YouTube Skeleton Clone Design

Introduction

This document designs a simplified YouTube clone, which I implemented within my **Full Stack Development course**.

The goal of this project is *not* to build a 1 to 1 clone of YouTube, but rather to build a rough skeleton where the core functionality of YouTube is implemented.

We are focused on keeping the design as simple as possible, while still addressing some scalability tradeoffs. We are focused on learning, not building a production ready system.

Background

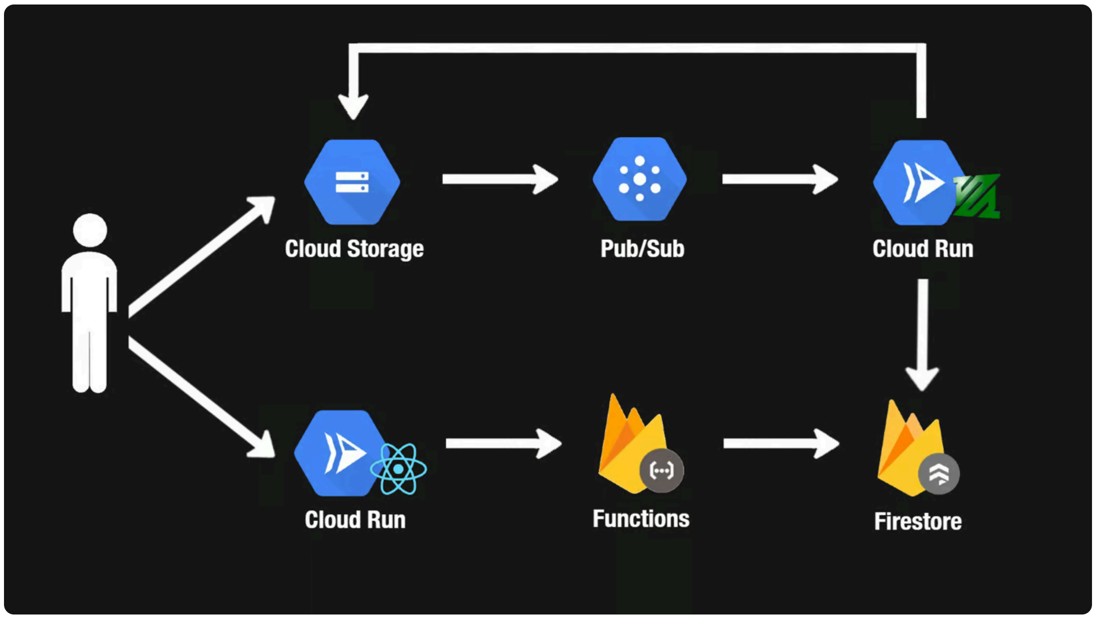
YouTube is a video sharing platform that allows users to upload, view, rate, share, and comment on videos.

The scope of YouTube is very large, such that even "trivial" features like rating and commenting on videos are actually quite complex at this scale (1B+ daily active users). For this reason, we will be focusing mostly on uploading videos, and a bit on viewing videos.

Requirements

- Users can sign in/out using their Google account
- Users can upload videos while signed in
- Videos should be transcoded to multiple formats (e.g. 360p, 720p)
- Users can view a list of uploaded videos (signed in or not)
- Users can view individual videos (signed in or not)

High Level Design



> The high level architecture of the app, specifically the cloud services we will use.

Video Storage (Cloud Storage)

Google Cloud Storage will be used to host the raw and processed videos. This is a simple, scalable, and cost effective solution for storing and serving large files.

Video Upload Events (Cloud Pub/Sub)

When a video is uploaded, we will publish a message to a Cloud Pub/Sub topic. This will allow us to add a durability layer for video upload events and process videos asynchronously.

Video Processing Workers (Cloud Run)

When a video upload event is published, a video processing worker will receive a message from Pub/Sub and transcode the video. For transcoding the video we will use **ffmpeg**, which is a popular open source tool for video processing and it's widely used in industry (including at YouTube).

The nature of video processing can lead to inconsistent workloads, so we will use Cloud Run to scale up and down as needed. Processed videos will be uploaded back to Cloud Storage.

Video Metadata (Firestore)

After a video is processed, we will store the metadata in Firestore. This will allow us to display processed videos in the web client along with other relevant info (e.g. title, description, etc).

Video API (Firebase Functions)

We will use Firebase Functions to build a simple API that will allow users to upload videos and retrieve video metadata. This can easily be extended to support additional Create, Read, Update, Delete (CRUD) operations.

Web Client (Next.js / Cloud Run)

We will use Next.js to build a simple web client that will allow users to sign in and upload videos. The web client will be hosted on Cloud Run.

Authentication (Firebase Auth)

We will use Firebase Auth to handle user authentication. This will allow us to easily integrate with Google Sign In.

Detailed Design

1. User Sign Up

Users can sign up using their Google account and this easily handled by Firebase Auth. A user record will be created including a unique auto-generated ID for the user, as well as the user's email address.

For us to include additional info about the user we will create a Firestore document for each user, which will be apart of the `users` collection. This will allow us to store additional info about the user (e.g. name, profile picture, etc).

Firebase Auth is mainly integrated into client code, which in our case will be within our Next.js app. While it's possible to trigger a user document creation directly from the client (after a new user signs up), this approach has some error prone edge cases.

What if there is an network issue or the user's browser crashes right after the user signs up but *before* the user document is created?

Fortunately, Firebase Auth provides **triggers** so that we don't have to rely on the client for this operation. We can just trigger a Firebase Function (i.e. a server-side endpoint) to create the user document whenever a new user is created.

2. Video Upload

Ideally, we should only allow authenticated users to upload videos. This will allow us to associate the uploaded video with the user who uploaded it. In the future this could also allow us to enforce quotas on video uploads (e.g. 10 videos per day).

While we could allow users to upload videos directly to a server we manage ourselves, it's more simple to use a service like Google Cloud Storage, which is specifically designed for arbitrarily large files like videos.

But to prevent unauthorized users from uploading videos, we will generate a **signed URL** that will allow the user to upload a video directly to Cloud Storage. We will implement this in a public Firebase Function, but this way we can easily ensure the user is authenticated before generating the URL.

> Note: We will invoke this function only *after* the user has specificed which video they want to upload. This is because we need to know the extension of the video file before we can generate the signed URL.

The signed URL can be used directly from the client to upload a video to our private Cloud Storage bucket for raw videos.

3. Video Processing

We'd like to process videos as soon as they come in, but it's possible that we could receive a large number of uploads at once which we can't immediately process. To solve this problem we will introduce a message queue to our system - Cloud Pub/Sub.

This provides many benefits:

1. When a video is uploaded to Cloud Storage, we can publish a message to a Pub/Sub topic. This will allow us to decouple the video upload from the video processing.
2. We will use a Pub/Sub subscriptions to ***push*** messages to our video processing workers. In the future we can add additional subscriptions to fan-out these messages, e.g. for analytics.

3. If the workers don't have enough capacity to process all the messages, Pub/Sub will automatically buffer the messages for us. This will allow us to scale our workers up and down as needed.

After a video is processed, we will upload it to a public Cloud Storage bucket. We will also store the video metadata in Firestore, including the video's processing status. This will allow us to display the processed videos in the web client.

There are some noteworthy limitations with this approach, such as:

- A Cloud Run request has a max timeout of 3600 seconds
- Pub/Sub will redeliver a message after at most 600 seconds, closing the previous HTTP connection
- We don't check for illegal content within videos

I won't discuss these in-depth here, but see the next section for more details.

Limitations & Future Work

Please see the **limitations** section for a detailed list of limitations and potential future tasks.

References

- Firebase Auth: **https://firebase.google.com/docs/auth**

- Cloud Storage Signed URLs: **https://cloud.google.com/storage/docs/access-control/signed-urls**
- Pub/Sub Push subscriptions: **https://cloud.google.com/pubsub/docs/push**
- Using Pub/Sub with Cloud Storage: **https://cloud.google.com/storage/docs/pubsub-notifications**
- Using Pub/Sub with Cloud Run: **https://cloud.google.com/run/docs/tutorials/pubsub**