im zweiten Fall muss der Client dann auf das Radio, das im Attribut “switch_to” genannt wurde, wechseln.

Datenstrukturen

Radio

Wird vom Server an den Client gesendet.

Attribut	Datentyp	Beschreibung
id	Integer	Eindeutige ID des Radios
name	String	Name des Radios
logo	String	URL zu dem Logo des Radios
stream_url	String	URL von der man die Audio des Radios streamen kann
currently_playing	String	Eines dieser Möglichkeiten: ● Name des aktuell gespielten Songs ● Name der aktuell gespielten Sendung ● Ein Nachrichtentext
current_interpret	String	Falls gerade ein Song gespielt wird: Die Interpreten des Songs. Sonst: leerer String
status_id	Integer	Referenziert den Status, den das Radio aktuell hat
status_label	String	Bezeichnung des aktuellen Status
ad_duration	Integer	Schätzung, wie viele Minuten das Radio Werbung am Stück spielt. Ausnahme: Wenn es einen Endjingle gibt, dann ist dieser Wert 0.
ad_until	String	Wenn keine Werbung abgespielt wird: null Sonst: Geschätzter Zeitpunkt, bis wann auf dem Radio Werbung läuft.
metadata_api	Integer	1: Wenn die Metadaten über radio.de abgerufen werden 2: Wenn die Metadaten über nrwlokalradio.com abgerufen werden

Status

Attribut	Datentyp	Beschreibung

id	Integer	ID des Status
label	String	Bezeichnung des Status

Vorhandene Status:

Status ID	Bezeichnung	Vom System umgesetzt?
1	Werbung	Ja
2	Musik	Ja
3	Nachrichten	Nein
4	<null>	Nein

SearchRequest

Anfrage des Clients, um die gesamte Auflistung aller verfügbaren Radios zu bekommen.

Attribut	Datentyp	Beschreibung
type	String	Muss auf “search_request” gesetzt werden
requested_updates	Integer	Anzahl der Updates, die der Server selbstständig senden soll, falls sich ein Radio seit dem vorherigen Update verändert hat

StreamRequest

Wird vom Client an den Server gesendet.

Attribut	Datentyp	Beschreibung
type	String	Muss auf “stream_request” gesetzt werden
preferred_radios	Integer[]	Geordnete Liste an Radio IDs. Sagt dem Server, welche Radios man hören möchte. Der erste Eintrag, der der preferred_experience (bspw. ohne Werbung) entspricht, gewinnt
preferred_experience	PreferredExperience	Welche Art von Inhalt man hören möchte (ob bspw. Nachrichten auch

		übersprungen werden sollen)

PreferredExperience

Ist eine Altlast. Unterstützt wird nur die folgende Konfiguration:

```
{
  "ad": false,
  "news": true,
  "music": true,
  "talk": true
}
```

Sendet man einen anderen Wert, ist das Verhalten des Servers undefiniert.

Attribut	**Datentyp**	**Beschreibung**
ad	Boolean	Ob der Client Werbung hören möchte
news	Boolean	Ob der Client Nachrichten hören möchte
music	Boolean	Ob der Client Musik hören möchte
talk	Boolean	Ob der Client Moderationen hören möchte

SearchUpdate

Wird vom Server an den Client gesendet.

Attribut	**Datentyp**	**Beschreibung**
type	String	Ist auf “search_update” gesetzt
radios	Radio[]	Liste aller Radios, die unterstützt werden
remaining_updates	Integer	Wie viele automatische Updates noch folgen werden

RadioStreamEvent

Wird vom Server an den Client gesendet.

Attribut	**Datentyp**	**Beschreibung**

type	String	Ist auf “radio_stream_event” gesetzt
requested_updates	Integer	Anzahl der Updates, die der Server selbstständig senden soll, falls sich ein Radio seit dem vorherigen Update verändert hat

Unsere Recherche

Dieses Kapitel ist dazu gedacht, unsere Überlegungen und Erkenntnisse festzuhalten. Hier kann man nachlesen, welche Fragestellungen und Probleme wir haben und aus welchen Gründen wir gewisse Entscheidungen gefällt haben.

Nutzung von Python

Python ist für seine analytischen Bibliotheken bekannt. Die Analyse von Audiostreams sollte durch diese Sprache also erheblich erleichtert werden. Daher wäre es von Vorteil, wenn DartJS mit Python kompatibel wäre.

Ansatz: Backend als Webservice durch Server anbieten

Es ist möglich über HTTP Request, dadurch kann das Frontend über Flutter/DartJS und das Backend über eine beliebige Programmiersprache entwickelt werden. (Muss nicht unbedingt Python sein!). Es wird dann aber eine Internetverbindung benötigt.

Quellen:

- https://stackoverflow.com/questions/64853113/how-to-integrate-flutter-app-with-python-code
- https://sites.google.com/view/geeky-traveller/app-development/flutter/how-to-link-python-script-py-file-with-flutter-app

Ansatz: Hybrid Python mit dem Modul “DartPy” integrieren

Das Modul “DartPy” ermöglicht die Interpretation von Pythoncodes (die als Strings bzw. Dateien in eine Funktion übergeben werden) und macht DartJs somit interoperabel mit Python. Das geht unter anderem dadurch, dass native C-APIs aufgerufen werden können.

Das Modul wird von einer Person verwaltet, hat 63 Stars, aktuell 5 von 9 Issues offen und hat seinen letzten Publish von vor 16 Tagen.

Der Autor gibt in einem Issue die Aussage, dass er nicht weiß, ob das Modul auch für Android klappt. Rein theoretisch sollte es aber gehen.
“It probably gets tricky with python libraries / python library paths”

Fazit: Es gibt nur wenig Informationen zu dieser Vorgehensweise. Man könnte ausprobieren ob es klappt, aber es könnte möglich sein, dass wir auf Probleme stoßen, die einen großen Rattenschwanz nach sich ziehen könnten (Pull Request auf die Bibliothek/Modul).

Quellen:

- https://www.geeksforgeeks.org/hybrid-programming-using-python-and-dart/
- https://pub.dev/packages/dartpy
- https://github.com/TimWhiting/dartpy/issues/8

Ansatz: Statt Python direkt DartJs nutzen

- Es wird Tensorflow Lite für KI Methoden angeboten.
- Zu Audio Fingerprinting konnte ich keine Bibliothek finden.

Quellen:

- https://pub.dev/packages/tflite_flutter

Ergebnis

Wir nutzen einen Server mit Python als Backend.

Erkennungsmechanismen

Basierend auf der Quelle https://www.adblockradio.com/en/

Ansätze nach “Designing an audio adblocker for radio and podcast”

- **Lautstärke**:
  In Radios mit klassischer Musik sei die Lautstärke von Werbung unterschiedlich zum restlichen Programm.

  Werbung würde oft mit acoustic compression verarbeitet werden, um lauter zu sein.
  Fazit: Nicht anwendbar

- **Zeit**:
  Werbung und Moderation seien oft zu speziellen Zeiten, jedoch nicht unbedingt sekundengenau.
  Fazit: Darauf basiert unser MVP, sollten wir ausprobieren

- **Metadata**:
  Die mitgegebenen Informationen über die aktuelle Sendung/Song könnte Aufschluss über mögliche Werbung geben, jedoch sei diese oft nicht sehr genau
  Fazit: Kaum anwendbar, möglicherweise für eine ungefähre Verifikation anderer genauerer Mechanismen

- **Durch Mensch**:
  Der User kann Werbung durch Klick auf einen Button markieren.
  Fazit: Könnte man als Erweiterung nutzen, aber nicht als alleiniges Mittel

- **Spracherkennung**:
  Durch NLP könnten Werbesprüche als solches erkannt werden.
  Fazit: Die Idee wurde vom Autor verworfen, jedoch als möglicherweise erfolgversprechend deklariert.
  Eigene Vermutung: Sekundengenau wird hier die Werbung auch nicht getroffen

- **Acoustic Fingerprints**:

  Bereits bekannte Werbung bekommt einen Fingerabdruck. Bereits bekannte Werbung kann so zuverlässig erkannt werden (jedoch nicht dessen Variationen)
  Fazit: Sollten wir auf jeden Fall nutzen!

  Update: Wurde umgesetzt

- **KI**:
  Klassifizierung basierend auf Muster (vgl. “Lautstärke”) führe zu 95% Präzision (mehr Details auf dem Blogpost)
  Fazit: Sehr gute Idee, vor allem in Kombination mit Fingerprints. Benötigt aber auch Trainingsdaten und viel Fachwissen für die Umsetzung.

  Update: Wurde nicht umgesetzt

Eigene Ansätze

- **Musikerkennung durch Fingerprints**:
  Als Erweiterung zu den bereits vorgeschlagenen Ideen, könnte man Acoustic Fingerprints gegen die Spotify Datenbank vergleichen, um Songs als (eindeutig) “nicht Werbung” zu klassifizieren.

  Update: Wurde nicht umgesetzt

Erkenntnisse über Werbungen

Radio	Werbezeiten	Werbelänge [Sekunden]	Stündliche Variation	Jingle vorhanden?	Weiteres
WDR2	6 - 22 Uhr oder bis 20 Uhr	10-60	nicht exakt	Ja (Anfang)	1-2 Werbeblöcke pro Stunde
1Live	6 - 22 Uhr	10-60	1-3 Minuten	Ja (Anfang)	
100,5 Hitradio	immer	10-60	unregelmäßig	Ja (Anfang)	
AntenneAC	immer	10-60	-	-	
bigFM	immer	60-120	unregelmäßig	Ja (Anfang und Ende)	Oft eigene Werbung

Privatradiogesetz

Für Radiosender, die 24/7 senden:

- Täglich maximal 206 Minuten Werbespots
- Jahresdurchschnitt maximal 172 Minuten Werbespots

Werbungsarten

(Auszug aus Recherche, wie man selbst Werbung in Radios schalten kann)

Spotart	Spotlänge	Zusatzinformation
Klassischer Radiospot	15-30 Sek.	Die klassischen Spotkampagnen dienen insbesondere dazu, spezielle Aktionen zu bewerben.
Single-Spot	mind. 30 Sek.	Der Single-Spot wird nicht als Teil des Werbeblocks gesendet, sondern isoliert im laufenden Programm gespielt. Der Sendeplatz wird maximal einmal pro Stunde vergeben.
Tandem Spot (Spot+Reminder)	Spot: ca. 15-30 Sek. Reminder: ca. 5-10 Sek.	Bei einem Tandem Spot handelt es sich um eine identische oder ergänzenden Werbebotschaft, die meist aus zwei oder mehreren Spots besteht. Die Spots sind durch einen anderen Spot oder ein Programmelement unterbrochen, folgen aber nahezu aufeinander.
Eckplatzierung	15-30 Sek.	Der Eckspot überzeugt durch seine besondere Platzierung direkten Beginn oder am Ende eines Werbeblocks und erzielt dadurch besondere Aufmerksamkeit.
Webradio	15-30 Sek.	Der klassische Radiospot wird auf dem Streaming-Angebot von UKW-Radiosendern oder reinen Internetradios abgespielt. Jetzt informieren

Fingerprinting Bibliotheken für Python

PyDejavu

- Analysiert Audiodatei und erstellt selbstständig Fingerabdrücke und speichert diese in eine MySQL Datenbank
- Kann auch nativ eine Erkennung über das Mikrofon starten
- Umsetzung einer Live-Erkennung
    - Livestream für kurze Intervalle aufnehmen
    - https://stackoverflow.com/questions/37497107/record-and-recognize-music-from-url
- Als Docker Image und Python-Bibliothek verfügbar
- Repository ist veraltet: Letztes Update im Jahr 2015
    - Aber aktuellere Forks verfügbar
    - Beispiel: https://github.com/worldveil/dejavu (aber auch schon 4 Jahre alt)
- Eigene Angaben über die Genauigkeit:

Number of Seconds	Number Correct	Percentage Accuracy
1	27 / 45	60.0%
2	43 / 45	95.6%
3	44 / 45	97.8%
4	44 / 45	97.8%
5	45 / 45	100.0%
6	45 / 45	100.0%

audiomatch

- https://github.com/unmade/audiomatch
- Erkennung von Audio Dateien

- Als Docker Image und Python-Bibliothek verfügbar
- Audiodateien sollten mindestens 10 Sekunden lang sein
- Relativ aktuell: Letztes Update 2022

Findit

- https://github.com/methi1999/Findit
- Funktioniert erst ab Python 3.x.x
- Eher veraltet: Letztes Update 2019

Jamaisvu

- https://github.com/CwbhX/Jamais-Vu
- Erweiterung zu Dejavu
- Eher veraltet: Letztes Update 2019
- Gibt Details über den Song (zB. Artist, Album, etc.)
    - API Anbindung findet dafür im Hintergrund statt
    - Benutzereingaben bei fehlerhaften Ausgaben möglich
- Schnelle Erkennung
- Geeignet für Radioshows

audiophile

- https://github.com/bshankar/audiophile
- hört audio und erstellt die fingerprints selber
- Veraltet: Letztes Update 2017

fingerprint-pro-server-api-python-sdk

- https://github.com/fingerprintjs/fingerprint-pro-server-api-python-sdk
- Unterstützt Python 2.7 und ab Python 3.4.x
- Sehr aktuell: Letztes Update 2023

Performance für das Fingerprinting verbessern

In der ersten Version des Fingerprintings haben wir für jeden Radiostream einen Thread erstellt, der seine eigene Analyse ausführt.

Uns ist aufgefallen, dass die Fingerprint Analyse sehr viel Rechenzeit benötigt und für die Analyse von 5 Sekunden Audio ca. 2 Sekunden Rechenzeit pro Kern benötigt.

Da es keinen Sinn macht, auf einem Kern mehrere Analysen gleichzeitig auszuführen, wollten wir die Ausführung der Analyse auf eine pro Kern gleichzeitig begrenzen.

Wir änderten das Konzept so um, sodass jedes Radio einen Aufnahme Thread bekommt. Die Threads nehmen von dem Radio einzelne Audio Schnipsel auf und leiten diese als Aufgabe an Worker Threads weiter. Ein Worker Thread nimmt sich einen Schnipsel aus der Job Queue und wendet die Fingerprint Analyse darauf an. Da wir nur so viele Worker Threads wie Kerne gestartet haben, bekamen wir das Problem besser in Griff.

Nach Recherche ist uns jedoch aufgefallen, dass Python mit einfachen Threads keine echte Parallelität zwischen Kernen herstellen kann. Dafür solle man stattdessen das Modul “multithreading” nutzen. Eine mögliche Implementierung mittels Prozessen kann man im Branch “better_fingerprint_parallelization” (https://git.fh-aachen.de/addblockergroup/backend/-/blob/better_fingerprint_parallelization/api/fingerprinting.py?ref_type=heads).

Metadaten der Radios

In diesem Abschnitt beschäftigen wir uns mit dem Songnamen und dem Interpreten des Songs, die auf einem Radio abgespielt werden.

Radio.de

Bei der Recherche, wie radio.de ihre Metadaten auf der Webseite darstellen, ist uns aufgefallen, dass sie REST-API dafür entwickelt haben. Diese kann man wie folgt nutzen:

https://prod.radio-api.net/stations/now-playing?stationIds=<station_id1>,<station_id2>,...

Im GET-Parameter “stationIds” kann man , getrennt alle Radios eingeben, zu denen man die Metadaten haben möchte. Die Antwort der REST-API ist dann ein JSON-Dokument mit einer Auflistung dieser Metadaten. Wie die stationIds jeweils aussehen haben wir uns dann für jedes Radio händisch herausgesucht.

Radio	stationId (Radio.de)
1Live	1live
WDR 2	wdr2
100,5 Das Hitradio	dashitradio
BAYERN 1	bayern1main
BremenNext	bremennext

Die API nur einen “title” mit und wir müssen Songname und Interpret uns selbst dort heraussuchen. Leider wird dieser Titel von jedem Radio anders formatiert. Unser Backend setzt daher mehrere Strategien ein, um dies umzusetzen.

Radio	Format
1Live	<Interpret> - <Songname>
WDR 2	<Interpret> - <Songname>
100,5 Das Hitradio	<Interpret> - <Songname>
BAYERN 1	<Interpret>: <Songname>
BremenNext	<Interpret> - <Songname>

NRW Lokalradios

Uns ist dann aufgefallen, dass die NRW Lokalradios wie AntenneAC und RadioRST in der API nicht umgesetzt sind. Daher haben wir nach einer weiteren API Ausschau gehalten und die von NRW Lokalradios selbst gefunden:

https://api-prod.nrwlokalradios.com/playlist/current?station=<station_id>

Hier muss pro Radio eine separate Anfrage gesendet werden. Die IDs musste man wieder händisch heraussuchen.

Radio	station (NRW Lokalradios)
AntenneAC	18
RadioRST	41

Umsetzung der Werbeerkennung

Auf dem zuvor zurate gezogenen Blogartikel wurden mehrere Optionen genannt, wie das Backend den Service bereitstellen kann. Aufgrund von rechtlichen und Performance Gründen macht es wenig Sinn, dass die Audio direkt gestreamt wird, sondern nur die Metadaten, die aussagen, was gerade im Radio (Musik, Talk, Werbung) läuft und zu welchem Radio ggf. gewechselt werden müsste.

Zeitschaltung

Die Zeiten für Werbung der einzelnen Radios werden von uns händisch ermittelt (siehe oben “Erkenntnisse über Werbungen”).

Diese werden dann in eine Tabelle in der Datenbank eingefügt.

Ein separater Thread stellt eine Anfrage an die Datenbank, wann das nächste Mal ein Wechsel von Werbung auf Musik, bzw. Musik auf Werbung ist und legt sich bis dahin schlafen. Beim Aufwachen werden dann alle Clients benachrichtigt, die benachrichtigt werden müssen.

Wurde nun vom Fingerprinting vollständig abgelöst.

Synchronisation Client und Server

Es wird aber auch herausgestellt, dass der Client und Radiostream miteinander gesynct werden müssten, da der Audiostream teils unterschiedliche Verzögerungen hat. Dabei könnte ein Störfaktor sein, dass der Stream für das gleiche Radio verschiedene Werbesequenzen verteilt.

Wurde von uns nicht umgesetzt.

Puffern des Streams

Der Client und der Server sollten die Audio für mindestens ein paar Sekunden puffern, damit früh genug erkannt werden kann, ob Werbung gespielt wird und der Wechsel rechtzeitig stattfinden kann.

Idee: Aufgrund der Wartezeit wird ein Slogan der eigenen App abgespielt “ADBLOCK RADIO die App um Werbefrei zu streamen”

Wurde von uns nicht umgesetzt.

Das KI-Modell

Dem Blogartikel zufolge wurden von dem Autor 4 Sekunden Chunks in 1 Sekunden Abstand aufgenommen, in eine 2D Map, die die Audio repräsentiert, umgewandelt und mit einem Long Short-Term Memory Recurrent Neural Network analysiert. Die Analyse ist zustandslos.

Eine mögliche Verbesserung könnte dem Autor nach eine zustandsbehaftete Analyse sein, die die Wahrscheinlichkeit eines Inhaltwechsels vorhersagt (empfohlen werden hier Hidden Markov Models).

Haben wir nicht näher verfolgt.

Der Fingerprinting Algorithmus

Acoustic Fingerprinting erzeugt zu einem Audioschnipsel einen Fingerabdruck, der in einer Datenbank gespeichert werden kann. In unserem Fall haben wir aus den Radios die Werbejingles herausextrahiert und zu solchen Fingerabdrücken umwandeln lassen.

Der Server hört sich jedes Radio an und analysiert mit der Acoustic-Fingerprinting-Methode, ob gerade ein Werbejingle kam. Der Abgleich mit der Datenbank funktioniert selbst bei einem riesigen Datensatz sehr effizient.

Für das Fingerprinting der Radios wird ein Fork von der Bibliothek von Dejavu (https://github.com/worldveil/dejavu) verwendet. Da die Bibliothek nicht das Fingerprinting von Live Streams unterstützt aber jedoch von Audiodateien (FileRecognizer), zeichnen wir jedes Radio in kleinen Schnipseln auf und fingerprinten sie dann mit dem FileRecognizer der Bibliothek. Bei der Radioaufnahme werden die Schnipsel überlappend aufgenommen, da es zur Nichterkennung kommen kann, wenn ein Jingle abgeschnitten in zwei Schnipseln liegt. Diese Überlappung geschieht mit dem Parameter “Offset”, der die Anzahl der Sekunden bestimmt, die vom letzten Schnipsel in den nächsten mit übernommen werden sollen.

Anforderungen an den Server

Nach dem Blogartikel reicht ein Raspberry PI A oder B für die Analyse eines einzelnen Streams aus. Für mehrere Streams gleichzeitig wird mindestens der RPI 3B bzw. 3B+ empfohlen.

Ein RPI 3B+ hat einen vier Kerne Prozessor mit 1,4 GHz Taktrate (Quelle).

Mehr haben wir uns nicht mehr damit beschäftigt. Am Ende kam heraus, dass wir auf unserem 4-Kerner sieben Radios gleichzeitig analysieren können.

Weitere Ideen

- Ein Puffer von 10 Minuten, sodass man Werbungen vorspulen/überspringen kann. Der Puffer hält laut Blogartikel ca. 1-2 Stunden, bis er aufgebraucht wurde.

Verwendeten Technologien

Frontend

- Figma zur Erstellung der Mockups
- Flutter zur Erstellung der App
- Material Design als verwendete Designsprache
- Android Emulator zum Debuggen
- scrcpy zum Spiegeln des Smartphones auf den Rechner, um die App unter realistischen Bedingungen zu testen

Backend

- Python 3.13
- Flask als Webserver
- SQLAlchemy als ORM für die Datenbank
- PyDejavu für die Nutzung von Fingerprints
- PostgreSQL als Datenbank
- MySQL als Datenbank für die Fingerprints

Getting Started: Frontend

Projekt aufsetzen

Der Endbenutzer muss lediglich die APK herunterladen und installieren.

Für den Fall, dass ein anderes Team diese weiterentwickeln möchte, sind folgende Punkte zu befolgen

- Flutter installieren
- Projekt von GitLab klonen
    - git@git.fh-aachen.de:addblockergroup/frontend.git
- Abhängigkeiten aktualisieren
    - flutter pub get
    - flutter pub upgrade

Im Sourcecode zurechtfinden

Pfad	**Erklärung**
/lib/model	Hier befinden sich Datenmodelle, die die Datenstrukturen der App definieren.
/lib/provider	Hier werden alle Provider definiert.
/lib/screens	Hier werden separate Dateien für verschiedene Bildschirme der App definiert: ● Homescreen ● Radioscreen ● Settingsscreen
/lib/screens/home/filter	Hier liegen Dateien, die für die Filterfunktionen zuständig sind
/lib/screens/home /radio_list	Hier ist die Funktionalität der Listenansicht des Homescreens implementiert.
/lib/services	Hier werden alle Dienste, die in der App verwendet werden, definiert. ● Persistierung ● Audiowiedergabe ● Serveranbindung
/lib/services /websocket_api_service	Hier ist die Verbindung zum Server implementiert.
/lib/shared	Hier befinden sich gemeinsam genutzter Code und Ressourcen. ● colors (alle verwendeten Farben) ● Autoscrolling Animation (wird nicht verwendet) ● Button für die Audiowiedergabe ● Custom Liste Tile
main.dart	Hier ist der Einstiegspunkt. Diese Datei wird ausgeführt.

Getting Started: Backend

Server starten

- Installation von Docker
- Projekt von GitLab klonen
- In das geklonte Verzeichnis hineingehen
- Den Server über Docker starten:

```
$ docker compose up -d
```

Der Server wird dann auf dem Port 8080 gehostet.

Server konfigurieren

Im Projektverzeichnis befindet sich eine .env Datei. Dort gibt es viele Konfigurationsmöglichkeiten, die auch jeweils mit einem Kommentar erklärt werden. Für die Vollständigkeit ist hier der Dateiinhalt:

```
# External port exposed by docker itself
PORT=8080

# Fingerprinting takes approx. 2 seconds per cpu thread/core.
# We figured that count should be approximately = cpu cores
FINGERPRINT_WORKER_THREAD_COUNT=4

# At least x of all given fingerprints need to match
FINGERPRINT_CONFIDENCE_THRESHOLD=20

# Length of one recorded piece thrown into the fingerprinter
FINGERPRINT_PIECE_DURATION=5s

# How much delay the client needs to not hear any ads
CLIENT_BUFFER=10s

# Overlap between recordings (to prevent that ads can be intersecting)
FINGERPRINT_PIECE_OVERLAP=1s
```

```
# It's quite often that ads get detected twice because of the overlapping. To prevent that we want it to skip
FINGERPRINT_SKIP_TIME_AFTER_AD_START=10s

# To prevent that an end jingle doesn't toggle status back to status 'ad' after the fallback timer hit (AD_FALLBACK_TIMEOUT)
FINGERPRINT_SKIP_TIME_AFTER_ARTIFICIAL_AD_END=5min

# The minimum size that is required for recording a piece. If an audio stream doesn't provide that, it will restart
FINGERPRINT_PIECE_MIN_SIZE=10kb

# Timeout for retrieving the audio stream
STREAM_TIMEOUT=10s

# Some radios stop working after like 6 hours. We want to restart those streams beforehand
STREAM_AUTO_RESTART=5h 50min

# Some radios have an end jingle. But if that's not happening somehow, we fallback after this given time:
AD_FALLBACK_TIMEOUT=6min

# Any details about radios and their stats are stored in postgres. Here are the connection details
POOL_SIZE=5           # Amount of idle connections to the database that are held by the python server
MAX_CONNECTIONS=50    # The maximum of connections that may access the database simultaneously. If it's full theres a 'waiting list'
# Used by docker, so there's nothing to worry about:
CORE_POSTGRES_HOST=core-db
CORE_POSTGRES_USER=postgres
CORE_POSTGRES_PASSWORD=postgres
CORE_POSTGRES_DB=adblock_radio

# Fingerprints are stored in mysql. Here are the connection details
# Used by docker, so there's nothing to worry about:
FINGERPRINT_MYSQL_HOST=fingerprint-db
FINGERPRINT_MYSQL_USER=root
FINGERPRINT_MYSQL_PASSWORD=root
FINGERPRINT_MYSQL_DB=fingerprint
```

Im Sourcecode zurechtfinden

Pfad	**Erklärung**
/src/api	Hier sind die Codedateien, die sich mit der Kommunikation mit dem Client beschäftigen
/src/api/routes	Hier werden die URLs definiert, die ein Client besuchen kann. Unter anderem aber auch der Websocket-Endpoint
/src/backend	Hier ist der wesentliche Code, der sich um die Analyse der Radios beschäftigt: ● Fingerprinting ● zeitbasiertes Umschalten (wird nicht mehr verwendet) ● Metadaten aus verschiedenen APIs zusammentragen
/src/db	Hier sind alle möglichen Codedateien, die sich mit der Datenbank beschäftigen
/src/db/inserts	Hier stehen die Daten, die beim aller ersten Start in die Datenbank geschrieben werden
/src/error_handling	Hier sind generische Funktionen, um Fehler abzufangen und Anfragen des Clients zu verifizieren
/src/main.py	Hier ist der Einstiegspunkt. Diese Datei wird ausgeführt
/simple_client/Client.py	Ein Client-Programm, das im Terminal läuft. Wir haben das benutzt, um nicht vom Frontend-Team abhängig zu sein
/logs/backend.log	Alle Logs, die der Server produziert
/logs/adtime.csv	Eine Logdatei, die mitschreibt, wann sich der Status eines Radios ändert. Daraus könnte man später statistische Auswertungen produzieren.
/logs/metadata.csv	Eine Logdatei, die mitschreibt, welcher Song auf welchem Radio gerade gespielt wird. Hieraus kann man ebenfalls statistische Auswertungen produzieren und ggf. sogar Spotify Playlisten für bestimmte Radios erstellen
/fingerprint_audio_files	Hier sind die händisch extrahierten Werbejingles abgespeichert. Neben den Werbejingles gibt es hier auch News Jingles, die bisher aber noch keine Verwendung im Projekt gefunden haben

Ein neues Radio hinzufügen

1. Besorgen der URL für den Radiostream
   Wichtig: Es sollte im mp3-Format sein.
2. Besorgen der URL für das Logo des Radios
3. Falls es ein NRW Lokalradio ist:
   a. `metadata_api = 2`
   b. Auf die Webseite des Radios gehen und in den Entwicklertools > Netzwerkanalyse und den API-Call finden, bei dem die Metadaten abgerufen werden.
   c. Dort steht eine entsprechende Id für das Radio:
      `station_id = <id>`
4. Falls es ein Radio ist, dessen Metadaten auf Radio.de angezeigt werden können:
   a. `metadata_api = 1`
   b. Auf die Webseite von radio.de gehen und in den Entwicklertools > Netzwerkanalyse und den API-Call finden, bei dem die Metadaten abgerufen werden.
   c. Dort steht eine entsprechende Id für das Radio:
      `station_id = <id>`
5. Falls es einen Endjingle gibt
   a. `ad_duration = 0`
6. Falls es keinen Endjingle gibt
   a. Überlegen, wie lange auf dem Radio durchschnittlich Werbung läuft
   b. `ad_duration = <Anzahl Minuten>`
7. Werbejingle aufnehmen
   a. Der Werbejingle muss auf 1 Sekunde gekürzt werden
   b. Als Datei in den Ordner fingerprint_audio_files/AD_SameLenghtJingles ablegen
8. Eintrag in src/db/inserts/radios.json hinzufügen:

```
{
      "id": <Neue Id>,
      "name": <Name des Radios>,
      "status_id": 2,
      "currently_playing": null,
      "current_interpret": null,
      "stream_url": <URL des Radiostreams>,
      "logo_url": <URL des Logos>,
      "station_id": <station_id>,
      "ad_duration": <ad_duration>,
```

```
      "ad_until": null,
      "metadata_api": <metadata_api>
}
```

9. Gegebenenfalls die Anzahl der Fingerprint Workers in der .env Datei erhöhen:

```
FINGERPRINT_WORKER_THREAD_COUNT = <Anzahl>
```