A Framework for Developing Custom Live Streaming Multimedia Apps

Advances in Software Engineering, Education, and e-Learning pp 657-670 | Cite as

- Abdul-Rahman Mawlood-Yunis (1) Email author (amawloodyunis@wlu.ca)

1. Physics and Computer Science Department, Wilfrid Laurier University, , Waterloo, Canada

Conference paper
First Online: 09 September 2021

Part of the Transactions on Computational Science and Computational Intelligence book series (TRACOSCI)

Abstract

The rise in the number of mobile-connected devices and the emergence of new streaming models, such as on-demand, on-the-go, and interactive streaming, has an impact on the viewing practice of consumers. Mobile devices will play a major role in new viewing habit changes. In this paper, we present a generic framework for developing custom live streaming multimedia apps for mobile devices, i.e., we identify the principal characteristics of mobile live streaming apps, the components, components' interactions, and the design decision needed to create such apps. To demonstrate how the generic framework can be used, we created an app for live streaming audio for Android devices using URLs. The app acts as an instance of the framework and validates it. The paper has two main contributions: (1) the generic framework which can be adapted when developing live streaming multimedia apps or similar apps and (2) developers and end users can reuse the instance app to their own specific needs using live streaming URLs.

Keywords

Modeling Application architecture Mobile application Android MediaPlayer Digital streaming

1 Introduction

The rise in the number of mobile-connected devices and the emergence of new streaming models, such as on-demand, on-the-go, and interactive streaming, has an impact on the viewing practice of consumers. Mobile devices will play a major role in new viewing habit changes [7, 9]. In this paper, we present a generic framework for developing custom live streaming multimedia apps for mobile devices, i.e., we identify the principal characteristics of mobile live streaming apps, the components, components' interactions, and the design decision needed to create such apps. The framework can help developers in different ways. It can be used as a starting point for designing live streaming app architecture, identifying the list of components needed to develop custom live streaming apps, distributing work among the development team, assessing the usability of individual components, and creating domain vocabularies enabling discussions and understudying among developers.

To demonstrate how the generic framework can be used, we build an app for live streaming audio for Android devices using URLs. The app acts as an instance of the framework and validates it. It is also beneficial to both developers and end users. Developers can reuse the instance app structure to create and publish new apps, and end users can customize the app to their own specific needs using live streaming URLs. For example, users can group their favorite music and news stations in one place and easily play and switch from one station to another. The app turns the user's device into a radio and lets them listen to live streaming stations while in the office, on the road, or in any other setting. The app is like Spotify but on a smaller scale (that's probably all you need; your favorite station and not all the stations that come with Spotify).

The paper has two main contributions: (1) the generic framework which can be adapted when developing live streaming multimedia apps or similar ones and (2) using new URLs, customizable live streaming apps can be created. The app's source code and complete documentation can be used by instructors to teach various mobile topics. The source code and complete documentation are available upon request.

This paper is organized as follows: in Sect. 2, we describe the generic framework components and their functionalities; in Sect. 3, we present the framework architecture and its uses; in Sect. 4, we describe an instance of the framework, the functionalities, and the trade-offs using different components and approaches to create an instance app; and in Sect. 5, we conclude the paper and describe some future works.

2 A Framework for Multimedia Live Streaming Apps

The framework is made of 11 key constructs or components. The components are user interface, background process or service, communication channel, media player, power management, wake lock, thread, file management, URL, data storage, and network permissions. The ten components comprise the minimum components required for any live streaming framework. In the following, we describe each of these model components, i.e., we discuss the functionality and the trade-offs using different components and approaches. We also present the class diagram, and the architecture, of the framework.

2.1 App Interface

The app interface is one of the main components of the framework. It is required for (1) interacting with the user; (2) communicating with the background process, for example, to start and stop the content player component running in the background; and (3) listening to the message broadcast from the background process.

2.2 Background Process

A background process is needed to perform long-running tasks, i.e., playback streams in the background with no graphical user interface. Once the process starts, it might continue running even if the original application is ended or the user moves to a different application. The run continuity is platform and setup dependent. The background process is the right choice to use when the process does not interact with the user, i.e., is not the forefront process. The process can be defined as private or public. When private, it is usable only by the app it belongs to; however, when public, it is usable by other apps, i.e., other apps can start process using process API (application programming interfaces).

2.3 Communication Channel

A communication channel component is needed to receive and handle broadcast messages sent back and forth between foreground and background processes via method calls. The communication channel component does not need to have a user interface, but it can create notifications to alert the user when a broadcast event occurs. The communication channel object needs to be instantiated and registered to process the broadcasted messages on arrival.

2.4 Content Player

A media content player component, such as Android Media Player, VideoView, or ExoPlayer, is needed to control the playback of multimedia streams. The media player needs to be prepared, started, and ultimately released. There are two ways for the player to enter the prepared state: *synchronous* or *asynchronous* way. The difference between the two is in what thread they are executed. The *synchronous* one executes in the foreground thread or the user interface thread, and the *asynchronous* one is executed in the background thread. To avoid users getting an ANR (application not responding) message, the asynchronous way needs to be used for playing the live data over stream. Additional actions might need to be taken in case the media player content is not prepared instantly. For example, a listener object can be set for informing when the media player can start.

2.5 Power Manager

If mobile devices go into a low-power state, it will prevent apps from running. To control the power state on devices, you need to use power management to keep the CPU running, prevent screen dimming or going off, or prevent the backlight from turning on.

2.6 Network Lock

Acquiring the Wi-Fi lock keeps the connection on until the application releases the lock. Live streaming apps need to keep the Wi-Fi component awake; hence, you need to acquire the lock. A design decision needs to be made whether to keep the app running even when the device screen is off.

2.7 Threads

Preparing media content asynchronously is not enough to avoid ANR prompts and your application hanging when playing live streaming multimedia apps. The best solution is to run the media content player instance in its thread, i.e., run Media Player in a separate thread within the service. The code snippet below shows one way how this step can be achieved when Android is used.

```
@Override
onPrepared(MediaPlayer mp) {
new Thread(new Runnable() {
volatile boolean running = true;
public void run() {
try {
if (null != MainActivity.url) {
if (!player.isPlaying()) {
player.start();
bufferingComplete();
}
}
} catch (Exception e) {
player.reset();
}
```

```
} }).start();
}
```

2.8 Data Storage

Storing data and operations such as reading, writing, updating, viewing, and deleting data are needed for almost all kinds of apps. Data can be saved in the database, in local files, or in the cloud and in different formats. For live streaming multimedia-type apps, the data most likely is limited to network URLs and icons of stream providers, user preferences, and settings.

2.9 URLs

When apps come with built-in streaming services such as a preset list of radio stations, TV channels, and live streaming events, the stream provider names the URLs and icons needed to be saved or referenced at the URL component. This enables easy app extensions and changes. For example, if you want to add a new station to your preset list, then you simply add the new station to the list of existing stations. There is no need to change other parts of the code.

2.10 User Permissions

For an app to read, write, update, view, or delete data, it must have permission from the user to do so. Similarly, for an app to have access to Wi-Fi and network, it needs the user's permission. The request for these permissions needs to be included in the app's code for the app to operate.

2.11 Other Components

There are other components live streaming apps might require. For example, apps might need to handle which app has *audio focus* when more than one app plays audio to the same output device. Here, the audio focus is not considered. This is because some systems, e.g., Android, have a built-in solution for the audio focus [1], meaning, when a second app requests audio, the currently running one pauses playing or lowers its volume to give the second one audio focus.

3 Framework Class Structure

In Fig. 1, all the framework components are put together, and the class diagram, i.e., the framework architecture, has been created. The class diagram shows the components and the dependency between model components. When not identified, the cardinality relationship between components is one-to-one, and the relationship type is an association.

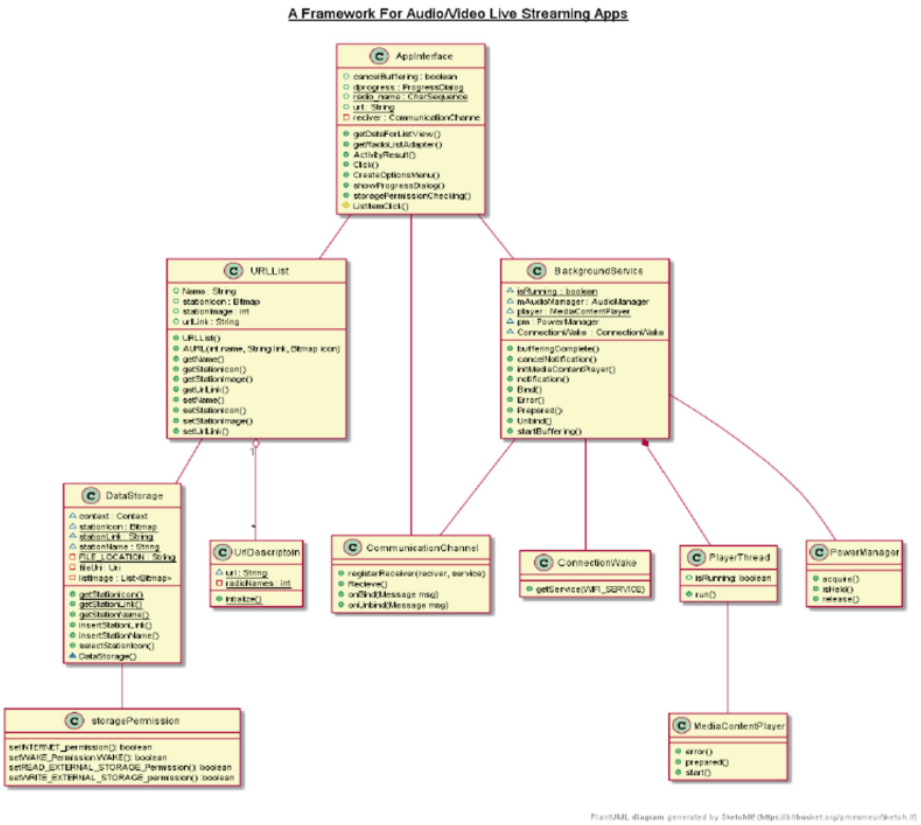

Fig. 1

Framework for live streaming multimedia content

4 Framework Implementation

Using the generic framework, we developed a live streaming app for Android devices using URL. The app groups one's favorite online radio stations and plays them on the user's devices. It is similar to Spotify but on a smaller scale (that is probably all you need; just your favorite station and not all the stations that come with Spotify).

The app acts not only as an instance of the framework and validator but is also useful for two reasons. First, all your favorite radio stations will be grouped in one place, and hence, you can play and easily switch from one station to another without any hassle. Second, it is an app that turns your device into radio and lets you listen to live streaming stations from anywhere in the world and any setting. For example, if you connect your phone with an audio cable or Bluetooth to the media player in your car, you can readily livestream your favorite online radio channels and listen to them with multiple speakers.

In implementing the framework, we use Android *Service* and *MediaPlayer* components to play live streaming radio stations using URLs. We also use *BroadcastReceiver* as a communication channel between the service running in the background and the app interface, i.e., the MainActivity. Users start the app from the main screen which gets the radio URL from the URLlist class and starts the MediaPlayer. The current app is a refactored and extended version of our previous work presented at [1]. The new features are described in Sect. 4.1.

Below, we describe the app components, implementations, and steps needed to develop such apps. We discuss the functionality and the trade-offs using different components and approaches. We also present the class structure, i.e., the architecture, of the app. The app interface is presented in Fig. 2.

IV. Framework Implementation

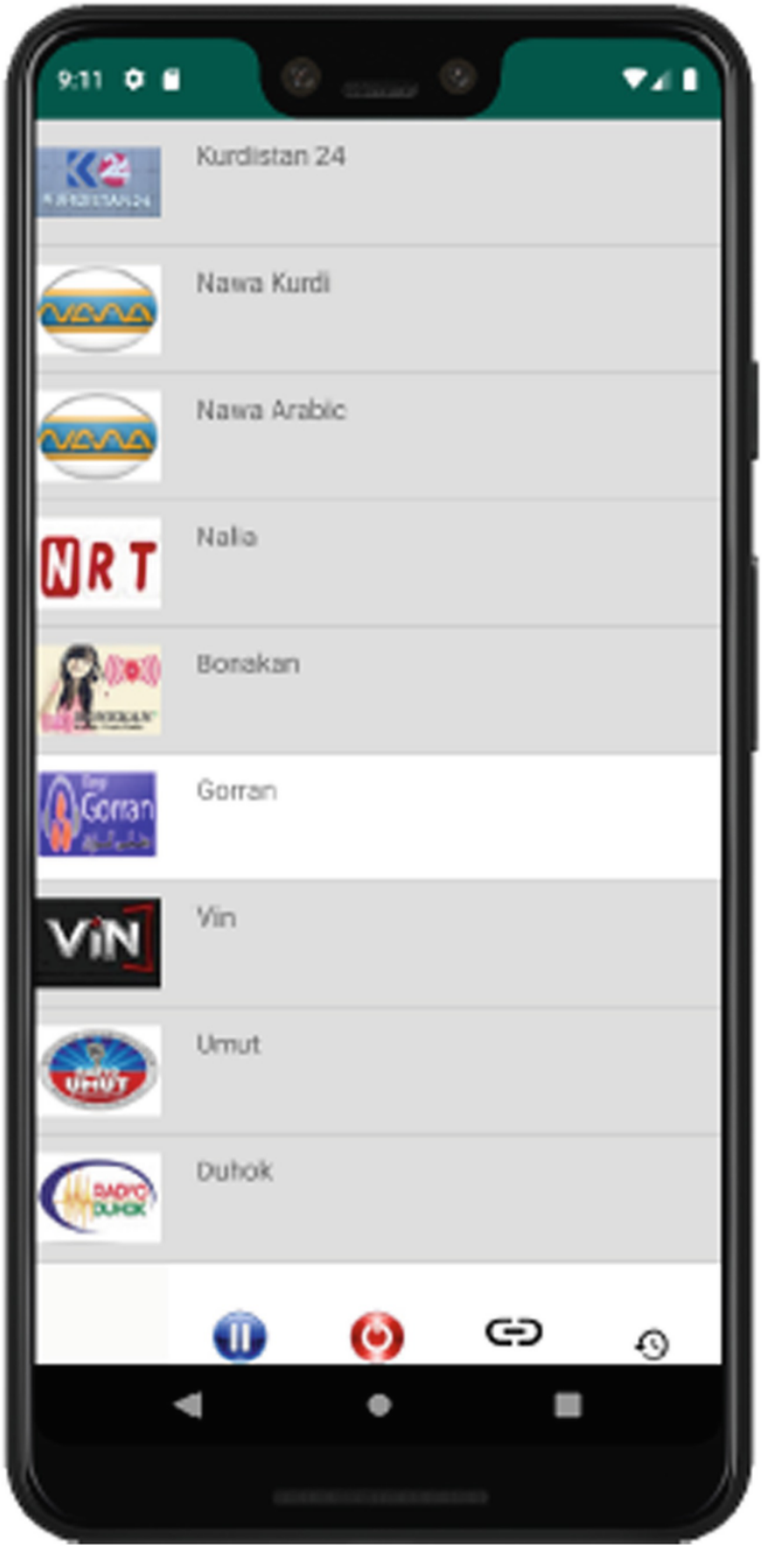

Fig. 2

App interface

4.1 App New Features

To validate the framework, we have refactored and extended our live streaming app presented in [1]. In this section, we elaborate shortly on the three new features added to the current version of the app. First, in the current version, users can delete any listed radio station. This is done by pressing and holding on an item in the list for a short period and confirming the delete by pressing the ok button on the popup dialog box. Second, users can add new stations to the list. This is done by pressing on the *link button* at the bottom of the screen and following the wizard which prompts users to enter the station name, link, and icon in sequence. This feature enables users to create a customized list of stations. Lastly, users can populate the list with predefined embedded stations when using the app for the first time or reset the list at any time by pressing the *reset* button. Figure 3 shows both the link and reset buttons at the bottom of the app.

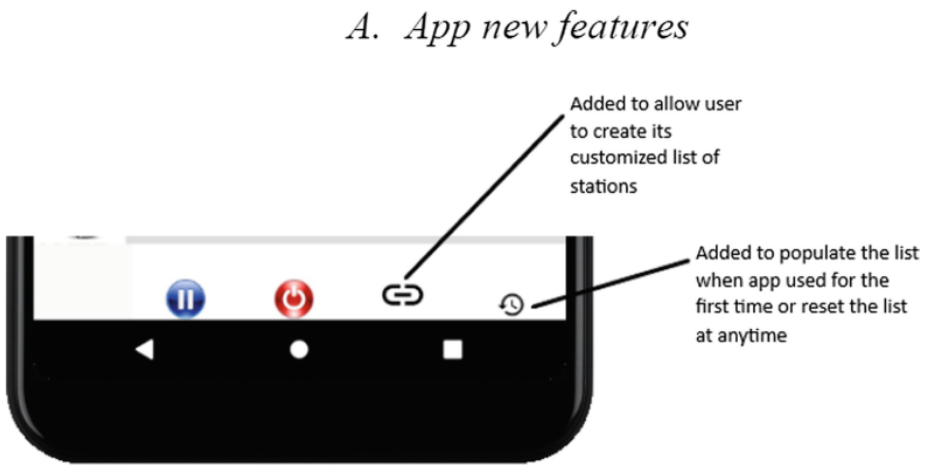

Fig. 3

App new features highlighted

4.2 Main Activity

The app's user interface and background service setup are all done at the MainActivity class. MainActivity can communicate with the background service to start and stop the MediaPlayer and listens to the message broadcast from the service, i.e., it maintains a reference to the service, makes calls on the service just as any other class, and can directly access members and methods of the service.

4.3 Service

Service [5, 6] is an app component that performs long-running tasks in the background with no graphical user interface. Once service starts, it can continue to run even if the original application is ended, or the user moves to a different app. Whether service runs continuously is platform and service setup dependent. We run service indefinitely until explicitly stopped and restart it if the Android system terminates it for any reason. Service is the right choice to use when an activity does not interact with the user, i.e., is not the forefront activity. Service can be private or public. When private, it is usable only by the app it belongs to; however, when it is public, it is usable by apps other than the app it belongs to, i.e., another app component can start Service using a call to the API. In the current app, the MainActivity component starts service with the method call startForegroundService() followed by calling startforeground(int, Notification) by the service.

4.4 Message Broadcast Receiver

We use Broadcastreceiver [2] as a communication channel to receive and handle broadcast messages sent from service by the sendBroadcast(Intent) method. It is another entry point to the app. The broadcast class does not have a user interface but can create a status bar notification to alert the user when a broadcast event occurs. The Android system delivers a broadcast Intent to all interested (registered) broadcast receivers. Apps can initiate broadcast messages to let other apps know, for example, that some data has been downloaded to the device and is available to use.

The BroadcastReceiver needs to be instantiated and registered to process the broadcasted messages on arrival. The four steps involved in message broadcast and receive are:

- Create the BroadcastReceiver object.
- Register BroadcastReceiver object to receiving messages.
- Message broadcasting.
- Actions performed upon receiving the broadcasted message.

4.5 Media Player

We use the MediaPlayer [3] class to control the playback of radio streams. MediaPlayer needs to be prepared, started, and ultimately released. The MediaPlayer's life cycle shows that the player must first enter the prepared state before playback can start. There are two ways that the prepared state can be entered:

1. 1.

   Synchronous way using the prepare () method

2. 2.

   Asynchronous way using the prepareAsync () method

The difference between those methods is in what thread they are executed. The *prepare ()* method runs in the UI (user interface) thread and thus takes a long time. It will block your UI thread and users might get an ANR (application not responding) message. The *prepareAsync ()* method, on the other hand, runs in a background thread, and thus the UI thread is not blocked; however, the MediaPlayer object might not prepare instantly. Therefore, you want to set onPreparedListener () in order to know when the MediaPlayer is ready for use. The prepareAsync () method is generally used for playing live data over streams. This is why the current app uses the prepareAsync () method. It allows playing without blocking the main thread.

4.6 Power Manager and Wake Lock

If the phone goes into a low-power state, it will prevent apps from running. To control the power state on device, you need to use power management to [4]:

1. 1.

Keep the CPU running

2. 2.

   Prevent screen dimming or going off

3. 3.

   Prevent the backlight from turning on

The Android *WifiLock* [8] class allows an application to keep the Wi-Fi component awake. Acquiring a WifiLock will keep the connection on until the app releases the lock. In the app, we have decided to keep the radio stations running even when the device screen is off; hence, we acquired the WifiLock.

The radio station names, URLs, and icon links for the preset list of radio stations are all saved or referenced at this component. This enables easy extensions and changes. For example, if you change a streaming URL or icon, you only change it here without the need to change any other parts of the code. Similarly, if you want to add a new station to your list, you simply add the new station to the list of existing stations at this component. There is no need to change other parts of the code.

4.7 User Permissions

To run a background service, you need to declare it inside the *manifest* file. Like the main method in a Java class, the manifest file is the entry point of the app. The app activities as well as proper permissions need to be added to the manifest file as well. Part of the manifest file is shown in Table 1 where the app uses the Internet, wake lock, and read and write access to the app directory permissions. Upon running the app, users must grant these permissions for the app to run. The code snippet below shows these steps for the Android app.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
super.onStartCommand(intent, flags, startId);
try {
...
player.setOnPreparedListener(this);
player.setDataSource(RadioMainActivity.url);
player.setWakeMode(getApplicationContext(),
PowerManager.PARTIAL_WAKE_LOCK);
...
player.prepareAsync();
} catch (IllegalArgumentException e) { ....}
return START_STICKY;
}
```

Table 1

App’s manifest showing the service and permission declaration

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android=http://schemas.android.com/apk/res/android
 (http://schemas.android.com/apk/res/android) ..>

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.WAKE_LOCK" />

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

<application

   ...

   <service

   android:name=".RadioService"

   android:enabled="true"

   android:description="@string/runningRadio"

   android:exported="false">

   </service>

 </application>

</manifest>
```

5 Apps Class Structure

The class structure of the instance app is presented in Fig. 4. Using a reverse engineering process, the structure is generated from the code using Android Studio and PlantUML plugin [8]. The diagram reveals that the structure, component, and relation between components match the framework architecture but have additional implementation details:

1. 1.

   The RadioService is a service, i.e., a subclass of the Android service, and implements both onPrepared and onError listeners.

2. 2.

   The MainActivity is the app interface and implements the listener interface to handle user interactions with the app, i.e., the item selected from the list.

3. 3.

   The power manager, WifiLock, and thread components are inner class members of the radio service.

4. 4.

The relationship between components is revealed as a package.

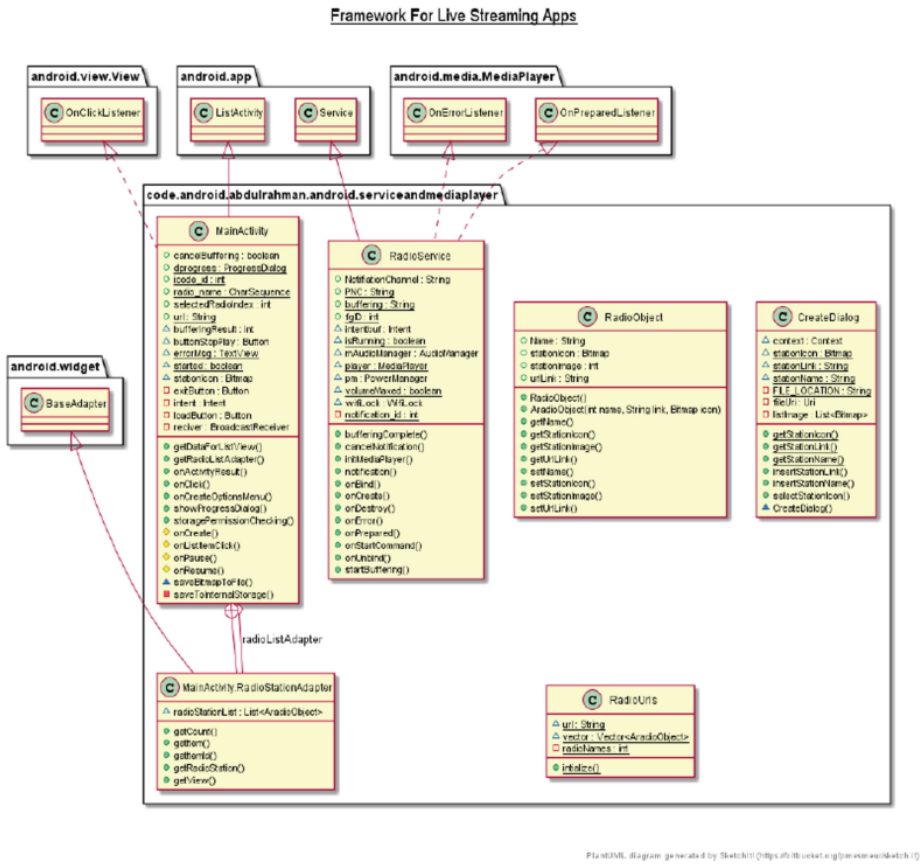

Fig. 4

Class diagram for live streaming radios app

You may note that the permissions are not shown in the class structure diagram. This is because they are included in the Android manifest XML file which is not translated to the class when generating class structure from the code.

6 Conclusion and Future Work

We have presented a preliminary framework for developing custom live streaming multimedia apps for mobile devices, i.e., we identified the principal characteristics of mobile live streaming apps and their components, components' interactions, and the design decision needed to create such apps. The framework is an app architecture that can be adapted to create apps that meet specific requirements. To demonstrate how the generic framework can be used, we presented an app for live streaming audio for Android devices using URLs. The app acts as an instance of the generic framework and validates it. The app is beneficial to both developers and end users. The paper's main contributions are (1) the generic framework which can be reused or adapted when developing live streaming multimedia apps or similar ones and (2) using new URLs, customizable live streaming apps can be created. Furthermore, the app's source code and complete documentation can be used by instructors to teach various Android topics.

The framework is a preliminary one. More work needs to be done to identify and describe properties for individual components and cardinalities between components. A framework-based app can be created for iPhones as well. We will use the knowledge gained from the identified feature works to revise the current preliminary framework and develop a more complete one.

References

1. A. Mawlood-yunis, A live streaming app for android devices, in *2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA*, (2019), pp. 1103–1106. https://doi.org/10.1109/CSCI49370.2019.00209 (https://doi.org/10.1109/CSCI49370.2019.00209)
2. MediaPlayer Overview, Retrieved May 11, 2020, from https://developer.android.com/guide/topics/media/mediaplayer (https://developer.android.com/guide/topics/media/mediaplayer)
3. Power management. Retrieved May 11, 2020, from https://developer.android.com/about/versions/pie/power (https://developer.android.com/about/versions/pie/power)
4. WifiManager. Retrieved May 11, 2020, from https://developer.android.com/reference/kotlin/android/net/wifi/WifiManager (https://developer.android.com/reference/kotlin/android/net/wifi/WifiManager)
5. Notification overview. Retrieved May 11, 2020, from https://developer.android.com/guide/topics/ui/notifiers/notifications (https://developer.android.com/guide/topics/ui/notifiers/notifications)
6. Audio focus. Retrieved May 11, 2020, from https://developer.android.com/guide/topics/media-apps/audio-focus (https://developer.android.com/guide/topics/media-apps/audio-focus)
7. https://www.theguardian.com/media-network/media-network-blog/2013/jan/31/mobile-changing-face-broadcast (https://www.theguardian.com/media-network/media-network-blog/2013/jan/31/mobile-changing-face-broadcast)
8. https://www.marketwatch.com/press-release/global-live-streaming-market-research-report-2019-2026-industry-share-and-size-by-value-and-volume-2019-11-28 (https://www.marketwatch.com/press-release/global-live-streaming-market-research-report-2019-2026-industry-share-and-size-by-value-and-volume-2019-11-28)
9. F. Bentley, D. Lottridge, Understanding mass-market mobile TV behaviors in the streaming era, in *CHI '19: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, (2019), Paper No.: 261 Pages 1–11, https://doi.org/10.1145/3290605.3300491 (https://doi.org/10.1145/3290605.3300491)

Copyright information

About this paper

Cite this paper as:

Mawlood-Yunis AR. (2021) A Framework for Developing Custom Live Streaming Multimedia Apps. In: Arabnia H.R., Deligiannidis L., Tinetti F.G., Tran QN. (eds) Advances in Software Engineering, Education, and e-Learning. Transactions on Computational Science and Computational Intelligence. Springer, Cham. https://doi.org/10.1007/978-3-030-70873-3_46

- First Online 09 September 2021
- DOI https://doi.org/10.1007/978-3-030-70873-3_46
- Publisher Name Springer, Cham
- Print ISBN 978-3-030-70872-6

- Online ISBN 978-3-030-70873-3
- eBook Packages Engineering Engineering (R0)

Personalised recommendations