Michael McGilvray

November 29, 2021

COMP 4630 Mobile App Programming I

Twitch Clips Finder – From Idea to Implementation

My app is called Twitch Clips Finder and its goal is to help Twitch users find clips easier. For anyone who hasn't heard of Twitch, Twitch is a live streaming platform where people stream anything from games to cooking to music. These live streams can be any length, meaning most viewers won't have the time or patience to watch an entire stream. For this reason, Twitch implemented clips to save sections of the live stream so they can easily be shared for viewers that want to see the highlights. The focus of Twitch Clips Finder is on the clips feature and its goal is to help Twitch users find clips easier by providing more specific filtering options and a user-friendly platform for mobile devices.

I am a frequent Twitch user and I regularly find myself running into issues when searching for clips from streamers that I enjoy watching. Using the Twitch app, users can only search for clips either by stream category or by a streamer. Although this may seem like acceptable criteria, when a user wants to simply see what their favorite streamers were doing yesterday, these criteria are not enough. In addition, the focus of the Twitch app is on the live streams and not the clips so inexperienced Twitch users might not even know the clips exist or where to find them through the Twitch app. The main objective of Twitch Clips Finder is to solve this problem and make the process of finding Twitch clips as straightforward as possible.

Before starting development, I performed some background research to determine the technologies that would be useful. The most important technology was the Twitch API since it is how I get all the relevant clip data and how the clips are played. Without this API working, my

app would be nearly useless, so I focused a great amount of time tinkering with it through the command line and in the Android Studio development environment. When looking through the Twitch API documentation, I thought that I would need to use OAuth 2.0 since it is their main method of authentication, but later discovered that it wouldn't be needed. Only the requests made through the Twitch API that are account specific require OAuth 2.0 and since my app uses publicly available clips, OAuth wasn't necessary. Although not initially obvious, I realized that I would need some way to parse the JSON data returned from the Twitch API and a method of downloading images from a URL for the clip thumbnails. I imported two Java libraries called Picasso and Org.json which helped me with image downloading and JSON parsing respectively.

Once I had a good understanding of the tools I need, I began designing a user interface on paper. At this point, I split my app into a clip searching activity and a clip watching activity. In the clip searching activity, there are filtering options followed by the clips that pertain to the filters selected. Each clip is made up of a thumbnail image, clip title, streamer name, and the number of views that clip received. The clip watching activity is opened when a user taps on any of the clip thumbnails. Once opened, that clip's video will start playing in an embedded video player with the title of the clip, streamer name, and the number of views below it. For the most optimal clip watching experience, the video player has a full-screen option but defaults to the not full-screened viewing to show the relevant clip information.

At this point in the design process, I ran into an issue regarding the Twitch API. I intended on placing a chatroom replay below the video player that shows a replay of the viewer messages when the clip was created. When looking into the Twitch API documentation, it became apparent that this wouldn't be possible on a mobile device without a clunky workaround. Currently, Twitch doesn't support the chatroom replay feature since they are in the process of

revamping their API from Legacy Twitch v5 to the New Twitch API. In the older version of their API, this feature was supported but is now unavailable to developers who created an app after July. To remedy this issue, I planned on replacing the chatroom replay feature with a section of similar clips to the one that is playing. The section of similar clips is essentially clips that are recommended if the user enjoys the current clip they are watching. Each clip will be displayed in the same way as the ones in the clip searching activity. In addition to this, at the bottom of the clip viewing activity is a button that will open the Twitch app with the live stream at the point when the clip was created. This means if a user wanted to see more than just the five to sixty-second clip, they could press this button to watch more of the live stream before or after the clip was created.

Once the app was designed, I had to first register the application through the Twitch developer console. Doing this gave me a client ID and an app access token that I could use to begin making API requests and receiving JSON data. Although I hoped it would be this simple, it took a lot of trial and error to get any response other than an unauthorized access message. To avoid hours of frustration and to continue making progress, I decided to simultaneously work on my app in Android Studio.

I first created two activities and added functionality to switch between them. The plan was for the user to tap on a thumbnail image and open the clip viewing activity and to return they could press the up button in the top left of the clip viewing activity. I started by creating a search button that would generate a list of clips and their relevant information below in a scroll view. At this point, I was still working out the kinks of the Twitch API, so I proceeded with sample data. This was straightforward until I tried creating an image view using a URL, at which I imported

Picasso, the Java image downloading and caching library. Using Picasso, I could load the thumbnail image from a URL and set it as the source of each image view.

Around the time of overcoming the thumbnail image problem, I returned to fiddling with the Twitch API. To get my first request, I used cURL through the command line to verify that I could get a valid response. I then attempted to implement the API requests in Java using the HttpURLConnection class and passing my app's authentication information. To make the rest of the development easier, I created a TwitchRequest class that returns a JSON string of the result data depending on the type of request that is made. Meaning I can make any Twitch API request in Java by passing the appropriate URL found in the Twitch API documentation and get the result data in the form of a JSON string. This planning ahead simplified working with the Twitch API and thus saved lots of time. To make use of the JSON string result, the data then must be processed into a more usable form. At this point, I imported the Org.json Java library to parse the JSON data and save the results using my Clip class. This class stores information such as the title of the clip, streamer name, thumbnail URL, and a few other pieces of data to make the clips easier to work with.

With the groundwork of the core functionality on my app complete, I began to work on the UI so the user could make use of the clip data I was receiving. In the clip searching activity, I added a search bar so the user can search for clips based on Twitch's stream categories. To make this feature more user-friendly, I added an autocomplete that recommends five categories most like the search query. To implement this feature, I create a TwitchRequest every time the search text is changed and display the list of categories returned directly below the search bar. Once a category is selected and the search button is pressed, a list of clips is shown below based on the

category. I planned on adding the other filtering options later because if I could filter by one option, adding additional filters would be a similar process.

With some of the basic filtering and clip selection process completed, I attempted to embed the clip video player in the clip watching activity. Before I could attempt this, I needed to pass the clip object that was selected from the clip searching activity to the clip watching activity. To do this, I added an extra piece of data to the intent that opened the clip watching activity that could be received later. With the clip object available in the clip watching activity, I experimented with methods of playing videos using Android Studio. Initially, I tried using a video view but discovered that I would have to download the clip as an mp4, so I looked for alternatives. To embed the Twitch video player, the API requires that the website that the video player is embedded in be specified. Since my app has no website associated with it, this option wasn't originally viable until I came to a creative realization. I created a website hosted using GitHub Pages that has the Twitch video player embedded in it. In my app, I created a web view that loads this website to then play clips and can pass an additional URL parameter to specify which clip to play. Through some simple JavaScript code, the URL parameter is received and updates the source of the iframe on the website. Meaning in the app, I can play any Twitch clip using the Twitch video player by loading my website on GitHub through a web view. Using Twitch's video player is extremely powerful because it comes with many features like play and pause buttons, volume controls, video quality controls, and much more.

At this point, most of the app's core features were implemented and all that was left was to fill in the gaps. Using the same methods from the category search functionality, I added options to search by streamer name, by date, and any combination of the three. Provided that there are enough clips that meet the search criteria, 20 will be clips generated when the search

button is pressed. Since the Twitch API only allows to search by one of these categories at a time, my app searches by the most specific criteria first then checks if the clips returned matches the other criteria. For example, if a search is made using the streamer's name, stream category, and date, I would first search for clips using the streamer's name then check all the clips returned if they had the correct category and date. In addition to the filtering, I finalized the video player controls by adding a full-screen button that rotates and changes the size of the video view accordingly. I also implemented the button to open Twitch at the point in the stream when the clip was created using the time the stream started and the time the clip was created. This way if a user wanted more context to a clip they could see more of the stream before and after the clip was created. Other than these last few features, I spent a while cleaning up the look of the app by tweaking the layout and styling of various components.

From beginning to end, this app served as a learning experience in not only designing a mobile app but also how to use the tools and some best practices in mobile app development. I learned a lot about the Android Studio development environment and coding in Java while at the same time how to work with an API. In previous classes, I have only briefly worked with a handful of APIs and never got to experience what is like to have a project's core functionality rely on an API. I enjoyed the freedom this project offered as I was able to learn the ins and outs of mobile app development from brainstorming to designing to implementation.

References

Org.json Reference. https://stleary.github.io/JSON-java/index.html.

"Picasso - A Powerful Image Downloading and Caching Library for Android." *Picasso*, https://square.github.io/picasso/.

"Twitch API Reference." *Twitch Developers*, https://dev.twitch.tv/docs/api/reference.