[no-title]

Video streaming platforms are one of the most challenging web apps to develop. You can find resources that talk about designing the backend (all the major resources are for the backend only), but there is hardly any resource that talks about designing the frontend system.

Real-life examples:

- Youtube
- Netflix
- Amazon prime
- Disney + Hotstar
- Jio Cinema

From your favorite stand-up comedy to your favorite anime to your favorite sport, we can binge-watch all this from the comfort of any device and live thanks to the video streaming platforms.

Video streaming platforms are engineering marvels where the backend and frontend are equally difficult to design and develop as we have to deal with multiple complexities such as video encoding, adaptive streaming, content delivery networks, user interface design, and more.

In designing the frontend system of a video streaming platform like Youtube, it is crucial to prioritize a seamless user experience with intuitive navigation, responsive layouts, and fast loading times. Additionally, incorporating features like personalized recommendations, user-generated content interactions, and social sharing functionalities can enhance user engagement and retention on the platform.

It is a vast topic to be covered in this article. Rather, we will just cover the important things that are at the core of the frontend or the web app of any video streaming platform.

Requirements

Create a video streaming platform like YouTube or Amazon Prime where the user can browse through the recommendations, search for or query the videos, and play them.

Core features:

- Browse the genre-categorized list (home page).
- Recommendation list (single video page).
- Video playback.
- Autoplay video on the user interaction, like hover.

Nice-to-have features:

- To provide users with the best experience, they should be able to stream low-quality videos on the flaky networks.
- The webapp should be primarily built for the desktop but should support mobile and tablet-responsive views.
- Minimize stuttering and buffering.
- Fast video startup time.

A video streaming platform can be split into two different parts, each of which could be tackled independently.

- Video player or video playback feature: The core of any video streaming website.
- Website, excluding the video playback: The skeleton of the video streaming website.

We will first design the website's skeleton, and then we will discuss the design of the video player. This will help us cover all the features.

Designing the skeleton of the video streaming website

Requirements

- A browse page to view the videos by genre.
- A single video page that will have the video details and the recommendation list.
- The webpage should be responsive for mobile and tablet screen sizes and primarily designed for the desktop.

Architecture

A video streaming website heavily relies on SEO for growth and business, so the rendering strategy that will have to be applied should provide optimal SEO results. At the same time, video streaming websites are also very performance-intensive, so we have to keep them fast.

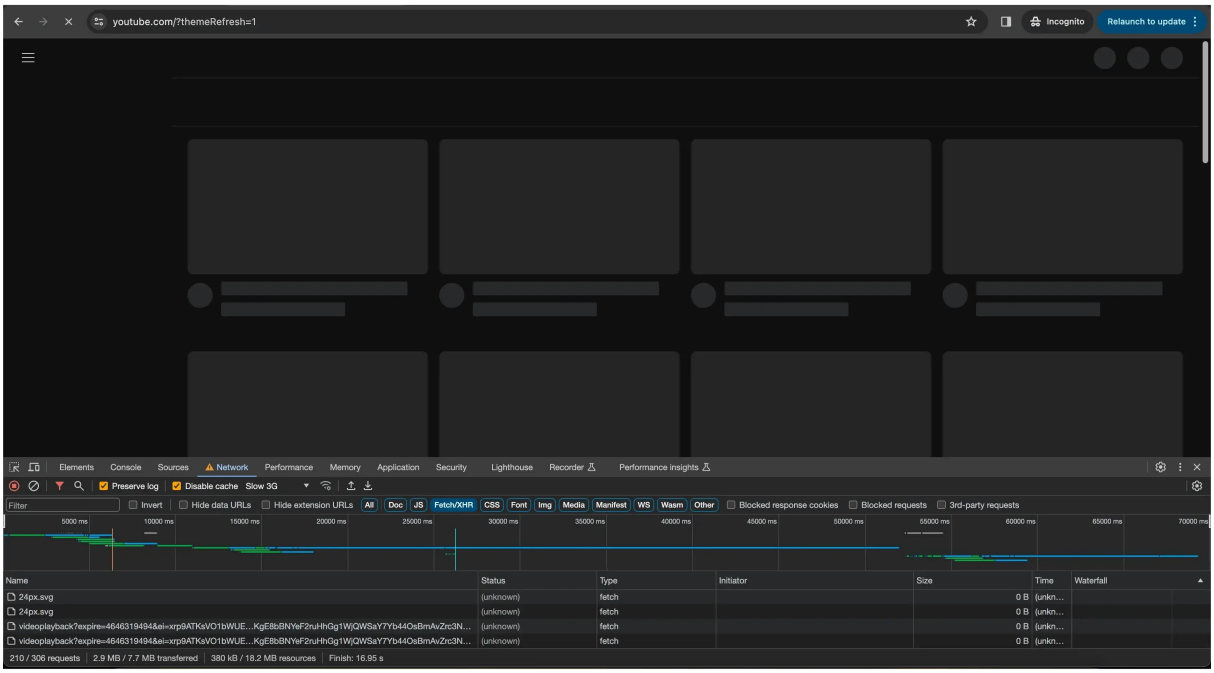
We can use any JavaScript framework that provides incremental static re-rendering, like Nextjs or NuxtJs, for developing the web app.

Rendering

Youtube uses server-side rendering with the 3-tier rendering strategy of "as little as possible, as early as possible," popularized by Facebook. Delivering only what we require is important, and we should aim to have the resources arrive just in time.

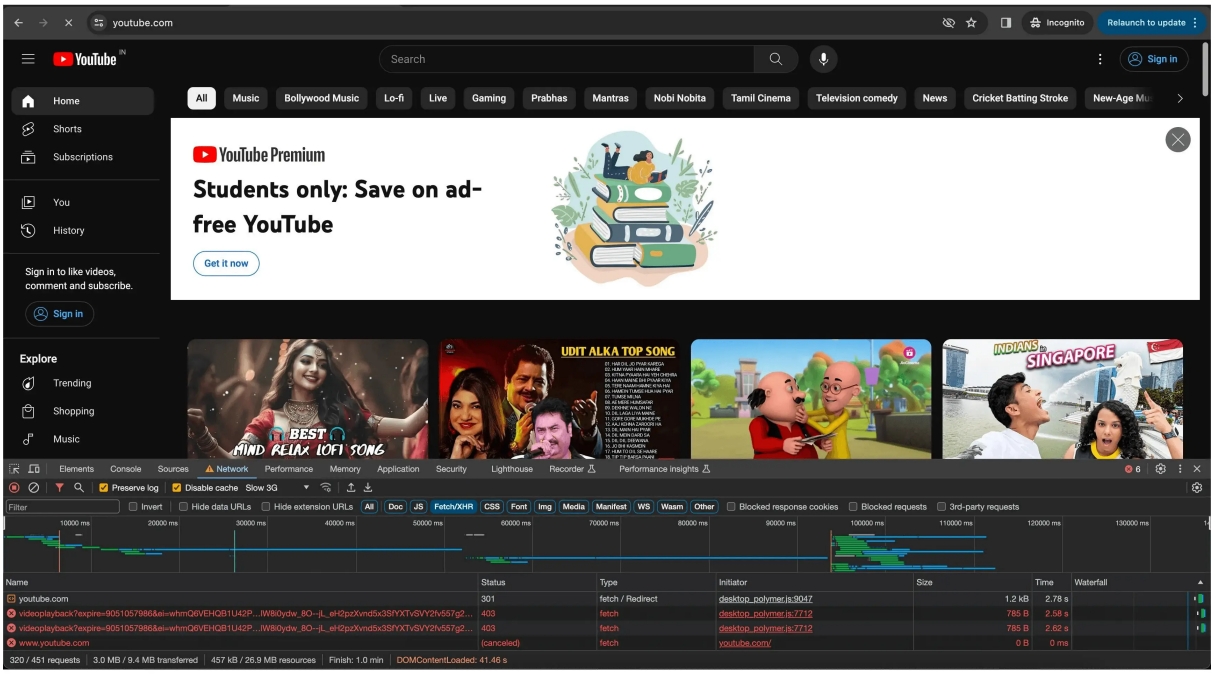
It splits the rendering into three parts:

In the first part, YouTube renders the skeleton UI on the server side, which loads the page extremely fast with all the necessary SEO meta and adhering to the core web vitals.

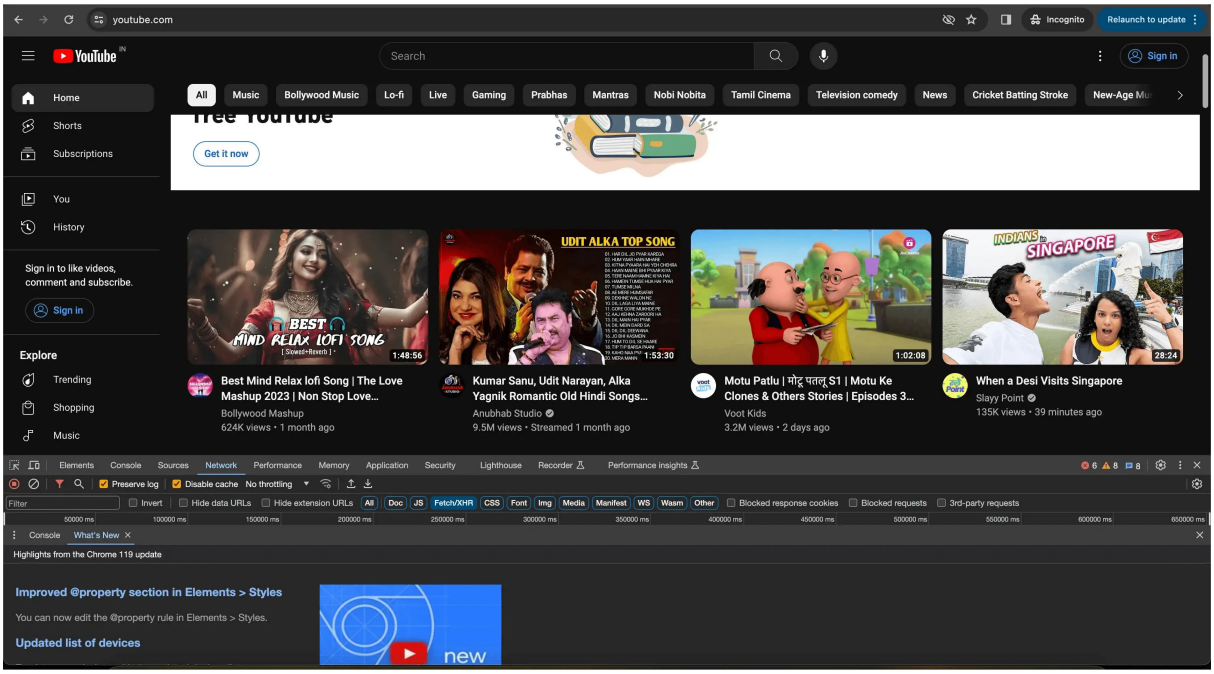


In the second part, it hydrates the skeleton UI with the video meta's like the text and the thumbnail, which are included in a JavaScript bundle.

When hovering over the video, you won't see the video duration, as it is just an image placeholder.

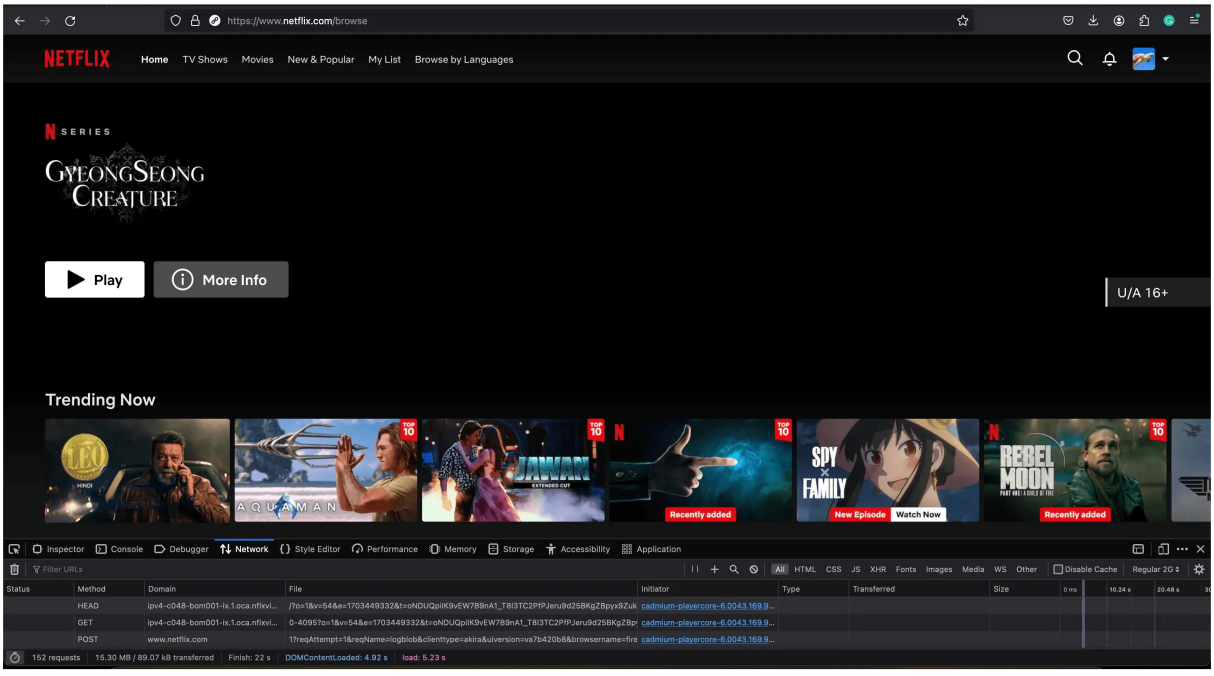


And after that, the playback bundle of JavaScript is loaded parallelly, which handles all user actions and hydrates the video cards with the video details so that they can be played on hover.



You can review the same by running YouTube.com in slow 3G network throttling.

Netflix, on the contrary, renders the HTML page with all the images and video meta data populated.



You will notice that on the slow network, the video thumbnails are still there, but once JavaScript is loaded and parsed, the images will scale on hover and the video will be autoplayed.

Examining both popular websites, it is clear that we can use server-side rendering along with progressive and selective hydration to get optimal results. Images can be pre-loaded so that they can be visible as early as possible.

What and how you want to do the hydration is completely subjective and comes from the various experiments and lots of researchs within the organizations.

This will provide us with the fast load time of client-side rendering and better SEO because of server-side rendering, providing a great experience to the users.

State-management

We are going to rely heavily on client-side caching for this type of application.

On the homepage of Netflix, except for the recommendation list and the new arrivals that can be updated frequently, all the genre lists can be cached for a given amount of time to boost performance.

Similary, the preference, the theme, and the settings of the website can be cached to avoid round-trips to the server.

Thus, we can use React-Query for query caching and Zustand or Redux for state management.

We can perform state management using the unidirectional flow of the Flux architecture, which Zustand and Redux both use.

Zustand provides a more native experience with hooks, making it easier to work with. Redux with the toolkit can also achieve the same, but it could become complex to scale the state.

The video list and recommendations can be in the shared state, while the video playback data will be in the local state as they will be specific to each video.

Styling

Given that all major video streaming platforms follow a design system for consistency and to provide the same user experience across different platforms, we can adopt the same.

All the components can be styled using styled-components, which helps to write CSS in JavaScript, making it easier to extend or mutate the styles of the existing components.

But you can also choose to have a separate style sheet for your application.

Because we have to create a responsive application, we can follow the approach of progressive enhancement while styling, which enforces mobile-first development and progressively updating the styles for different screen sizes. This results in less CSS.

If you are opting for the desktop first approach, you do the graceful degradation of the style, in which you will have to override the styles for lower screen sizes.

Learn about 3 different ways to write CSS in React.

Accessibility and internationalization

It is important to focus on accessibility and internationalization during component creation and styling so that each component is designed to provide the best user experience and support different languages and typographies.

Accessibility refers to the practice of making web content and applications usable for people with disabilities, ensuring that all users can access and navigate through the components easily. This includes considerations such as providing alternative text for images, using proper semantic markup, and ensuring keyboard accessibility.

Internationalization, on the other hand, involves adapting the components to different languages and cultural preferences, allowing users from various regions to understand and interact with the content effectively. By incorporating accessibility and internationalization into component creation and styling, developers can create inclusive.

The website should be accessible to all the different types of users with disabilities through keyboard, mouse, touch devices, and screen readers.

Similarly, users from different demographics and languages should also be supported. All types of fonts (Japanese is the most challenging language to handle as they write from top to bottom). We should also be able to provide support to the languages that are written from right to left, like Hebrew, Arabic, and Urdu.

Routing

From the requirements for the website, we can finalize the routes that we are going to need in the application.

To design the current system, we will need only two routes.

- Home or Browse page (/): It will display the new arrival list, the recommended video list, and the videos by genre.
- Single video page (/watch?video="video-slug"): This will show the single video details along with the video playback and suggestion list.

The reason why the single video page is taking the video URL as a query parameter on the watch page is that we can also pass additional details in the parameter, like whether the video is subtitled or not, the time from which the video should start playing, etc.

You can have it as a request parameter `/watch/:video-slug?q=""` which is also fine.

Depending on the platform, we can have public as well as private pages that require the user to be authenticated to access them. For example, in Netlfix, we can browse the homepage and see a part of the video or trailer, but we will have to login for full access.

We can define outlets in React-Router-V6 to redirect the routes if not authroized.

privateRoutes.js

```
import React from "react";
import { Outlet, Navigate } from "react-router-dom";
import useStore from "../store/user";

const Private = () => {
  const isLoggedIn = useStore((state) => state.isLoggedIn);
  return isLoggedIn ? <Outlet /> : <Navigate to="/login" />;
};

export default Private;
```

pulicRoutes.js

```
import React from 'react';
import { Outlet, Navigate } from 'react-router-dom';
import  useStore  from '../store/user';

const Public = () => {
  const isLoggedIn = useStore((state) => state.isLoggedIn);
```

```
  return !isLoggedIn ? <Outlet /> : <Navigate to="/dashboard" />;
};

export default Public;
```

route.js

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Login from "./pages/login";
import Signup from "./pages/signup";
import PrivateRoutes from "./routes/private";
import PublicRoutes from "./routes/public";

const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<PublicRoutes />}>
          <Route index element={<h1>Browse</h1>} />
          <Route path="login" element={<Login />}></Route>
          <Route path="signup" element={<Signup />}></Route>
        </Route>
        <Route path="/" element={<PrivateRoutes />}>
          <Route path="dashboard" element={<h1>Dashboard</h1>}></Route>
        </Route>
        <Route path="*" element={<NotFound />} />
      </Routes>
    </BrowserRouter>
  );
};

export default App;
```

There can be restrictions on the page as well, pushing the user to login or sign up to perform certain actions.

Components and modules

Referencing both YouTube and Netflix, or any video streaming website, we can think of the following common components:

- Card: Displays the video in the 16:9 ratio dimension or 4:3 dimension depending upon the resolution of the video; more on this later when we will discuss the video playback; thus, the dimension of this component will be dynamic. This should also play the video on user interactions like hovering.
- Video Player: The most important component that will be incorporated in the cards for autoplaying will also be discussed in detail.
- Video details accordion or drawer: This will display the video details upon clicking Show More or More Details.

Modules

:

We will need a single module to display the list of videos; now this list can be vertical (YouTube) as well as horizontal (YouTube and Netflix).

The card and the module will both be presentational components that will have the local state only for navigation or lazy loading of the details.

The data that has to be viewed in these components will be passed to them as props. This is known as the Hook/View component design pattern, which helps to maintain separation of concern while making the components reusable.

Authentication and authorization

In both applications, there is a provision to create a family account to onboard friends and family; thus, providing a good authentication and authorization experience is a vital part of the frontend application.

Youtube, being a part of the Google family, allows single sign-on on Gmail's credentials along with two-factor authentication.

Later, if you opt for the premium features, you can add limited users by using the email addresses of your friends and family to share it with them.

Which means the email address is the primary key for authentication and authorization on Youtube.

Similar to Netflix, you can login with an email and password for password-based authentication or a phone number + otp for password-less authentication, and then later take the email address as a secondary identifier.

Providing quick access to the application is key to user retention, and then, using the email or phone number, the account can be shared with a limited set of users.

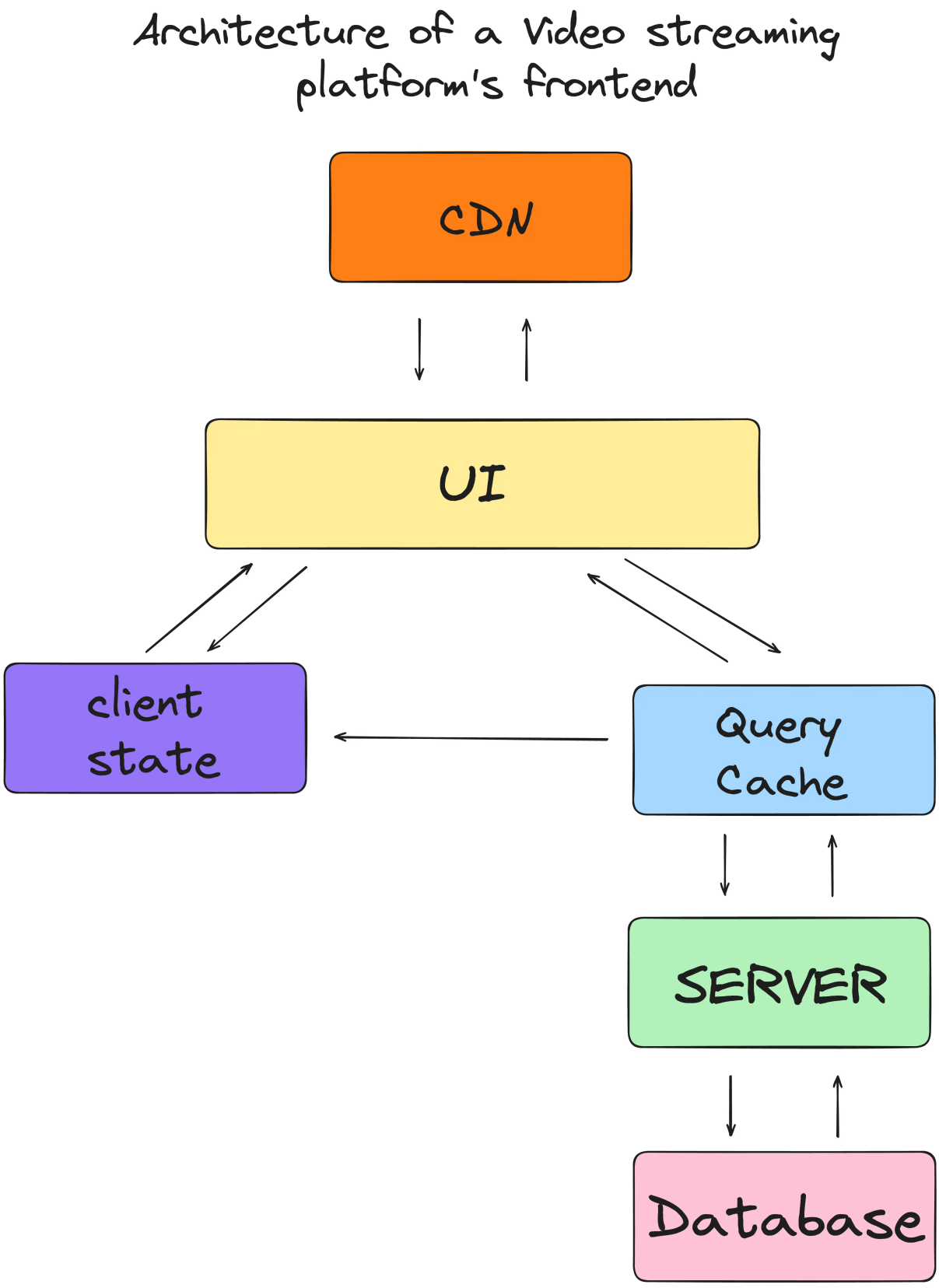
After a successful login, either token-based or cookie-based authentication can be used. A token-based approach is well suited for the microservice architecture.

Read more about authentication and authorization best practices.

Data Modelling

Data modeling for the video streaming platform is fairly straightforward, as we don't have to store any type of media on the client side. The CDN URL of the media will be provided, which we will have to consume for streaming.

The following can be considered as the architecture of the video streaming platform with the feature set that we are designing:



All the API responses will be cached using query-cache and will persist in local storage. We can also define the stale time for the cache to invalidate them and pull the new data.

Things that are common to the components, like the video's current playtime, playback rate, etc., will be stored in the local state.

Things that have to be shared, like auto-next-play, have closed captions enabled for all videos, etc., should be stored in the shared state.

Having a normalized state will really help to create a robust and scalable frontend application.

Exploring one of the most important features, which is the list of videos by recommendation, the new arrival, or the genre,.

There is no need to load all the data at once, but rather the twice-list of videos that will be visible at a time on the screen width can be pulled for each category, and all others can be lazy loaded.

To avoid redundancy and duplication of data, we will follow the byId and allId structures for state normalization.

A single video data schema could be like this:

```
{
  id: "xxx-xxx-xxx",
  thumbnail_url: "xxx-xxx-xxx.jpg",
  title: "abcd",
  description: "",
  created_on:"",
  updated_on:"",
  chapters: [],
  pulisher_details: {
  },
  video_meta: {
    available_resolutions: [],
    playback_speed: [],
    subtitles: [],
    closed_captions: []
  }
}
```

And in the normalized state, we can store the video details as IDs.

```
const normalized_data = {
  byIds: {
    "video-1": {
      id: "video-1",
      thumbnail_url: "xxx-xxx-xxx.jpg",
      title: "abcd",
      description: "",
      created_on:"",
      updated_on:"",
      chapters: [],
      pulisher_details: {
      },
      video_meta: {
        available_resolutions: [],
        playback_speed: [],
        subtitles: [],
        closed_captions: []
      }
    },
    "video-2": {
      id: "video-3",
      thumbnail_url: "xxx-xxx-xxx.jpg",
      title: "abcd",
```

```
      description: "",
      created_on:"",
      updated_on:"",
      chapters: [],
      pulisher_details: {
      },
      video_meta: {
        available_resolutions: [],
        playback_speed: [],
        subtitles: [],
        closed_captions: []
      }
    },
    "video-3": {
      id: "video-3",
      thumbnail_url: "xxx-xxx-xxx.jpg",
      title: "abcd",
      description: "",
      created_on:"",
      updated_on:"",
      chapters: [],
      pulisher_details: {
      },
      video_meta: {
        available_resolutions: [],
        playback_speed: [],
        subtitles: [],
        closed_captions: []
      }
    }
  }
}
```

And then, for each category, we can have an array with the list of video IDs.

```
const normalized_data = {
  byIds: {
    ...
  },
  categorized: {
    recommended: {
      label: "",
      description: "",
      videos: ["video-1", "video-2", "video-3"]
    },
    commedy: {
      label: "",
      description: "",
      videos: ["video-1", "video-2", "video-3"]
    },
    action: {
      label: "",
      description: "",
      videos: ["video-1", "video-2", "video-3"]
```

```
    },
  }
}
```

This will allow us to have a single source of data for the videos, and whenever a new video is fetched, it will be stored by ID, and its ID will be added to the list.

Also it allows fetching the list of videos in linear time and accessing the video details in constant time.

```
const getVideoById = (id) => {
  return normalized_data.byIds[id];
}

const getVideosByCategory = (category) => {
  return normalized_data.categorized[category].videos.map((e) => getVideoById(e));
}
```

On the browse page, use as little data as possible, and all the other video details can be pulled when the single video page is loaded.

As this data is specific to the video, it does not have to be shared and can be kept in the local state.

The recommendation list on the browse page and the single video page can be different, so they can be stored in separate categories.

Interface design (API)

To meet the requirements of the feature sets we are designing, we are going to need only two APIs.

To get the list of videos by category, we can have a single API with a query parameter that will fetch us the videos by genre or category.

1. Path: /list?category="recommendation"&start=0&limit=10, method: GET

```
request-payload: null
response:200 - ok
response-payload: [{
  id: "xxx-xxx-xxx",
  thumbnail_url: "xxx-xxx-xxx.jpg",
  title: "abcd",
  description: "",
  created_on:"",
  updated_on:"",
  chapters: [],
  pulisher_details: {
  },
  video_meta: {
    available_resolutions: [],
    playback_speed: [],
    subtitles: [],
    closed_captions: []
  }
}]
```

We have added the pagination option to the API to lazy load the list.

Second, to get the details of the single video.

Path: (/watch?v="video:slug") or (/watch/:video-slug), method: GET

```
request-payload: null
response:200 - ok
response-payload: {
  id: "xxx-xxx-xxx",
  thumbnail_url: "xxx-xxx-xxx.jpg",
  title: "abcd",
  description: "",
  created_on:"",
  updated_on:"",
  chapters: [],
  pulisher_details: {
    cast_details: {},
    crew_details: {},
  },
  video_meta: {
    resolutions: [],
    playback_speed: [],
    subtitles: [],
    closed_captions: [],
    audio_tracks: []
  }
}
```

It will contain the complete details of the video with the slug.

Optimization

Performance is the key to creating video streaming platforms; every aspect has to be considered where the application can be optimized.

- Using the CDN is the de facto here, as we are dealing with multiple large media assets, and serving them as soon as possible is extremely important. We can also utilize the multi-CDN architecture to split the load.
- On the initial rendering, we can reduce the JavaScript used to increase the performance. The bundle can be split into two parts, where the HTML with the skeleton UI is rendered on the server, and then the first bundle is loaded that hydrates the screen with the details, like rendering the components where we can have all the event listeners and user interactions.
- The video playback bundle has to be loaded separately from the main bundle and asynchronously, as it is the blocking resource for the videos to autoplay.
- The category list on the browse page should be lazy-loaded on the scroll. The videos in each list should also be lazy loaded; we can utilize list virtualization to avoid rendering large lists and have the best performance.
- Different sizes of media should be served according to the screen size for faster rendering on devices with a limited network.
- The mobile and tablet web apps can be independently developed and served separately.
- Browser-specific builds can be created to reduce the bundle size of the JavaScript. An older browser requires lots of polyfill to support the modern features that increase the bundle size.
- Skeleton UI helps to fix the cumlative layout shift, which is one of the core vitals to determining website performance. If you are not using Skeleton UI, you can set the dimensions on the media tags, like the image and the video.
- Images can be preloaded to avoid the delay in loading.

- Important JavaScript libraries like Reactjs can be prefetched using XHR when the HTML is parsed for faster UI rendering that boosts the time-to-Interactive metrics.
- Also, pages that have very few actions, like the logout page, can be built using plain HTML and vanilla JavaScript, like Netflix has done to improve their performance.

Designing the video playback feature.

Video player, or playback, is the core of video streaming platforms and requires extensive knowledge to provide the best possible user experience to the users.

In this part, we will see how the streaming moguls have designed their video player over time and what we should have knowledge about so that we can design the same.

Before exploring video player creation or designing the video playback experience, let us understand many things regarding media streaming over the internet.

These glossaries will help you understand things better:

- Streaming: Streaming refers to the process of continuously transmitting data, typically audio or video, over a network in real-time. It allows users to access and view content without having to download it fully before playback. This technology has revolutionized the way we consume media, providing instant access to a wide range of content on various devices. Understanding streaming protocols, codecs, and adaptive bitrate streaming is crucial for designing an efficient and seamless video playback experience.
- Buffering: Preloading video content in order to guarantee uninterrupted playback and avoid disruptions from sluggish network connections is known as buffering.
- Bandwidth: The maximum amount of data that can be transmitted over a network connection in a given amount of time is known as bandwidth. It is an important factor in determining the quality and speed of video streaming, as higher bandwidth allows for smoother playback and faster loading times.
- Resolution: The resolution of a video refers to the number of pixels displayed on the screen, determining the clarity and quality of the image. Higher resolutions, such as 4K or HD, provide sharper and more detailed visuals but require faster internet speeds and more bandwidth to stream smoothly. It is important for streaming services to offer different resolution options to accommodate varying network conditions and device capabilities.
- Bitrate: Bitrate is another crucial factor in video streaming, as it determines the amount of data transmitted per second. Higher bitrates result in better image quality, but they also require more bandwidth to stream without buffering or interruptions. Streaming platforms often adjust the bitrate dynamically based on the viewer's internet connection to ensure a seamless streaming experience.
- Frame rate: Frame per second (FPS), or frame rate, is the number of frames that are displayed in a second. A video is a group of images that render one after the other to provide motion. That is why videos are also known as motion pictures. The optimal experience is provided at 24 fps, but the best results are received at 30 fps and 60 fps. The higher the rate of frames, the more fluid the motion.
- Codec: A codec is a software or hardware device that compresses and decompresses digital media files. It is in charge of encoding the video and audio data into a format that the streaming platform can easily transmit and decode. Different codecs have different levels of compression and quality, with popular ones including H.264, VP9, and AV1. The choice of codec can impact the streaming quality and file size of the video content.
- Latency: Latency refers to the delay between the time a video is streamed and when it is actually displayed on the viewer's screen. It is an important factor in live streaming as it can affect the real-time interaction between the streamer and viewers. Lower latency is generally

preferred for live streaming applications, as it allows for more immediate and responsive communication.

- Display poster: A thumbnail image or poster of the video.
- Subtitles: A translated text of the audio in a different language that is displayed along the video, which is used by people who are not able to understand the audio to get a meaning. Everyone uses subtitles, not just the disabled.
- Closed Captions (CC): A form of subtitles that also includes texts representing the action, dialogues, and music along with the translated text of the audio. Like "[drum rolling, people with their heads down], I serve my lord to the life." Usually, deaf people use this to get a sense of the video through the described text.
- Playback control: Provided user interface action to control the video, like pause, play, stop, seek, volume control, etc.
- Seeking: The ability to move forward or backward in the video to a specific point in time. This feature is helpful for users who want to skip or review certain parts of the content.
- Scrubbing: The ability to view the frames or thumbnails of particular scenes or moments by dragging the playhead or seek bar of a video player.
- Picture-in-Picture mode: The PIP mode allows users to continue watching a video in a small, resizable window while performing other tasks on their device. This feature enhances multitasking capabilities and provides a convenient way to keep an eye on the video while using other applications. Additionally, it can be especially useful for users who want to watch videos while taking notes or browsing the internet.
- DRM: DRM (Digital Rights Management) is a technology that protects copyrighted content from unauthorized use or distribution. It ensures that only authorized users can access and view the content, preventing piracy and copyright infringement. DRM systems typically use encryption and licensing mechanisms to control the usage of digital media, such as videos, music, or ebooks. This helps content creators and distributors protect their intellectual property and generate revenue from their work.

Video formats

- WebM is a popular video streaming format for the web that uses VP8 or VP9 video codecs and Vorbis or Opus for audio. VP8 and VP9 codecs are known for efficient compression, making them suitable for online streaming with less bandwidth usage. All the major desktop browsers support WebM, while support on mobile and non-web devices is limited. Google created it specifically for the web.
- MP4 is the most popular video format, with universal support for all browsers on all devices. It is best for video storage, video editing, broadcasting, and streaming as it uses H.264 (or AVC) video codecs and AAC for audio. H.264 is widely regarded for its high compression efficiency and excellent video quality, even at lower bitrates.
- OGG is another popular video format that is known for its open-source nature and compatibility with multiple platforms. It uses the Theora video codec and Vorbis audio codec, providing good-quality compression and support for streaming. However, its adoption is not as widespread as MP4 due to limited support in some browsers and devices.

The simplest way to render and play a video on the web is by including a video source in the video HTML element. It can directly source the video in multiple formats and play it.

```
 <video width="320" height="240" controls>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
  Your browser does not support the video tag.
</video>
```

Multiple sources are added as fallback; if the video tag is not supported, then the text passed after the source will be displayed.

While it is possible to render a high-quality video with the native HTML5 video element, it does not provide the optimization to serve the video on the flaky internet.

Progressive video downloading

HTML5 video elements use the conventional approach of progressive video downloading, in which the videos are linearly streamed. The video will be downloaded completely, irrespective of the available network bandwidth of the user, and it will start playing as soon as it has enough data downloaded for uninterrupted playback. The video is played simultaneously as it is being downloaded.

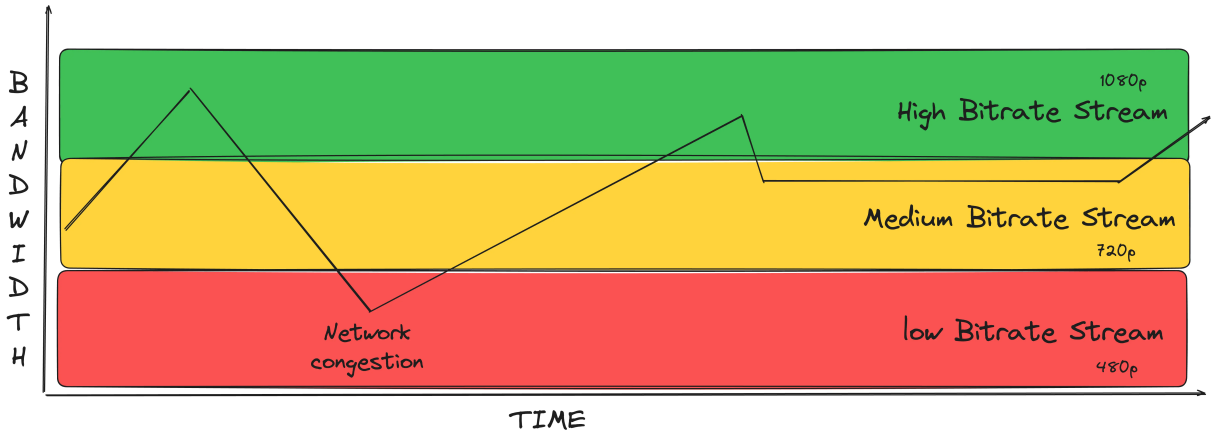
The video can be sought at any given point using the HTTP range request. The range request downloads the appropriate segment to resume the playback. Media players can benefit from an HTTP Range request, which instructs the server to return only a portion of an HTTP message to the client in order to facilitate random access to a file.

This conventional method of progressive download is a very expensive way of streaming video, as the same quality of video will be downloaded irrespective of the user's network bandwidth and device capabilities. For example, even if you are on a mobile device where you can view only 720p video, the full 1080p (HD) video will be downloaded.

Adaptive bitrate streaming

All the modern video streaming giants use adaptive bitrate streaming (ABR) to provide the best possible video streaming experience to their users, irrespective of their network bandwidth and device capabilities.

Adaptive bitrate streaming introduces dynamic delivery of the videos. With ABR, the video is split up into sections that are manageable, each of which is encoded at many different quality levels and sent to the client by the streaming server. The best-quality portion that can be broadcast smoothly at that moment is chosen by the streaming client, which is the video player that is currently being utilized, after analyzing the viewer's internet connection. This implies that in order to prevent interruptions to the video playing, the video quality is automatically modified if the viewer's internet speed decreases.



For example, an uploaded video is converted to multiple video formats, and then it is broken into multiple segments of length anywhere between 2 and 12 seconds. Which means a 2-hour-long video broken into chunks of 10 seconds each would yield 720 segments.

segments/

- 00001.ts
- 00002.ts
- 00003.ts
- 00004.ts

These segments will be downloaded as the user keeps streaming while maintaining a buffer in case of a fluctuating network connection.

ABR takes this one step further by creating a video of multiple resolutions (1080p, 720p, 480p, 360p, and 240p) of each format and creating segments in each of those.

1080p/

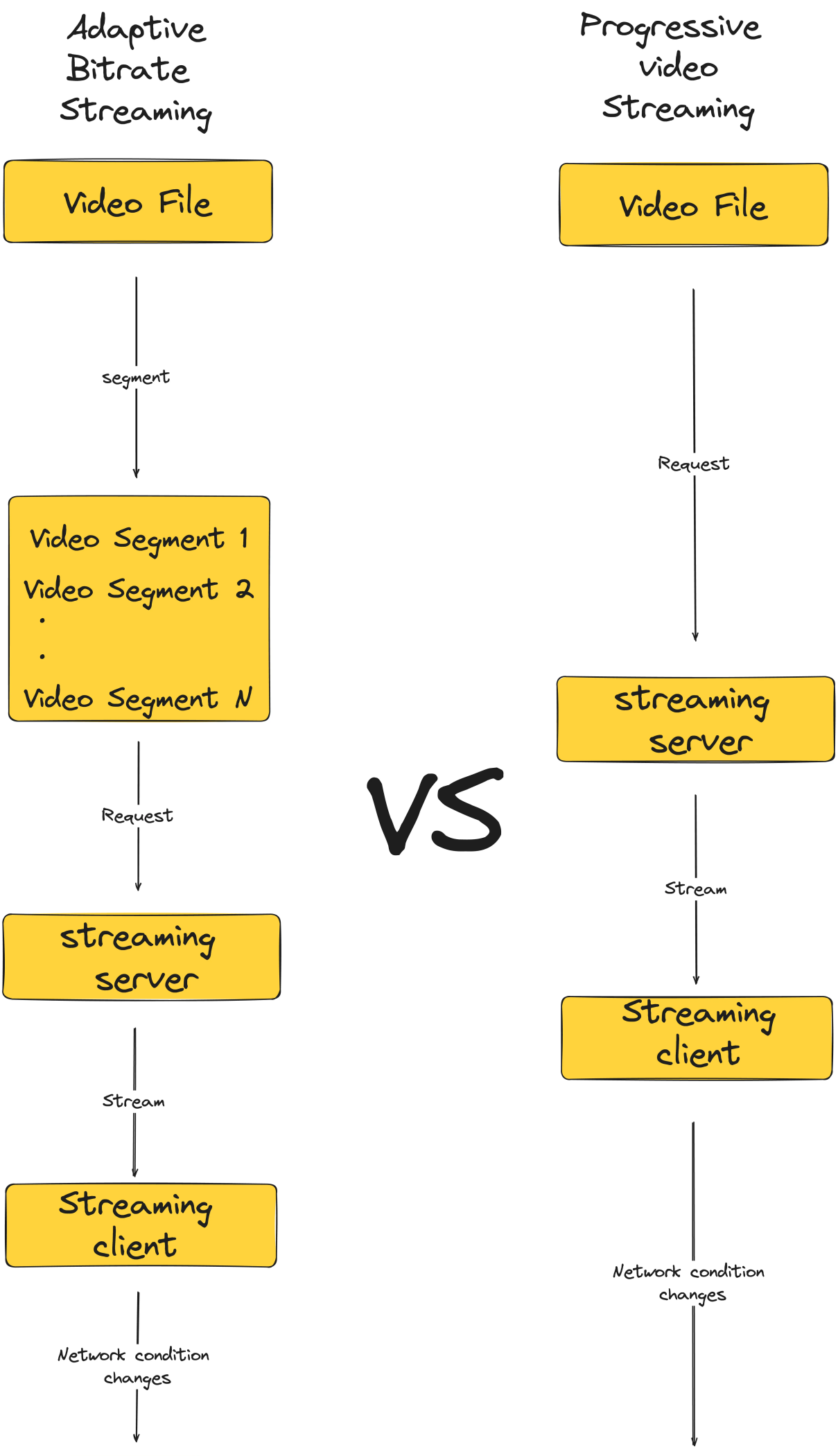
- 00001.ts
- 00002.ts
- 00003.ts
- 00004.ts

720p/

- 00001.ts
- 00002.ts
- 00003.ts
- 00004.ts

This makes it easier to serve the best-quality videos to the user depending on their network bandwidth and their device's support, providing the experience of viewing the same video on slower networks.

It also helps them to switch the video quality in case a poor network is encountered, as the next segment can be pulled from the lower quality. All that had to be made sure of was that all the segments were equally divided for each resolution.


Adaptive Bitrate Streaming
Video File
segment
Video Segment 1
Video Segment 2
Video Segment N
Request
streaming server
Stream
Streaming client
Network condition changes
VS
Progressive video Streaming
Video File
Request
streaming server
Stream
Streaming client
Network condition changes

The manifest files that store the segment details are in charge of handling this video segmentation. They follow a master slave pattern, where there is a master manifest file that has the details of all the available resolutions, and then a manifest file for each resolution has the segment details.

In adaptive streaming, the manifest files are passed to the video source, which contains details about the segments and the video resolution using which the videos will be streamed on-demand.

Sample maifest file of the DASH protocol (.mpd file):

Please enroll into the course to get the full access