How video streaming works on the web: An introduction

Paul Berberian Follow
Jan 10, 2018 · 14 min read

> *Note: this article is an introduction to video streaming in JavaScript and is mostly targeted to web developers. A large part of the examples here make use of HTML and modern JavaScript (ES6).*
> *If you're not sufficiently familiar with them, you may find it difficult to follow through, especially the code example.*
> *Sorry in advance for that.*

The need for a native video API

From the early to late 2000s, video playback on the web mostly relied on the flash plugin.

The Adobe Flash Player is required for video playback.
Get the latest Flash Player

Screen warning that the user should install the flash plugin, at the place of a video

This was because, at the time, there was no other mean to stream video on a browser. As a user, you had the choice between either installing third-party plugins like flash or Silverlight, or not being able to play any video at all.

To fill that hole, the WHATWG began to work on a new version of the HTML standard including, among other things, video and audio playback *natively* (read here: without any plugin). This trend was even

more accelerated following Apple stance on flash for its products.
This standard became what is now known as HTML5.

The HTML5 Logo. HTML5 would be changing the way videos are streamed on web pages

Thus HTML5 brought, among other things, the `<video>` tag to the web.

This new tag allows you to link to a video directly from the HTML, much like a `<img>` tag would do for an image.
This is cool and all but from a media website's perspective, using a simple img-like tag does not seem sufficient to replace our good ol' flash:

- we might want to switch between multiple video qualities on-the-fly (like YouTube does) to avoid buffering issues
- live streaming is another use case which looks really difficult to implement that way
- and what about updating the audio language of the content based on user preferences while the content is streaming like Netflix does?

Thankfully, all of those points can be answered *natively* on most browsers, thanks to what the HTML5 specification brought. This article will detail how today's web does it.

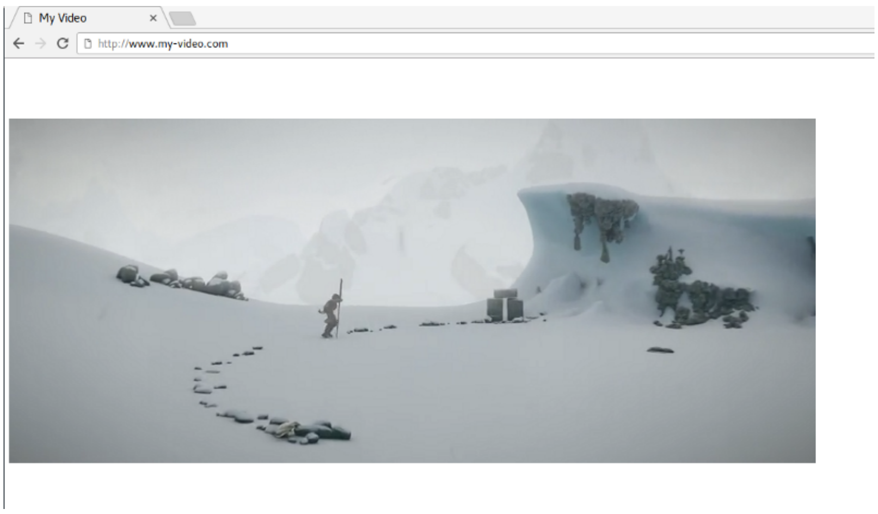
The video tag

As said in the previous chapter, linking to a video in a page is pretty straightforward in HTML5. You just add a video tag in your page, with few attributes.

For example, you can just write:

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>My Video</title>
  </head>
  <body>
    <video src="some_video.mp4" width="1280px" height="720px
```

This HTML will allow your page to stream *some_video.mp4* directly on any browser that supports the corresponding codecs (and HTML5, of course).
Here is what it looks like:

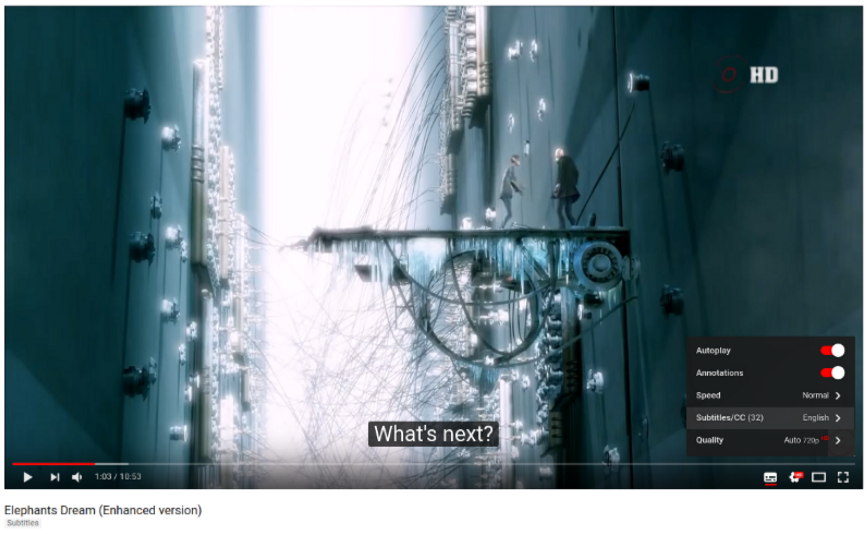


Simple page corresponding to the previous HTML code

This video tag also provides various APIs to e.g. play, pause, seek or change the speed at which the video plays.
Those APIs are directly accessible through JavaScript:

```
//pause the video
myVideo.pause()

// seek to 10 seconds
```

However, most videos we see on the web today display much more complex behaviors than what this could allow. For example, switching between video qualities and live streaming would be unnecessarily difficult there.



Autoplay
Annotations
Speed Normal
Subtitles/CC (32) English
Quality Auto 720p

Elephants Dream (Enhanced version)
Subtitles

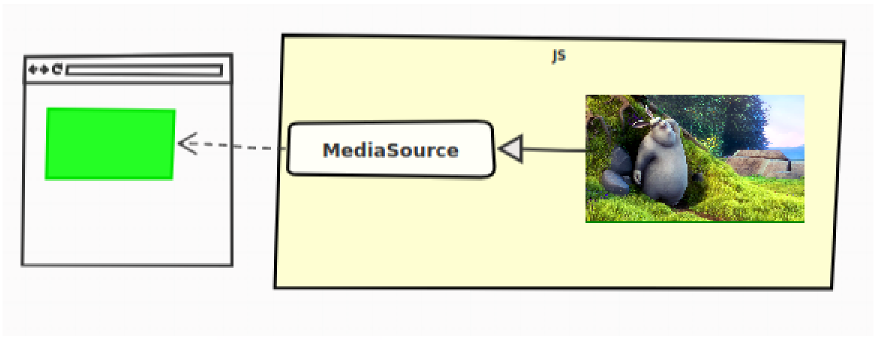
YouTube displays some more complex use-cases: quality switches subtitles a tightly controlled progressive-download of the video…

All those websites actually do still use the video tag. But instead of simply setting a video file in the *src* attribute, they make use of much more powerful web APIs, the **Media Source Extensions**.

The Media Source Extensions

The “Media Source Extensions” (more often shortened to just “MSE”) is a specification from the W3C that most browsers implement today. It was created to allow those complex media use cases directly with HTML and JavaScript.

Those “extensions” add the **MediaSource** object to JavaScript. As its name suggests, this will be the source of the video, or put more simply, this is the object representing our video’s data.



The video is here "pushed" to the MediaSource, which provides it to the web page

As written in the previous chapter, we still use the HTML5 video tag. Perhaps even more surprisingly, we still use its *src* attribute. Only this time, we're not adding a link to the video, we're adding a link to the **MediaSource** object.

You might be confused by this last sentence. We're not talking about an URL here, we're talking about an abstract concept from the JavaScript language, how can it be possible to refer to it as an URL on a video tag, which is defined in the HTML?

To allow this kind of use cases the W3C defined the `URL.createObjectURL` static method. This API allows creating an URL, which will actually refer not to a resource available online, but directly to a JavaScript object created on the client.

This is thus how a MediaSource is attached to a video tag:

```
const videoTag = document.getElementById("my-video");

// creating the MediaSource, just with the "new" keyword, an
const myMediaSource = new MediaSource();
const url = URL.createObjectURL(myMediaSource);

```

And that's it! Now you know how the streaming platforms play videos on the Web!
… Just kidding. So now we have the MediaSource, but what are we supposed to do with it?

The MSE specification doesn't stop here. It also defines another concept, the SourceBuffers.

The Source Buffers

The video is not actually directly "pushed" into the MediaSource for playback, SourceBuffers are used for that.

A MediaSource contains one or multiple instances of those. Each being associated with a type of content.

To stay simple, let's just say that we have only three possible types:

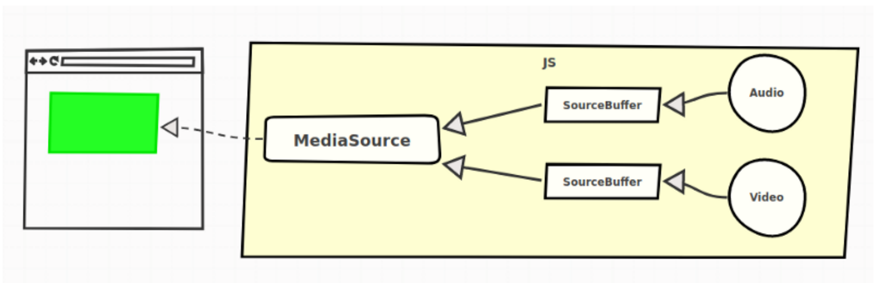
- *audio*

- *video*
- *both audio and video*

> *In reality, a “type” is defined by its MIME type, which may also include information about the media codec(s) used*

SourceBuffers are all linked to a single MediaSource and each will be used to add our video’s data to the HTML5 video tag directly in JavaScript.

As an example, a frequent use case is to have two source buffers on our MediaSource: one for the video data, and the other for the audio:



Relations between the video tag, the MediaSource, the SourceBuffers and the actual data

Separating video and audio allows to also manage them separately on the server-side. Doing so leads to several advantages as we will see later. This is how it works:

```javascript
// -- Create a MediaSource and attach it to the video (We a

const videoTag = document.getElementById("my-video");
const myMediaSource = new MediaSource();
const url = URL.createObjectURL(myMediaSource);
videoTag.src = url;

// 1. add source buffers

const audioSourceBuffer = myMediaSource
  .addSourceBuffer('audio/mp4; codecs="mp4a.40.2"');
const videoSourceBuffer = myMediaSource
  .addSourceBuffer('video/mp4; codecs="avc1.64001e"');

// 2. download and add our audio/video to the SourceBuffers

// for the audio SourceBuffer
fetch("http://server.com/audio.mp4").then(function(response
  // The data has to be a JavaScript ArrayBuffer
  return response.arrayBuffer();
```

And voila!
We're now able to manually add video and audio data dynamically to our video tag.

· · ·

> It's now time to write about the audio and video data itself. In the previous example, you might have noticed that the audio and video data were in the mp4 format.
> "mp4" is a *container format*, it contains the concerned media data but also multiple metadata describing, for example, the start time and duration of the media contained in it.

> The MSE specification does not dictate which format must be understood by the browser. For video data, the two most commons are **mp4** and **webm** files. The former is pretty well-known by now, the latter is sponsored by Google and based on the perhaps more known Matroska format (".mkv" files).

> Both are well-supported in most browsers.

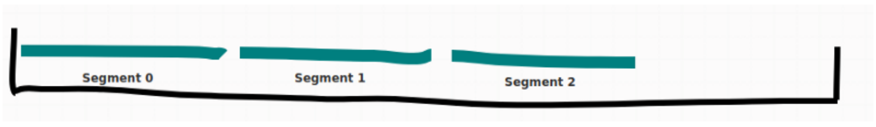
Media Segments

Still, many questions are left unanswered here:

- Do we have to wait for the whole content to be downloaded, to be able to push it to a SourceBuffer (and therefore to be able to play it)?
- How do we switch between multiple qualities or languages?
- How to even play live contents as the media isn't yet finished?

In the example from the previous chapter, we had one file representing the whole audio and one file representing the whole video. This can be enough for really simple use cases, but not sufficient if you want to go into the complexities offered by most streaming websites (switching languages, qualities, playing live contents etc.).

What actually happens in the more advanced video players, is that video and audio data are split into multiple "segments". These segments can come in various sizes, but they often represent between 2 to 10 seconds of content.



Artistic depiction of segments in a media file

All those video/audio segments then form the complete video/audio content. Those "chunks" of data add a whole new level of flexibility to our previous example: instead of pushing the whole content at once, we can just push progressively multiple segments.

Here is a ***simplified*** example:

This means that we also have those multiple segments on server-side. From the previous example, our server contains at least the following files:

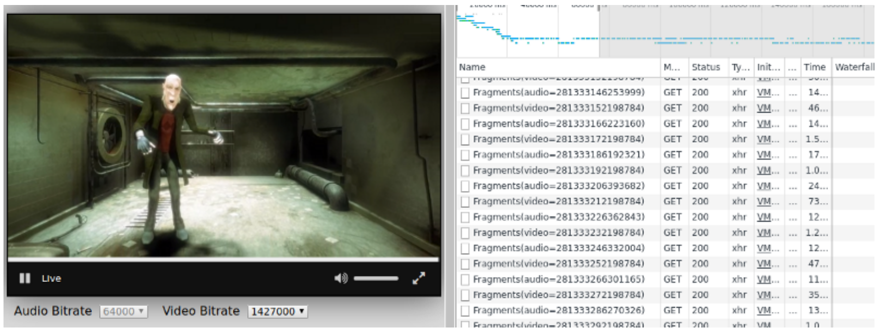
```
./audio/
  ├── segment0.mp4
  ├── segment1.mp4
  └── segment2.mp4

./video/
  └── segment0.mp4
```

*Note: The audio or video files might not truly be segmented on the server-side, the **Range** HTTP header might be used instead by the client to obtain those files segmented (or really, the server might do whatever it wants with your request to give you back segments).*
However, these cases are implementation details. We will here always consider that we have segments on the server-side.

All of this means that we thankfully do not have to wait for the whole audio or video content to be downloaded to begin playback. We often just need the first segment of each.

Of course, most players do not do this logic by hand for each video and audio segments like we did here, but they follow the same idea: downloading sequentially segments and pushing them into the source buffer.

A *funny* way to see this logic happen *in real life* can be to open the network monitor on Firefox/Chrome/Edge (on Linux or windows type “Ctrl+Shift+i” and go to the “Network” tab, on Mac it should be Cmd+Alt+i then “Network”) and then launching a video in your favorite streaming website.
You should see various video and audio segments being downloaded at a quick pace:



Audio Bitrate 64000 Video Bitrate 1427000

Name	M...	Status	Ty...	Init...	...	Time	Waterfall
Fragments(audio=281333146253999)	GET	200	xhr	VM...	...	14...	
Fragments(video=281333152198784)	GET	200	xhr	VM...	...	46...	
Fragments(audio=281333166223160)	GET	200	xhr	VM...	...	14...	
Fragments(video=281333172198784)	GET	200	xhr	VM...	...	1.5...	
Fragments(audio=281333186192321)	GET	200	xhr	VM...	...	17...	
Fragments(video=281333192198784)	GET	200	xhr	VM...	...	1.0...	
Fragments(audio=281333206393682)	GET	200	xhr	VM...	...	24...	
Fragments(video=281333212198784)	GET	200	xhr	VM...	...	73...	
Fragments(audio=281333226362843)	GET	200	xhr	VM...	...	12...	
Fragments(video=281333232198784)	GET	200	xhr	VM...	...	1.2...	
Fragments(audio=281333246332004)	GET	200	xhr	VM...	...	12...	
Fragments(video=281333252198784)	GET	200	xhr	VM...	...	47...	
Fragments(audio=281333266301165)	GET	200	xhr	VM...	...	11...	
Fragments(video=281333272198784)	GET	200	xhr	VM...	...	35...	
Fragments(audio=281333286270326)	GET	200	xhr	VM...	...	13...	

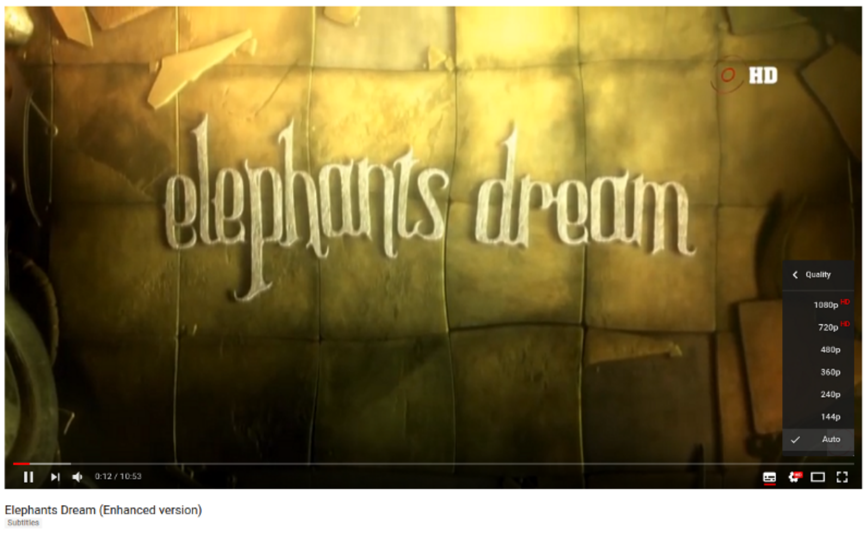
Screenshot of the Chrome Network tab on the Rx-Player's demo page

By the way, you might have noticed that our segments are just pushed into the source buffers without indicating WHERE, in terms of position in time, it should be pushed.
The segments' containers do in fact define, among other things, the time where they should be put in the whole media. This way, we do not have to synchronize it at hand in JavaScript.

Adaptive Streaming

Many video players have an "auto quality" feature, where the quality is automatically chosen depending on the user's network and processing capabilities.

This is a central concern of a web player called *adaptive streaming*.



YouTube "Quality" setting. The default "Auto" mode follows adaptive streaming principles

This behavior is also enabled thanks to the concept of media segments.

On the server-side, the segments are actually encoded in multiple qualities. For example, our server could have the following files stored:

```
./audio/
  ├── ./128kbps/
  |      ├── segment0.mp4
  |      ├── segment1.mp4
  |      └── segment2.mp4
  └── ./320kbps/
         ├── segment0.mp4
         ├── segment1.mp4
         └── segment2.mp4
```

```
./video/
  ├── ./240p/
  |      ├── segment0.mp4
  |      ├── segment1.mp4
  |      └── segment2.mp4
  └── ./720p/
         ├── segment0.mp4
         ├── segment1.mp4
         └── segment2.mp4
```

A web player will then automatically choose the right segments to download as the network or CPU conditions change.

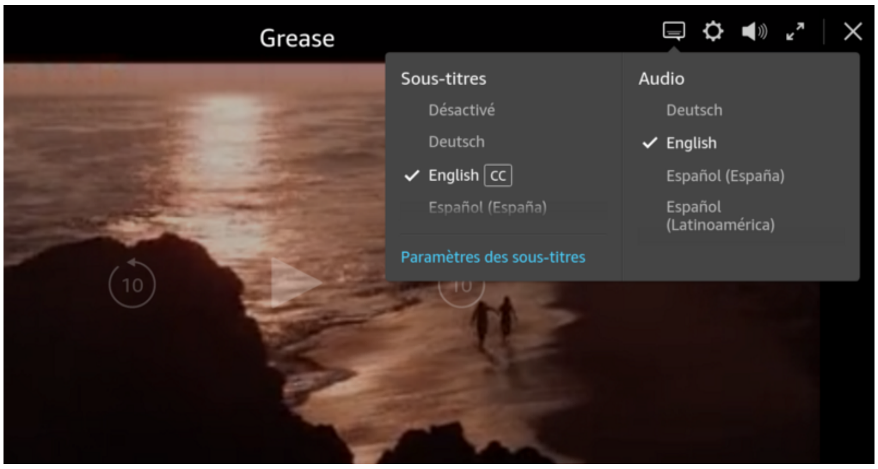
This is entirely done in JavaScript. For audio segments, it could, for example, look like that:

```javascript
/**
 * Push audio segment in the source buffer based on its num
 * and quality
 * @param {number} nb
 * @param {string} language
 * @param {string} wantedQuality
 * @returns {Promise}
 */
function pushAudioSegment(nb, wantedQuality) {
 // The url begins to be a little more complex here:
 const url = "http://my-server/audio/" +
 wantedQuality + "/segment" + nb + ".mp4");
return fetch(url)
 .then((response) => response.arrayBuffer());
 .then(function(arrayBuffer) {
 audioSourceBuffer.appendBuffer(arrayBuffer);
 });
}

/**
 * Translate an estimated bandwidth to the right audio
 * quality as defined on server-side.
 * @param {number} bandwidth
 * @returns {string}
 */
```

As you can see, we have no problem putting together segments of different qualities, everything is transparent on the JavaScript-side here. In any case, the container files contain enough information to allow this process to run smoothly.

Switching between languages

On more complex web video players, such as those on Netflix, Amazon Prime Video or MyCanal, it’s also possible to switch between multiple audio languages depending on the user settings.



Example of language options in Amazon Prime Video

Now that you know what you know, the way this feature is done should seem pretty simple to you.

Like for adaptive streaming we also have a multitude of segments on the server-side:

```
./audio/
  ├── ./esperanto/
  |       ├── segment0.mp4
  |       ├── segment1.mp4
  |       └── segment2.mp4
  └── ./french/
          ├── segment0.mp4
          ├── segment1.mp4
          └── segment2.mp4

./video/
  ├── segment0.mp4
  ├── segment1.mp4
  └── segment2.mp4
```

This time, the video player has to switch between language not based on the client’s capabilities, but on the user’s preference.

For audio segments, this is what the code could look like on the client:

```
// ...

/**
 * Push audio segment in the source buffer based on its num
 * @param {number} nb
 * @param {string} language
 * @returns {Promise}
 */
function pushAudioSegment(nb, language) {
  // construct dynamically the URL of the segment
  // and push it to the SourceBuffer
  const url = "http://my-server/audio/" +
    language + "/segment" + nb + ".mp4"
  return fetch(url);
    .then((response) => response.arrayBuffer());
    .then(function(arrayBuffer) {
      audioSourceBuffer.appendBuffer(arrayBuffer);
```

You may also want to “clear” the previous SourceBuffer’s content when switching a language, to avoid mixing audio contents in multiple languages.

This is doable through the *SourceBuffer.prototype.remove* method, which takes a starting and ending time in seconds:

```
// remove pushed content from 0 to 40s
audioSourceBuffer.remove(0, 40);
```

Of course, it’s also possible to combine both adaptive streaming and multiple languages. We could have our server organized as such:

```
./audio/
  ├── ./esperanto/
  |     ├── ./128kbps/
  |     |     ├── segment0.mp4
  |     |     ├── segment1.mp4
  |     |     └── segment2.mp4
  |     └── ./320kbps/
  |           ├── segment0.mp4
  |           ├── segment1.mp4
  |           └── segment2.mp4
  └── ./french/
        ├── ./128kbps/
        |     ├── segment0.mp4
        |     ├── segment1.mp4
        |       └── segment2.mp4
        └── ./320kbps/
                ├── segment0.mp4
                ├── segment1.mp4
                └── segment2.mp4


./video/
  ├── ./240p/
  |       ├── segment0.mp4
  |       ├── segment1.mp4
  |       └── segment2.mp4
  └── ./720p/
          ├── segment0.mp4
          ├── segment1.mp4
          └── segment2.mp4
```

And our client would have to manage both languages and network conditions instead:

```javascript
/**
 * Push audio segment in the source buffer based on its num
 * @param {number} nb
 * @param {string} language
 * @param {string} wantedQuality
 * @returns {Promise}
 */
function pushAudioSegment(nb, language, wantedQuality) {
  // The url begins to be a little more complex here:
  const url = "http://my-server/audio/" +
    language + "/" + wantedQuality + "/segment" + nb + ".mp

  return fetch(url)
    .then((response) => response.arrayBuffer());
    .then(function(arrayBuffer) {
      audioSourceBuffer.appendBuffer(arrayBuffer);
    });
```

As you can see, there’s now a lot of ways the same content can be defined.

This uncovers another advantage separated video and audio segments have over whole files. With the latter, we would have to combine every possibility on the server-side, which might take a lot more space:

```
segment0_video_240p_audio_esperanto_128kbps.mp4
segment0_video_240p_audio_esperanto_320kbps.mp4
```

```
  segment0_video_240p_audio_french_128kbps.mp4
  segment0_video_240p_audio_french_320kbps.mp4
  segment0_video_720p_audio_esperanto_128kbps.mp4
  segment0_video_720p_audio_esperanto_320kbps.mp4
  segment0_video_720p_audio_french_128kbps.mp4
  segment0_video_720p_audio_french_320kbps.mp4
  segment1_video_240p_audio_esperanto_128kbps.mp4
  segment1_video_240p_audio_esperanto_320kbps.mp4
  segment1_video_240p_audio_french_128kbps.mp4
  segment1_video_240p_audio_french_320kbps.mp4
  segment1_video_720p_audio_esperanto_128kbps.mp4
  segment1_video_720p_audio_esperanto_320kbps.mp4
  segment1_video_720p_audio_french_128kbps.mp4
  segment1_video_720p_audio_french_320kbps.mp4
  segment2_video_240p_audio_esperanto_128kbps.mp4
  segment2_video_240p_audio_esperanto_320kbps.mp4
  segment2_video_240p_audio_french_128kbps.mp4
  segment2_video_240p_audio_french_320kbps.mp4
  segment2_video_720p_audio_esperanto_128kbps.mp4
  segment2_video_720p_audio_esperanto_320kbps.mp4
  segment2_video_720p_audio_french_128kbps.mp4
  segment2_video_720p_audio_french_320kbps.mp4
```

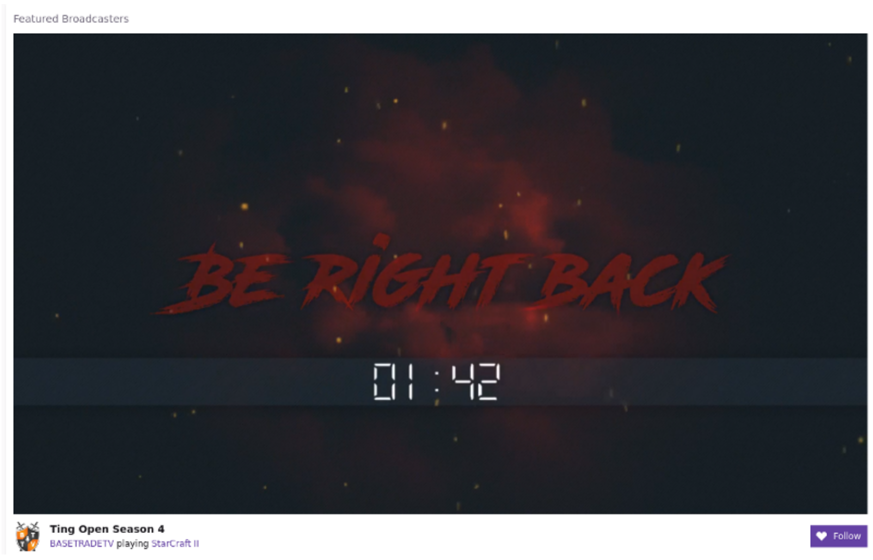
Here we have more files, with a lot of redundancy (the exact same video data is included in multiple files).

This is as you can see highly inefficient on the server-side. But it is also a disadvantage on the client-side, as switching the audio language might lead you to also re-download the video with it (which has a high cost of bandwidth).

Live Contents

We didn’t talk about live streaming yet.

Live streaming on the web is becoming very common (twitch.tv, YouTube live streams…) and is again greatly simplified by the fact that our video and audio files are segmented.



Screenshot taken from twitch.tv, which specializes in video game live streaming

To explain how it basically works in the simplest way, let’s consider a YouTube channel which had just begun streaming 4 seconds ago.

If our segments are 2 seconds long, we should already have two audio segments and two video segments generated on YouTube’s server:

- Two representing the content from 0 seconds to 2 seconds (1 audio + 1 video)
- Two representing it from 2 seconds to 4 seconds (again 1 audio + 1 video)

```
./audio/
  ├── segment0s.mp4
  └── segment2s.mp4

./video/
  ├── segment0s.mp4
  └── segment2s.mp4
```

At 5 seconds, we didn’t have time to generate the next segment yet, so for now, the server has the exact same content available.

After 6 seconds, a new segment can be generated, we now have:

```
./audio/
  ├── segment0s.mp4
```

```
    ├── segment2s.mp4
    └── segment4s.mp4
```

```
  ./video/
    ├── segment0s.mp4
    ├── segment2s.mp4
    └── segment4s.mp4
```

This is pretty logical on the server-side, live contents are actually not really continuous, they are segmented like the non-live ones but segments continue to appear progressively as time evolves.

Now how can we know from JS what segments are available at a certain point in time on the server?

We might just use a clock on the client and infer as time goes when new segments are becoming available on the server-side.
We would follow the “*segmentX.mp4*" naming scheme, and we would increment the “*X*” from the last downloaded one each time (*segment0.mp4*, then, 2 seconds later, *Segment1.mp4* etc.).

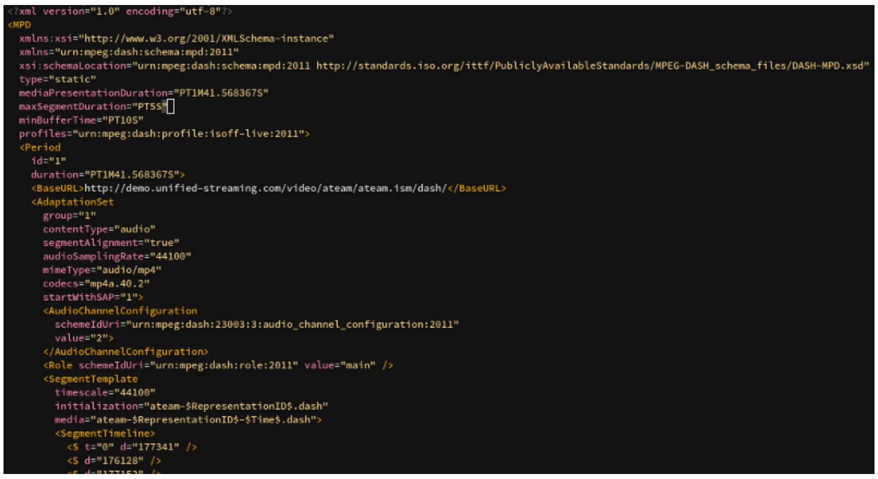
In many cases, however, this could become too imprecise: media segments may have variable durations, the server might have latencies when generating them, it might want to delete segments which are too old to save space…
As a client, you want to request the latest segments as soon as they are available while still avoiding requesting them too soon when they are not yet generated (which would lead to a 404 HTTP error).

This problem is usually resolved by using a transport protocol (also sometimes called Streaming Media Protocol).

Transport Protocols

Explaining in depth the different transport protocol may be too verbose for this article. Let’s just say that most of those have the same core concept: the **Manifest**.

A **Manifest** is a file describing which segments are available on the server.



```xml
<?xml version="1.0" encoding="utf-8"?>
<MPD
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="urn:mpeg:dash:schema:mpd:2011"
  xsi:schemaLocation="urn:mpeg:dash:schema:mpd:2011 http://standards.iso.org/ittf/PubliclyAvailableStandards/MPEG-DASH_schema_files/DASH-MPD.xsd"
  type="static"
  mediaPresentationDuration="PT1M41.568367S"
  maxSegmentDuration="PT5S"
  minBufferTime="PT10S"
  profiles="urn:mpeg:dash:profile:isoff-live:2011">
  <Period
    id="1"
    duration="PT1M41.568367S">
    <BaseURL>http://demo.unified-streaming.com/video/ateam/ateam.ism/dash/</BaseURL>
    <AdaptationSet
      group="1"
      contentType="audio"
      segmentAlignment="true"
      audioSamplingRate="44100"
      mimeType="audio/mp4"
      codecs="mp4a.40.2"
      startWithSAP="1">
      <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011"
        value="2">
      </AudioChannelConfiguration>
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main" />
      <SegmentTemplate
        timescale="44100"
        initialization="ateam-$RepresentationID$.dash"
        media="ateam-$RepresentationID$-$Time$.dash">
        <SegmentTimeline>
          <S t="0" d="177341" />
          <S d="176128" />
```

Example of a DASH Manifest, based on XML

With it, you can describe most things we learn in this article:

- Which audio languages the content is available in and where they are on the server (as in, "at which URL")
- The different audio and video qualities available
- And of course, what segments are available, in the context of live streaming

The most common transport protocols used in a web context are:

- ***DASH***
  used by YouTube, Netflix or Amazon Prime Video (and many others). DASH' manifest is called the Media Presentation Description (or MPD) and is at its base XML.
  The DASH specification has a great flexibility which allows MPDs to support most use cases (audio description, parental controls) and to be codec-agnostic.
- ***HLS***
  Developed by Apple, used by DailyMotion, Twitch.tv, and many others. The HLS manifest is called the playlist and is in the m3u8 format (which are m3u playlist files, encoded in UTF-8).
- ***Smooth Streaming***
  Developed by Microsoft, used by multiple Microsoft products and MyCanal. In Smooth Streaming, manifests are called… Manifests and are XML-based.

In the real—web— world

As you can see, the core concepts behind videos on the web lay on media segments being pushed dynamically in JavaScript.

This behavior becomes quickly pretty complex, as there’s a lot of features a video player has to support:

- it has to download and parse some sort of manifest file
- it has to guess the current network conditions
- it needs to register user preferences (for example, the preferred languages)
- it has to know which segment to download depending on at least the two previous points
- it has to manage a segment pipeline to download sequentially the right segments at the right time (downloading every segment at the same time would be inefficient: you need the earliest one sooner than the next one)
- it has also to deal with subtitles, often entirely managed in JS
- Some video players also manage a thumbnails track, which you can often see when hovering the progress bar
- Many services also require DRM management
- and many other things…

Still, at their core, complex web-compatible video players are all based on MediaSource and SourceBuffers.

Their web players all make use of MediaSources and SourceBuffers at their core

That’s why those tasks are usually performed by libraries, which do just that.
More often than not, those libraries do not even define a User Interface. They mostly provide a rich APIs, take the Manifest and various preferences as arguments, and push the right segment at the right time in the right source buffers.

This allows a greater modularization and flexibility when designing media websites and web application, which, in essence, will be complex front-ends.

Open-source web video players

There are many web video players available today doing pretty much what this article explains. Here are various open-source examples:

- rx-player: Configurable player for both DASH and Smooth Streaming contents. Written in TypeScript— Shameless self-plug as I’m one of the dev.
- dash.js: Play DASH contents, support a wide range of DASH features. Written by the DASH Industry Forum, a consortium promoting inter-operability guidelines for the DASH transport protocol.
- hls.js: well-reputed HLS player. Used in production by multiple big names like Dailymotion, Canal+, Adult Swim, Twitter, VK and more.
- shaka-player: DASH and HLS player. Maintained by Google.

By the way, Canal+ is hiring! If working with that sort of stuff interests you, take a look at http://www.vousmeritezcanalplus.com/ (⚠ French website).