PELICAN FINAL REPORT

Sebastian Battle and Kyle Glaws

Index

Chapter 1: Requirements

Introduction

For our capstone project, we will be building a video sharing app for Android phones. Users will record short 7-10 second videos, and be able to share them with other users. The purpose of having this hard limit on the length of the videos is to maximize interesting content within a short period. Users will also have the option to follow their friends and favorite app users, and accumulate a video feed which they can browse through. By simply tapping their finger, a consumer can quickly cycle through all videos from their followed producers. Ideally, popular content creators will upload particularly funny or amusing videos. Users can react to videos with a thumbs up or thumbs down (subject to change). Our ideal market are millennials, and we will target our app to a generally younger audience.

Technical Details/Challenges

Android apps are built using Java, so we will be using the Android Studio IDE. In order to accomplish our design, we will also require the use of a database with a remote server from which to host the app's video content. We will need to connect multiple users to the database and load the videos that they record in the app onto the server. Also, we will need to maintain a cache of videos on the users local device to streamline the viewing speed.

We will require the use of a server, however we are already in possession of a tower with ample storage and processing power. Our server will be running Mint with

MySQL and Apache. We will rely on Flask for the server-side operations. In terms of the client, we will be utilizing the Android API for most of our goals, and we will acquire additional libraries as needed. For transfering files to the server, we will be writing our own class that inherits from Android's AsyncTask. We will also rely on the phone's back facing and front facing camera, so we will obviously need to request the user for those permissions in order for the app to fully function. If the app is successful or if we are able to make significant progress, we will port the app to iOS.

Hardware

- Dell PowerEdge T320 (The Server)
    - 16 GB RAM
    - 4 TB Storage
    - 8 Cores
- Android Phones
    - LG G6
    - Motorola Droid Turbo 1
    - Various virtual machines

Deliverables

Android Application

- Video
    - Android Camera API

        - VideoView Class
        - MediaPlayer Class
    - UI
        - Android views
        - ViewPager Class
        - View Fragments
    - Userbase
        - Firebase for user authentication
        - MySQL database using Firebase ID
    - Server Interaction
        - Custom upload and download classes

Server Software

- Mint
- Python Flask Microframework
- MySQL
- Apache

Timeline

March 5th - Basic video recording with UI, server online

March 12th - Video File uploading

March 19th - Working database (properly store videos)

March 26th - User database/ account creation

March 31st - Flesh out UI

April 7th - Video Streaming

April 14th - User interaction, following/video upvoting/downvoting

April 21st - Continued work on user interaction

April 28th - Testing multiple users

May 5th - Finalize UI

May 12th - Deploy

Stretch Goals

Video Caching

Personalized User "Wall"

Framework Comparison

Possible Options:

Flask - (Our current preference) Python microframework, easy to implement, ample documentation, previous experience with Python (Kyle). We have officially decided to use flask for our project because we believe it is best suited for our needs. It's simple and more barebones, which will be easier for us to use as this is our first time working with a web framework.

Django - Python framework, more built-in functionality, bells and whistles, more out-of -the-box ready to use. It comes with things like an admin panel, database interfaces, and an ORM. These are great additions but we honestly want to keep things as simple and straightforward as possible for our first app, so we be using Flask so we don’t get bogged down with managing “extra” features.

Express.js with Node.js - JavaScript “un-opinionated” framework, previous experience with Node.js (Seb), easy asynchronous functions. Express is useful because it is relatively easy for beginners, and there is a large amount of plugin support. The only problem with express is there is no clear-cut way of doing things, which may be confusing or make it somewhat more difficult for us to implement as this is our first time using a framework

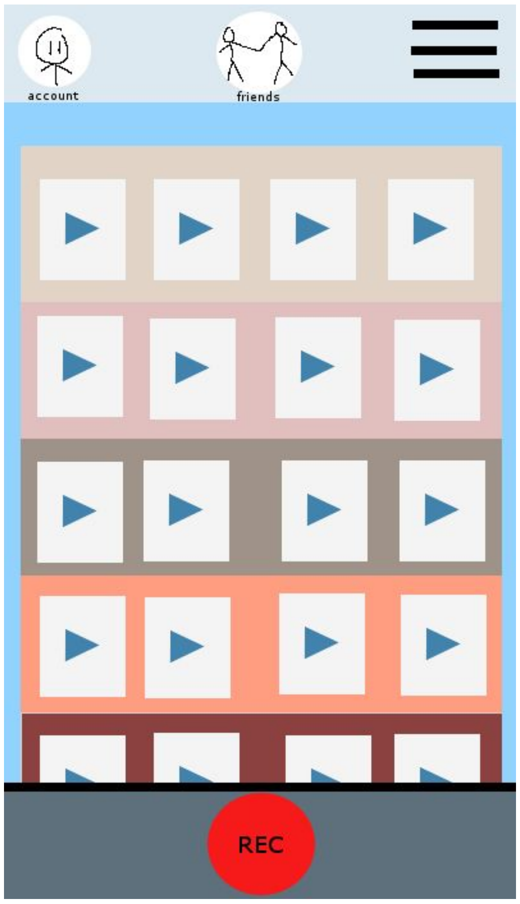
Skills to Acquire

We will need to learn how to send files to a server and how retrieve them. We will have to dedicated a large amount of time to learning the Flask framework, and also a large amount of time to learning the Android API. We will both need to brush up on our Python. We will need to learn the specific classes very well, in particular the ones involving the camera and videos. Also, we have made simple apps in Android Studio before, but this one will be much more involved and have a far steeper learning curve. Neither of us has worked with a database before, so that will be entirely new terrority.

Chapter 2: Design Considerations

Early Mockup:

The below design is a mockup of our potential user interface for the app. Easy access should be provided to the record button, and one should be able to easily see their "followed" friends. In our very early concept, the videos are presented in the style of a building, with each floor representing one of a user's followed producers. Each window represents one of their uploaded videos that can be clicked on. Once clicked, the video will blow up to a full screen size, and one can simple tap their finger to cycle through the videos.



Possible Project Names

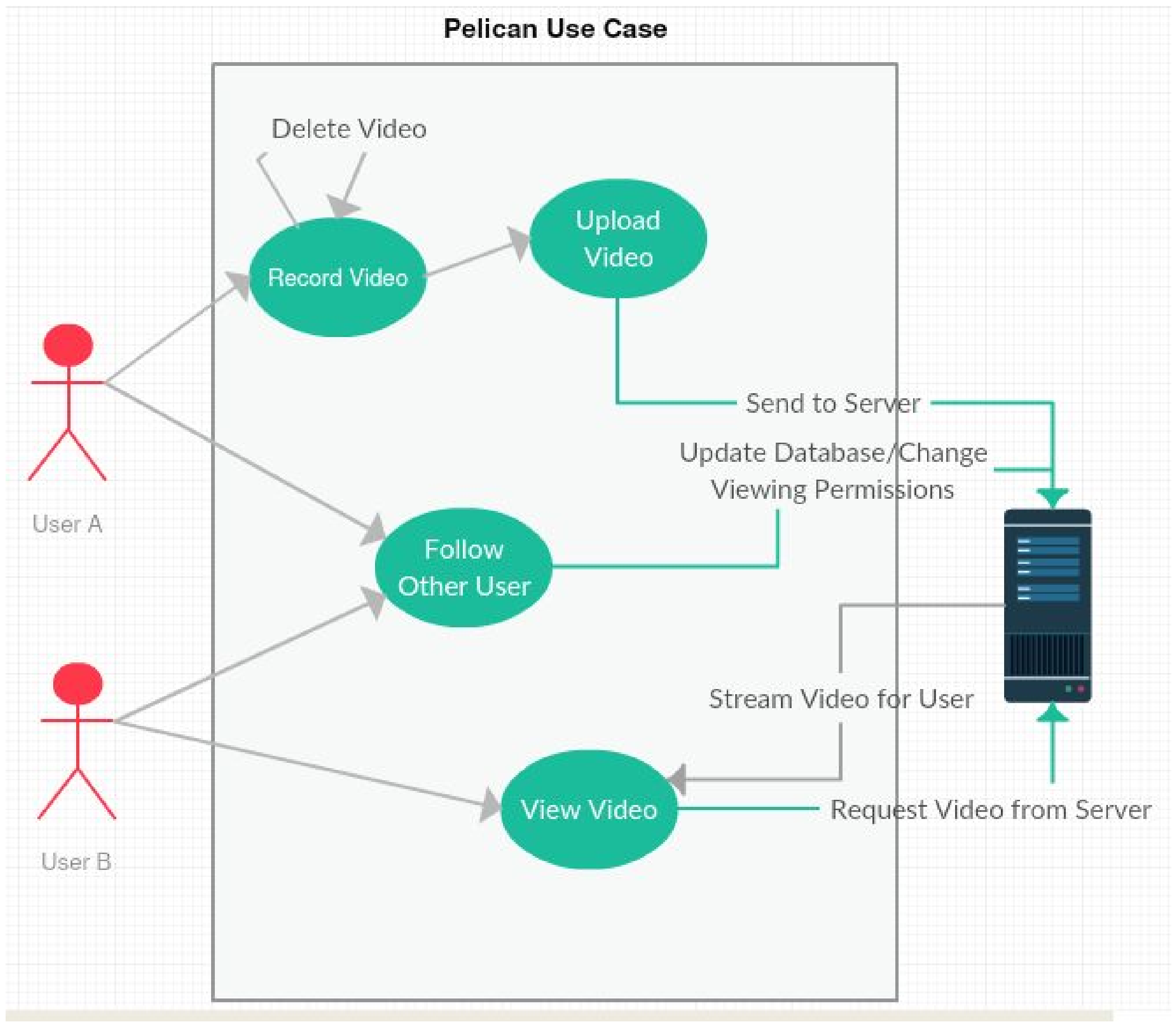
Views

Pelican

Floors

Heights

Use Case:



Use Case Description:

User A - Records video, has the option to delete and record a new video or upload to server where it will be viewable to other users who have followed User A.

User B - Follows User A/other users and is then able to stream videos belonging to that user from the server.

Server - When one user follows another, the SQL database is updated to enable that user to access the followed users videos. The server then streams those videos to other users when requested. When a video is uploaded, the video is associated with the user on the database.

Chapter 3: Coding Experience

The coding experience for Pelican^{TM} was an entirely new challenge, as we had very little experience with Android development before jumping into the creation of our application. There was a long period of time before real development got started; we made a few basic activities, but ultimately we had to spend several hours just learning how everything worked. We greatly underestimated the labor involved in order to just get started. However, eventually we both felt comfortable with our knowledge and were able to move forward.

The first thing we did was divide the workload: Kyle focused on getting user registration and login working with Firebase, and Sebastian worked on the app layout and getting swiping, crossfades, and the view cycle all working properly. Eventually, we switched to focusing on ironing out the registration bugs, and getting our custom camera working along with the camera preview and front-facing camera.

Throughout our initial development, we often had to pause and spend a lot of time researching the specific feature we were trying to implement. We spent a lot of time looking at examples and official documentation provided by Google themselves, as that was the most helpful. When we had a specific bug or issue to solve, we could occasionally find a solution on StackOverflow. Although we had established foundational knowledge at the beginning of our project, most of the true, relevant knowledge was learned by “doing.” We are both kinetic learners and by actually jumping straight in and trying to figure it out, we were able to make progress. However, our triumphs were not without immense trial and error.

After accomplishing these tasks, Sebastian worked on the video playback while Kyle worked our initial backend PHP scripts. We were attempting to emulate the video recording followed by immediate playback seen in the popular app “Snapchat”; we were following the mantra that “if it isn’t broken, don’t fix it.” Once these tasks were accomplished, we coded our VideoUploadTask (this took a lot of research to understand exactly how it worked, as the video file has to be broken down into MultipartEntity in order to be sent). We also learned about multithreading and handling tasks asynchronously, as Android apps primarily run in the UI thread, and doing a long task like uploading a video could potentially freeze the app.

After accomplishing this, we decided that it was time to shift over to Flask, as the PHP scripts were mainly a quick way to see if our VideoUploadTask was working properly. We got our server up and running and installed MySQL, and created our backend in Flask, which made the entire process much simpler. Although we would have liked to entirely use Flask for our authentication, due to time constraints we decided to stick with Firebase for the time being.

Also during this process, we begun working on our video feed and establishing a way to pull the videos from the server. We created a profile screen, added logout functionality, and made the entire application look far better. We enhanced all layouts, added various animations, tweaked our colors and graphics, etc. We accomplished getting a smooth scrolling video feed, where videos are displayed in the order they were uploaded in, and a refresh function to reset the list when a video is added to it.

To finish, we did some polishing and final tweaking. We ran some tests, making sure that the app functioned correctly across multiple devices. Overall, the coding experience was a very research-heavy, labor intensive process, but very rewarding and eye-opening. We will cover our closing thoughts at a greater length in Chapter 6.

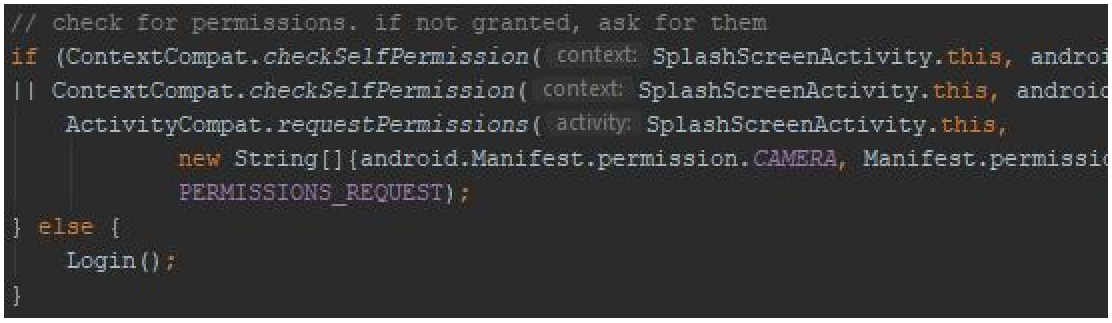
Chapter 4: How It Works

Our project was written primarily in Java and XML using the Android Studio IDE. We used several classes that come standard with the Android SDK, but also wrote a few classes of our own. The backend of the app went through a number of iterations, originally using Firebase for both the user database and for video storage. We later wrote a separate backend for serving videos that was written in PHP, but scrapped it before finally arriving at a Python based backend using Flask and MySQL.

Our java classes all have corresponding XML layouts, we also have additional custom resources for the UI. Android Studio provides vector images for commonly used buttons, but some were entirely custom. Our list of java classes is as follows:

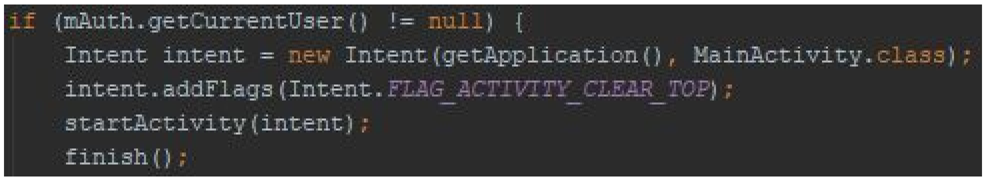
- ➔ CameraFragment
- ➔ ChooseLoginRegistrationActivity
- ➔ GetRequest
- ➔ HomeFragment
- ➔ LoginActivity
- ➔ MainActivity
- ➔ Profile
- ➔ RegistrationActivity
- ➔ SplashScreenActivity
- ➔ TabsView
- ➔ Video
- ➔ VideoAdapter
- ➔ VideoPlayer
- ➔ VideoUploadTask

Upon opening the app, the SplashScreenActivity is the first class to be executed. First the Pelican^{TM} logo is displayed to the user for two seconds before requesting permissions to use the camera and the microphone.



```
// check for permissions. if not granted, ask for them
if (ContextCompat.checkSelfPermission( context: SplashScreenActivity.this, androi
|| ContextCompat.checkSelfPermission( context: SplashScreenActivity.this, androic
    ActivityCompat.requestPermissions( activity: SplashScreenActivity.this,
            new String[]{android.Manifest.permission.CAMERA, Manifest.permissic
            PERMISSIONS_REQUEST);
} else {
    Login();
}
```

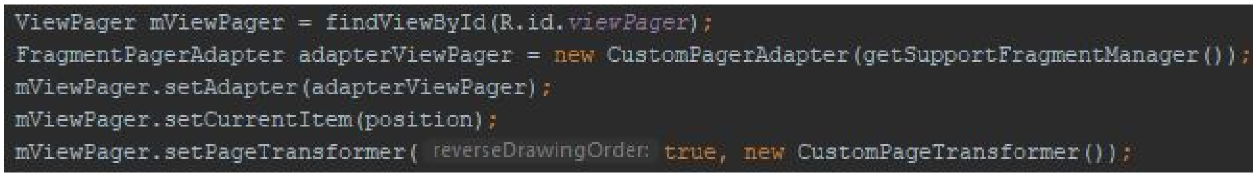
Once granted, or if already granted, the app checks if a user is already logged in on the device using a FirebaseAuth object.



```
if (mAuth.getCurrentUser() != null) {
    Intent intent = new Intent(getApplication(), MainActivity.class);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    startActivity(intent);
    finish();
```

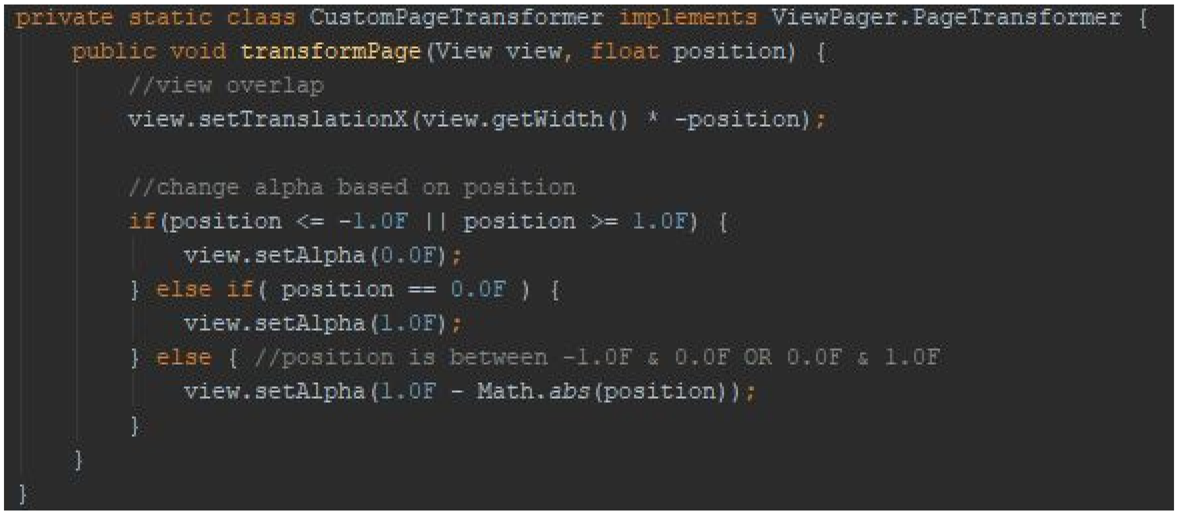
If a user is logged in, the app proceeds to the MainActivity class. There the user can: view all uploaded videos, view their profile screen (limited for now), sign out, and record a video and upload it. If a user is not already logged in, the app proceeds to the ChooseLoginRegistrationActivity class where the user can choose to log into an existing account or to create a new account. If the “Register” button is selected, the app proceeds to the RegistrationActivity class. Here the user inputs their username, email address, password, and the password again to ensure they entered it correctly. Once the user submits their information, the information is sent to Firebase on a separate thread. If the task returns without an error, the user has been successfully registered, and the app then proceeds to the MainActivity class.

Once the user has logged in/registered with Pelican^{TM}, they will now have access to the primary functionalities of the app. The MainActivity class was divided into two sections: the HomeFragment, and the CameraFragment. Users can easily swipe between the two fragments by using a ViewPager. When setting up a ViewPager, one must establish a ViewPager adapter and a ViewPager transformer.



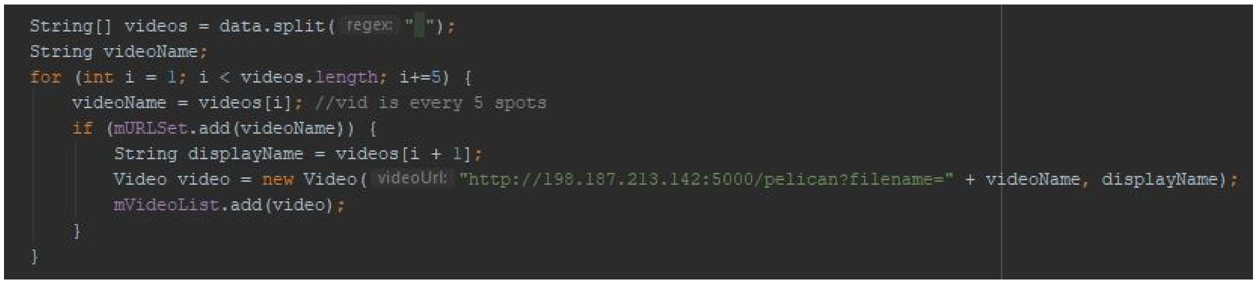
```
ViewPager mViewPager = findViewById(R.id.viewPager);
FragmentPagerAdapter adapterViewPager = new CustomPagerAdapter(getSupportFragmentManager());
mViewPager.setAdapter(adapterViewPager);
mViewPager.setCurrentItem(position);
mViewPager.setPageTransformer( reverseDrawingOrder: true, new CustomPageTransformer());
```

The ViewPager adapter handles the actual creation of the the fragment instances by checking the current position (an integer variable) of the ViewPager (HomeFragment is 0, CameraFragment is 1). The transformer is what handles the actual transition between the fragments. We designed a custom transformer that gradually adjusts the alpha, creating a crossfade effect, and also changes the xTranslation which keeps the views overlapped, as seen below.



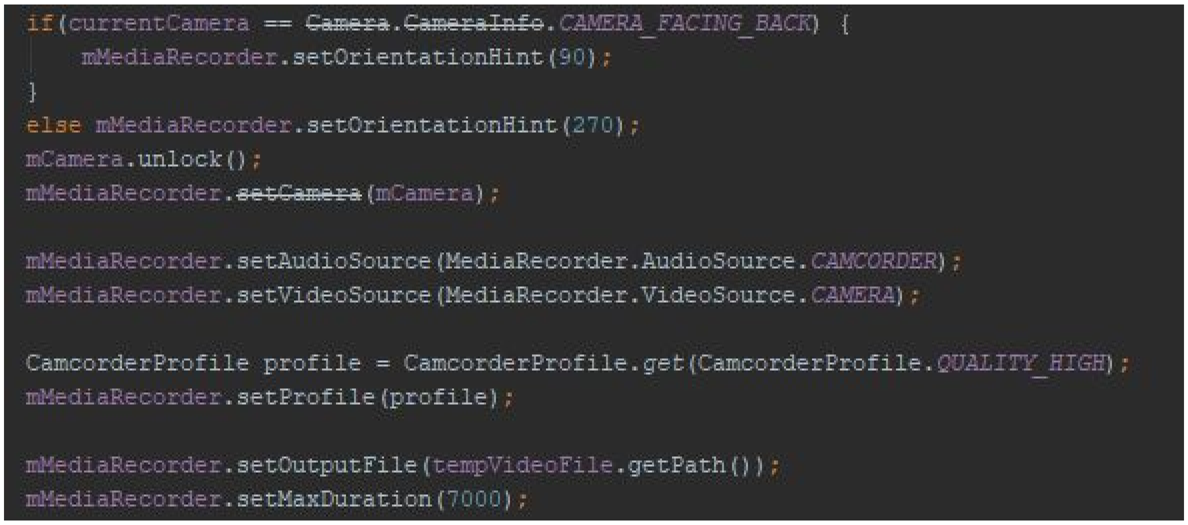
```
private static class CustomPageTransformer implements ViewPager.PageTransformer {
    public void transformPage(View view, float position) {
        //view overlap
        view.setTranslationX(view.getWidth() * -position);

        //change alpha based on position
        if(position <= -1.0F || position >= 1.0F) {
            view.setAlpha(0.0F);
        } else if( position == 0.0F ) {
            view.setAlpha(1.0F);
        } else { //position is between -1.0F & 0.0F OR 0.0F & 1.0F
            view.setAlpha(1.0F - Math.abs(position));
        }
    }
}
```

The HomeFragment contains the video feed in addition to the main UI. The video feed generates a scrolling list of videos that will play when clicked. All of the videos are played in a VideoView, which is stored in a CardView, which is then stored in a ListView. We fetch the videos from the server with the GetRequest class, which extends AsyncTask. We fetch a string from our backend, sort it, and pull what we need from it (in this case, the video file name).



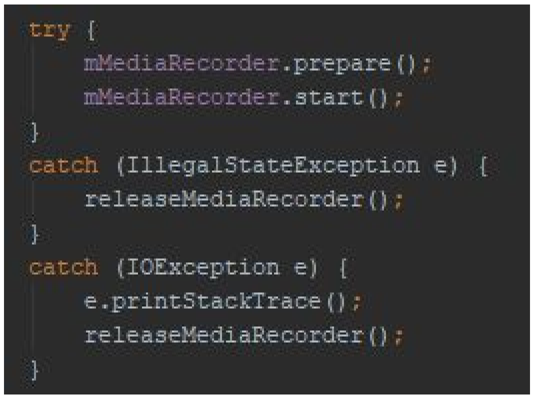
```
String[] videos = data.split( regex: " ");
String videoName;
for (int i = 1; i < videos.length; i+=5) {
    videoName = videos[i]; //vid is every 5 spots
    if (mURLSet.add(videoName)) {
        String displayName = videos[i + 1];
        Video video = new Video( videoUrl: "http://198.187.213.142:5000/pelican?filename=" + videoName, displayName);
        mVideoList.add(video);
    }
}
```

The CameraFragment follows the general design pattern of creating a custom camera in Android using the Camera class (not camera2). First, the camera preview had to be established, which is shown through a TextureView. The choice to use a TextureView instead of a SurfaceView was intentional, as the TextureView allows things such as transformations (critical for the alpha crossfade). When creating a camera, one must establish various parameters and settings. This is all handled through the MediaRecorder class, which handles all things related to video recording.



```
if(currentCamera == ~~Camera~~.~~CameraInfo~~.CAMERA_FACING_BACK) {
    mMediaRecorder.setOrientationHint(90);
}
else mMediaRecorder.setOrientationHint(270);
mCamera.unlock();
mMediaRecorder.~~setCamera~~(mCamera);

mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

CamcorderProfile profile = CamcorderProfile.get(CamcorderProfile.QUALITY_HIGH);
mMediaRecorder.setProfile(profile);

mMediaRecorder.setOutputFile(tempVideoFile.getPath());
mMediaRecorder.setMaxDuration(7000);
```

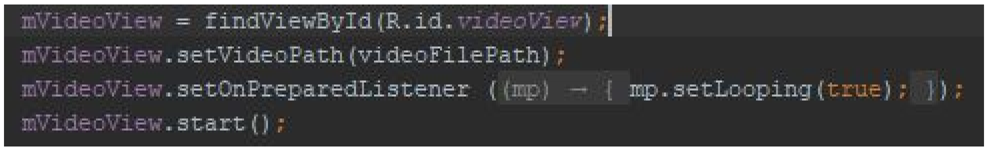
After linking the camera with the view, we had to start the actual recorder. This is done as follows:



```
try {
    mMediaRecorder.prepare();
    mMediaRecorder.start();
}
catch (IllegalStateException e) {
    releaseMediaRecorder();
}
catch (IOException e) {
    e.printStackTrace();
    releaseMediaRecorder();
}
```

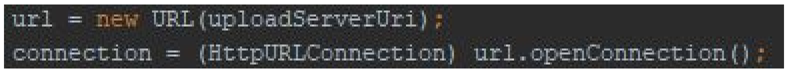
All of this code is within one function, which essentially prepares and starts the MediaRecorder. This function is called when the user holds down on the record button, which is within an OnTouchListener. This listener allows us to detect when a user initially presses the button, and also when they lift their finger.

When the user lifts their finger, they are automatically sent to the VideoPlayer activity using an Intent. The VideoPlayer is simply a VideoView that grabs the video file from the app's cache, and instantly begins playing it on a loop.



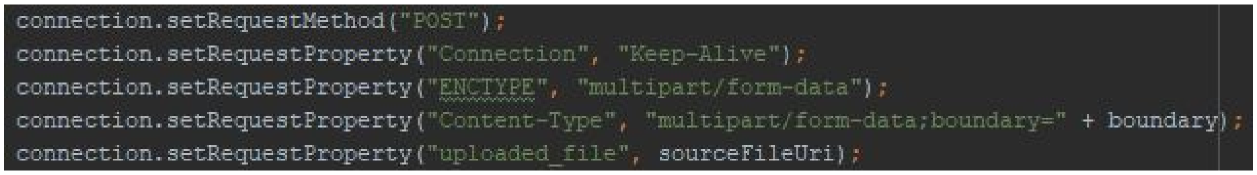
```
mVideoView = findViewById(R.id.videoView);
mVideoView.setVideoPath(videoFilePath);
mVideoView.setOnPreparedListener ((mp) → { mp.setLooping(true); });
mVideoView.start();
```

Users have the option to upload the videos that they have recorded to the Pelican™ database where it can be viewed by other users. Videos are uploaded with the VideoUploadTask, which is a class we wrote that inherits the functionality of AsyncTask. This is important because AsyncTask (obviously) operate asynchronously from the the main GUI thread. We wanted this functionality in the VideoUploadTask because it allows the main thread to remain responsive to user interaction while the upload task is being handled. If the user chooses to upload their video to the server, the path to the temporary video file that was created by the CameraFragment class is passed to an instance of VideoUploadTask. From there, a connection to the server is established.



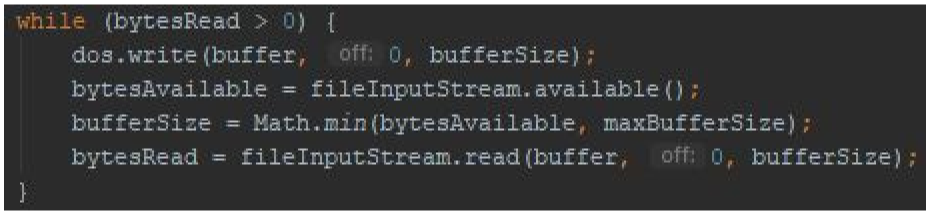
```
url = new URL(uploadServerUri);
connection = (HttpURLConnection) url.openConnection();
```

Next, the request properties are set.



```
connection.setRequestMethod("POST");
connection.setRequestProperty("Connection", "Keep-Alive");
connection.setRequestProperty("ENCTYPE", "multipart/form-data");
connection.setRequestProperty("Content-Type", "multipart/form-data;boundary=" + boundary);
connection.setRequestProperty("uploaded_file", sourceFileUri);
```

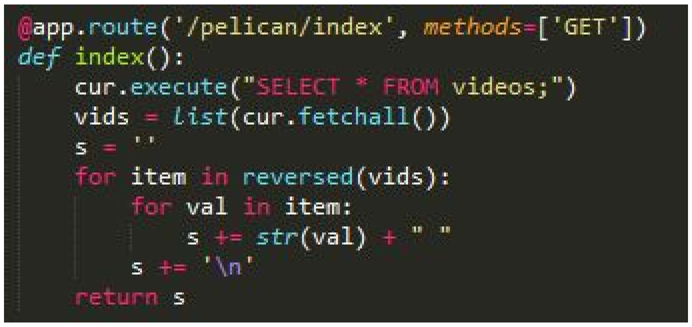
Then, the video data is read into a buffer and sent to the server in chunks.



```
while (bytesRead > 0) {
    dos.write(buffer, off: 0, bufferSize);
    bytesAvailable = fileInputStream.available();
    bufferSize = Math.min(bytesAvailable, maxBufferSize);
    bytesRead = fileInputStream.read(buffer, off: 0, bufferSize);
}
```

On the server side, there is a backend written with Python Flask that awaits requests from the app. Depending on the kind of request (POST or GET) and the URL being hit, the backend will respond accordingly. The backend served http from two routes: 5000/pelican and 5000/pelican/index. From /pelican the apps can both upload and download videos from the server. The purpose of /pelican/index is only to grab a list of all available videos, which the app can then use to grab each video from the server at /pelican.

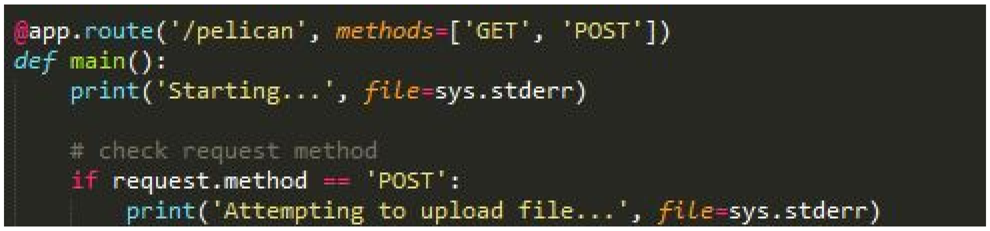
Once a request has been received, if the request URL is at /pelican/index, the following code is executed:



```python
@app.route('/pelican/index', methods=['GET'])
def index():
    cur.execute("SELECT * FROM videos;")
    vids = list(cur.fetchall())
    s = ''
    for item in reversed(vids):
        for val in item:
            s += str(val) + " "
        s += '\n'
    return s
```

The decorator in this block of code informs the Flask framework to call index() whenever the /pelican/index url is hit with a GET request. The ‘cur’ object is the database cursor object that is used to interact with the MySQL server we set up. After retrieving the entire contents of the videos database, we return it as a string and in reverse order so that the most recent videos are at the top. After receiving that string, the app can parse each video name it needs and request it from /pelican.

When a request is sent to /pelican, the main() function is called.



```python
@app.route('/pelican', methods=['GET', 'POST'])
def main():
    print('Starting...', file=sys.stderr)

    # check request method
    if request.method == 'POST':
        print('Attempting to upload file...', file=sys.stderr)
```

All POST methods are assumed to be video file uploads. In the future, this would also be where the user database would be modified (i.e. new user created, user follows another user, etc). Once the video file is received, the file name goes through a series of safety checks (to check the file extension, filename cannot be empty, etc.) before being saved to the uploads/ directory, and new entry is place into the videos database. In the future we need to check for valid MIME types as well. If a GET request is received, the requested file name is retrieved. If the file name exists in the videos in the database, the video is retrieved from uploads/ and sent to the client.

Chapter 5: The Decisions We Made

We had to make a variety of crucial design decisions when working on Pelican™ which varied in size and scope. First, we knew we wanted to have a specific look; we wanted the app to be entirely self-contained and professional. Because of that, we decided we absolutely needed to have a custom camera UI, as the camera is one of the core features of our application. We did not want to have to call any external camera applications (such as the default camera) because it can make for a poorer user experience and we wanted the UI to look a specific way. Because of this decision, we realized that we would have to create our own custom camera. We allotted a fair amount of design time for that feature, as we knew it require a lot of research and trial and error.

Initially, we thought the camera2 Android class would be the best option for interfacing with the phone camera. After looking through the documentation, however, we decided that it was far too complicated for us to figure out in time. Although deprecated, the older Camera class appeared to be far simpler to implement, so we went with that instead.

In terms of our backend, we went through a couple different ideas. Initially, Firebase seemed appealing because of its ease of use and it is rather friendly towards new developers. We planned on utilizing Firebase for our user authentication, registration, and video storage. However, we soon learned that although Firebase makes everything very straightforward, it wasn't ideal for our purposes. In order to have true scalability, we realized we would have to shift to our own backend (plus it is much more fun to do it yourself).

We initially wrote some PHP scripts to test if our video upload was working. For some time, we considered staying sticking with the PHP as it was working perfectly. However, we understood that if our app was to grow and gather a user base, we would need a backend that would better suit our needs. Because of this, we decided to go with Flask.

We had a couple different choices when it came to web frameworks, but ultimately we decided Flask was best because it was much lighter and easier to learn. We didn't want something with an extraordinary amount of features since our needs were relatively barebones. Flask is well documented, minimalistic, and very straightforward, and incorporating it into our project streamlined the entire process. Other options included frameworks like Django and Express.js. Django is another Python framework that contains many more features than Flask. However, whereas Flask is unopinated and gives the developer far more flexibility, Django is rather opinionated. Because of all of the bells and whistles involved, there is more or less a set way to do things. Express.js was another excellent choice, but we decided against it because we would have rather worked in Python, as it seemed to be a more straightforward implementation than it would be in JavaScript. In order to properly divide up our time and resources, we decided that going with Flask was the best option.

Chapter 6: Lessons Learned and the Future

First and foremost, we both learned right off the bat that app development means labor. Even though we are both experienced in Java (in fact, it is literally our first language), jumping into Android Studio with almost no experience was like learning an entirely new language. On top of that, we also had the opportunity to learn XML, PHP, SQL, and Python and applied these languages during the development of our app.

Prior to this project, neither of us had any experience with databases, frameworks, HTTP, Git, and several other tools we had to use. We had to learn all of these from scratch and under strict time constraints. We both found that the pressure and difficulty we faced when working through this project brought out our best work, and we are very proud of the app we made. This project was an incredibly valuable experience for us. It gave us a glimpse of what real professional software development looks like. More importantly, it showed us that we have a lot more to learn, and we could not be more eager to do so.