Introduction

When HTML first introduced the `video` tag I was pumping my fist in joy. No more plugins to play video content. It was as simple as creating marlup like the one below to play and video in an MP4/ACC container with English and Swedish subtitles that can be changed as needed.

```
<!-- Video with subtitles -->
<video src="foo.mp4" poster="foo-poster.png"
       width="640" height="480" controls>
  <track kind="subtitles" src="foo.en.vtt" srclang="en" label="English">
  <track kind="subtitles" src="foo.sv.vtt" srclang="sv" label="Svenska">
</video>
```

But it wass never as simple as it looked. Because there was no standard video format for HTML5 video, different browsers supported different container formats and different audio and video codecs. So the video turned into something like this:

```
<video height="480" width="640" controls
  poster="https://archive.org/download/WebmVp8Vorbis/webmvp8.gif" >
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8.webm"
    type="video/webm">
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8_512kb.mp4"
    type="video/mp4">
  <source
    src="https://archive.org/download/WebmVp8Vorbis/webmvp8.ogv"
    type="video/ogg">
  Your browser doesn't support HTML5 video tag.
</video>
```

which produces the following video player:

Your browser doesn't support HTML5 video tag.

Each source element loads a different version of the video encoded with a different set of audio and video codecs. These files must be encoded separately and hosted separately.

There are also patent issues around MP4/h264 and ACC codecs. The MPEG Licensing Authority create a "patent pool" of essential technologies for MP4 encoding and decoding.

I had hoped that the new HEVC/h265 technology would not be encumbered by MPEG-LA style patent trolls but it was too much, apparently, as MPEG-LA already has an HVEC patent pool

So the fight has remained a stalemate with Mozilla and Opera on one side who refuse to pay the MP4 licensing fee and Microsoft, Google and Apple who have caved in and support MPEG4 playback as part of their HTML5 video implementations.

So, if it's not MPEG4 or HVEC/h265 then what alternatives do we have

available?

While Google implements MPEG4 in Chrome it has not remained static in the video codec front. In 2009 Google purchased On2 Technologies and have worked hard to make VP8, VP9 and its successor, WebM

MPEG-LA must have seen the benefit of VP8 becausse they began forming a patent pool for the technology. Google didn't like that and the conflict ended with an agreement that would remove the MPEG-LA as a factor in VP8 licensing so that Google can continue to offer the code free and unencumbered for personal and commercial use, for now.

Why is this important?

> Because this means that VP8 is a hell of lot safer and more free from possible legal repercussions than H.264 itself. What many H.264 proponents do not understand, either wilfully or out of sheer ignorance, is that those H.264 licenses embedded in Windows, OS X, iOS, your 'professional' camera, and so on, do not cover commercial use. If you shoot a video with your camera in H.264, upload it to YouTube, and get some income from advertisements, you're in violation of the H.264 license (and the MPEG-LA made it clear they had no qualms about going after individual users). The extension the MPEG-LA announced (under pressure from VP8 and WebM) changed nothing about that serious legal limitation.
>
> — Google called the MPEG-LA's bluff, and won
>
> Why Our Civilization's Video Art and Culture is Threatened by the MPEG-LA

The other codec worth looking at (mostly because it's supported by Firefox) is OGG Theora from the Xiph.org Foundation. Like VP8 and WebM Theora is free and unencumbered by patents.

MP4 containers can be optimized for a kind of pseudo streaming by re-arranging the "atoms" of the movie (atoms, in this context, are the chunks of data that make up the movie). The video player is looking or the `moov` atom and will not play the movie until it finds it.

If your server is configured for HTTP Range Requests it will request smaller

chunks until it find the atom it needs.

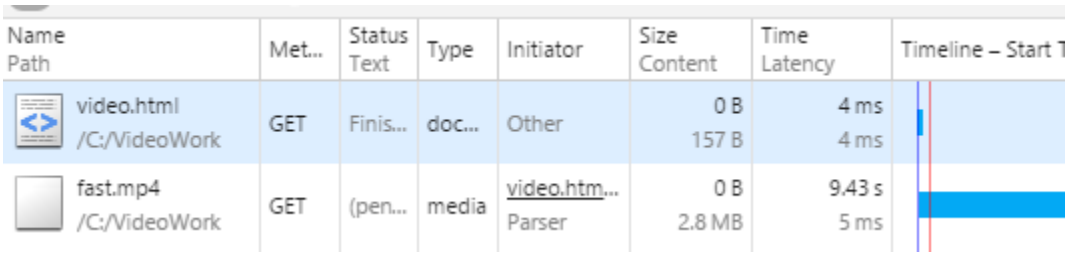

Figure 1: Different requests for the same video

Unfortunately for on-demand movies the `moov` atom is at the end of the file. So if the server is not configured to handle range requests then the player will have to download the complete file before it can start playing it.

If you're using already made content or don't want to re-encode the video you can use tools like Handbrake to optimize the video file for streaming across the web by moving the `moov` atom to the beginning of the file.

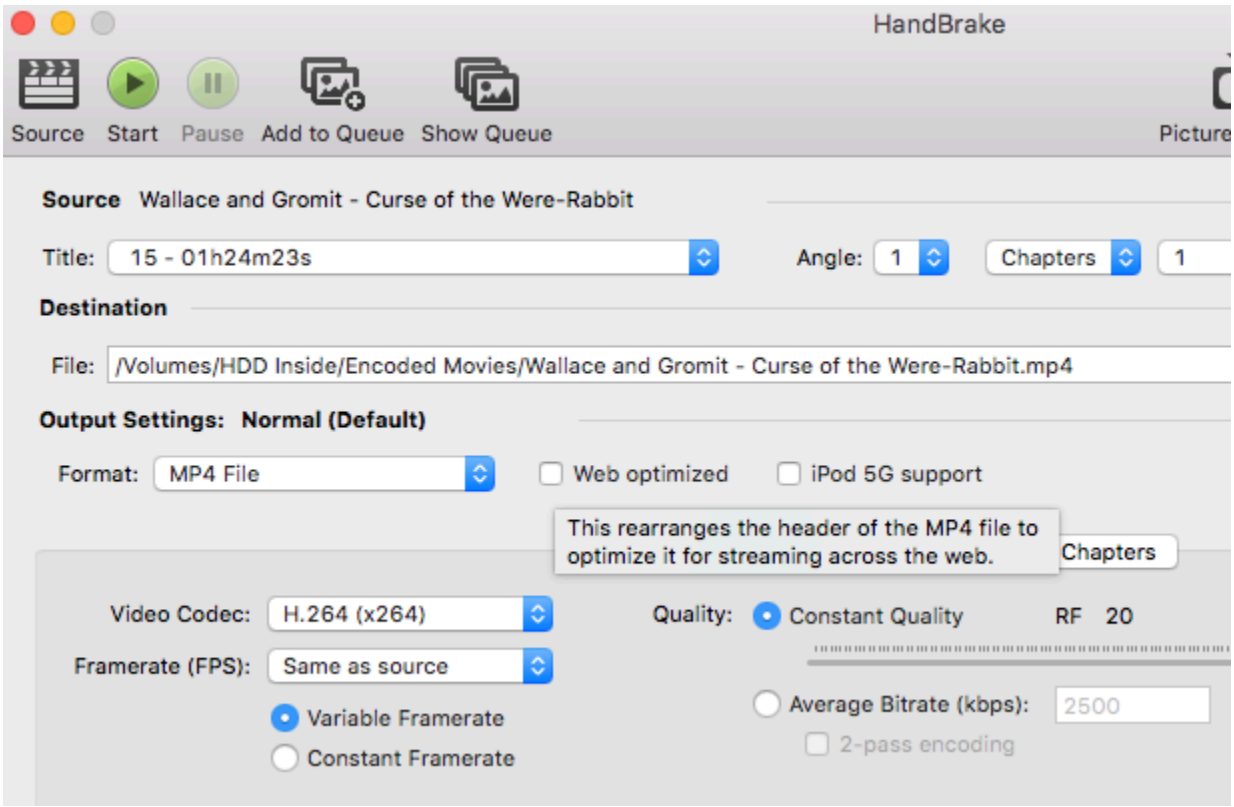

Figure 2: Using Handbrake to re-arrange the video atoms

If you're working with multiple files or are more comfortable you can use ffmpeg to encode the file or add the appropriate flag to fast start playback. In the example below we add the `faststart` flag and use the same audio and video codecs as t he original file.

```
ffmpeg -i input.mp4 -movflags faststart \
-acodec copy \
-vcodec copy \
output.mp4
```

You can do something similar with WebM videos. The format is based around the Matroska container, either VP8 or VP9 video codecs and either Opus or Vorbis audio codecs. Matroska files, usually just called MKV files, use a kind of binary XML called EBML to store different things like video tracks, audio tracks, subtitles, and other data. These data chunks are called elements and they are similar in concept to the atoms in an MP4 file.

As with all video formats to start playing a WebM video, a browser has to know where the audio and video data is stored in elements. The element we're looking for is SeekHead. By default most video creation tools put a SeekHead element at the start of the video. The problem is that each video can have an unlimited number of SeakHead. In this case, the first SeekHead will container a pointer to a second SeekHead located at the end of the file.

Even if the first SeekHead contains pointers to the video and audio tracks, the browser still must go fetch the second SeekHead element, to see if there are additional video or audio tracks in the file, and determine which one has preference. Even if the second SeekHead is completely empty the browser must download and parse all SeekHead elements in the WebM file before it can play video content.

When playing a WebM video locally we don't need to worry about the file structure since we have all the content available for playback. When streaming a video over HTTP the order of elements does matter because the browser doesn't have the complete file yet. If the browser doesn't get certain elements at the beginning of the file it has to send range HTTP requests until it finds the data it needs. This can have impact on how quickly the file starts playing and overall page performance. The discussion below is all about rearranging the elements in the container.

Another aspect of WebM streaming performance is to optimize for seeking inside a video. This is another element, Cues. For the same reasons we are optimizing for fast start we want the Cues element downloaded as early as possible so that, if a user fast forwards the video, they will get a few HTTP downloads as possible.

To accomplish both goals, fast playback and fast seeking we'll use a single tool, mkclean, a tool specifically designed to address both the fast start and the fast seek problems. Using `original.webm` we run the following command to create the resulting `optimized.web` ready for the web

```
mkclean --doctype 4 \
--keep-cues \
--optimize \
original.webm optimized.webm
```

DASH and HLS

> Although DASH is designed for both on-demand and live streaming events, I'll concentrate on on-demand content.
>
> Also important to note. Even though EME is part of DASH we will not work with EME extensions as I don't believe they should be part of the web platform.

Because our ecosystem for playing video on the web has changed considerably and now smaller devices (phone, tablets) access our content over unreliable networks subject the way we deliver video has changed. We need to account for these elements in how we deliver our video.

We'll work with two specs for streaming video for web delivery are DASH (also known as MPEG-DASH) and Apple's HSL.

Dynamic Adaptive Streaming over HTTP (DASH) is an adaptive bit-rate streaming technique delivered from conventional HTTP web servers.

MPEG-DASH breaks the content into a sequence of small HTTP-based file segments, each segment containing a short interval of playback time of content that is potentially many hours in duration, such as a movie or a live broadcase of a sports event.

Another important consideration is your audience. How many different streams will you provide for your audience? Are these videos encoded and packaged properly?

When creating DASH content you can choose what bit rates you make the content available for by providing videos encoded to those bitrates, the packager creates alternative segments encoded at the target bit rates covering the same, short, intervals of play back time.

While the content is being played back by a DASH client (in this case the web browser), the client automatically selects from the alternatives the next segment to download and play back based on current network conditions. The client selects the segment with the highest bit rate possible that can be downloaded in time for play back without causing stalls or re-buffering events in the playback. Thus, a browser playing back DASH content can seamlessly adapt to changing network conditions, and provide high quality play back with fewer stalls or re-buffering events.

DASH doesn't resolve the HTML5 codec issue. DASH is codec agnostic which means that it can be implemented in either H.264 or WebM. This means that we're back at square one in terms of what code we use and that will mean an increase in costs associated with storage and, potentially, bandwidth delivery.

MPEG-LA has a DASH Patent Pool and this has definite impact in adoption among open source purists and adopters including Mozilla, according to Chris Blizzard (at Mozilla when the quote was made):

> Mozilla has always been committed to implementing widely adopted royalty-free standards. If the underlying MPEG standards were royalty free we would implement DASH. However, MPEG DASH is currently built on top of MPEG Transport Streams, which are not royalty free. Therefore, we are unlikely to implement at this time.
>
> — What is MPEG DASH? / November 22, 2011

Taking out Firefox market share (almost 12% of the browser market) doesn't make much sense to deploy a technology that will make more work for us in the long run. On the other hand, we can look at what happened with the support of MP4 and the debacle still ongoing with what combination of container/video/audio codec to support the picture looks a little less bleak, but not by much :)

HLS (HTTP Live Streaming) is a technology developed by Apple as part of the OS X/ iOS media stack that works in a similar fashion to DASH but with different requirements, technologies and features.

The same concerns I raised about DASH apply to HLS. There is also the issue of this being a single vendor specification; there an IETF Informational Internet Draft there hasn't been any action to ratify the draft as an IETF standard.

For the rest of this post we'll concentrate on DASH as it's the one that has the widest level of support and it means that I don't have to create the packager or the player later on.

DASH Process

As I understand it the process to create content ready to play in a DASH-enabled web browser is as follows:

1. Encode the video to the target bit rates you want to use
2. Create the DASH manifest using the Shaka Packager
3. Upload the content to your server
4. Create the video tag using the Shaka Player or Dash.js

The process assumes that you've already encoded the videos to your target bit rate(s).

Packaging the video

Figure 3: Shaka Packager, what we'll use to create the DASH Manifest

Shaka Packager is a tool developed by Google to create DASH manifest for our content. It will also create separate streams for audio and video.

In the example we'll work with in these sections we'll generate a manifest three diferent versions of the same video. ***Shaka Packager will not encode the video... that's your job and it should be done before we start working in packaging and playing DASH content***.

```
path/to/packager \
input=media/SavingLight.mp4,stream=audio,output=audio.mp4 \
input=media/SavingLight.mp4,stream=video,output=video.mp4 \
input=media/SavingLight-baseline.mp4,stream=audio,output=audio-baseline.mp
input=media/SavingLight-baseline.mp4,stream=video,output=video-baseline.mp
input=media/SavingLight-high.mp4,stream=audio,output=audio-high.mp4 \
input=media/SavingLight-high.mp4,stream=video,output=video-high.mp4 \
--mpd_output example.mpd
```

If you're comfortable compiling tools manually you can clone the Shaka Packager Github Repository and compile the tools following the instructions in the README file.

The DASH Manifest

One of the files produced by the Packager is the DASH manifest file. It is an XML file that describes the audio and video tracks that make up the video separately from one another.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Generated with https://github.com/google/shaka-packager
  version 593f513c83-release-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xlink="http://www.w3<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xsi:schemaLocation="urn:mpeg:dash:schema:mpd:2011 DASH-MPD.xsd"
     xmlns:cenc="urn:mpeg:cenc:2013"
     profiles="urn:mpeg:dash:profile:isoff-on-demand:2011"
     minBufferTime="PT2S"
     type="static"
     mediaPresentationDuration="PT355.614S">54">
      <Representation id="0" bandwidth="1778582"
                      codecs="avc1.640028"
                      mimeType="video/mp4"
```

```
                    sar="1:1"
                    height="1080">
      <BaseURL>video.mp4</BaseURL>
      <SegmentBase indexRange="816-1231" timescale="90000">
        <Initialization range="0-815"/>
      </SegmentBase>
    </Representation>
    <Representation id="1" bandwidth="2261351"
                    codecs="avc1.640028"
                    mimeType="video/mp4"
                    sar="1:1"
                    height="800">
      <BaseURL>video-high.mp4</BaseURL>
      <SegmentBase indexRange="817-1208" timescale="90000">
        <Initialization range="0-816"/>
      </SegmentBase>
    </Representation>
    <Representation id="2" bandwidth="2606321"
                    codecs="avc1.42c028"
                    mimeType="video/mp4"
                    sar="1:1"
                    height="800">
      <BaseURL>video-baseline.mp4</BaseURL>
      <SegmentBase indexRange="815-1218" timescale="90000">
        <Initialization range="0-814"/>
      </SegmentBase>
    </Representation>
  </AdaptationSet>
  <AdaptationSet id="1" contentType="audio" subsegmentAlignment="true">
    <Representation id="3" bandwidth="127078"
                    codecs="mp4a.40.2"
                    mimeType="audio/mp4"
                    audioSamplingRate="44100">
      <AudioChannelConfiguration
      schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:201
      value="2"/>
      <BaseURL>audio.mp4</BaseURL>
      <SegmentBase indexRange="745-1208" timescale="44100">
```

```
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
      <Representation id="4" bandwidth="156471"
                      codecs="mp4a.40.2"
                      mimeType="audio/mp4"
                      audioSamplingRate="44100">
        <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:201
        value="2"/>
        <BaseURL>audio-high.mp4</BaseURL>
        <SegmentBase indexRange="745-1208" timescale="44100">
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
      <Representation id="5" bandwidth="156471"
                      codecs="mp4a.40.2"
                      mimeType="audio/mp4"
                      audioSamplingRate="44100">
        <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:201
        value="2"/>
        <BaseURL>audio-baseline.mp4</BaseURL>
        <SegmentBase indexRange="745-1208" timescale="44100">
          <Initialization range="0-744"/>
        </SegmentBase>
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

Captions

Subtitles and captions are also part of the packaging process. The example belows takes the movie Sintel and splits it into audio and video streams and adds an english caption track.

```
packager \
  input=sintel.mp4,stream=audio,output=sintel_audio.mp4 \
  input=sintel.mp4,stream=video,output=sintel_video.mp4 \
  input=sintel_english_input.vtt,stream=text,output=sintel_english.vtt \
--mpd_output sintel_vod.mpd
```

Playing content: Shaka Player

Figure 4: Shaka Player, the player we'll use for this project

Shaka Player is the playback component of the Shaka ecosystem. Also developed by Google and open sourced on Github.

If you're used to HTML5 the way you add DASH video is a little more complicated than you're used to. The process is:

First we create a simple HTML page with a video element. In this page we make sure that we add thescripts we need:

- The shaka-player script
- The script for our application

The `video` element is incomplete on purpose. We will add the rest of the video in the script later on.

```
<!DOCTYPE html>
<html>
  <head><html><!-- Shaka Player compiled library: -->    <script src="path
    <<script src="path/to/shaka-player.compiled.js"><!-- Your application
    <video id="vide<body></script>
  </head>
```

```
  <body>
    <video id="video"
           width="640"
           poster="media/SavingLight.jpg"
           controls autoplay></video>
  </body>
</html>
```

We'll break the script into three parts:

- Application init
- Player init
- Error handler and event listener

We initialize the application by installing the polyfills built into the Shaka player to make sure that all the supported players behave the same way and that there won't be any unexpected surprises later on.

The next step is to check if the browser is supported using the built in `isBrowserSupported` check. If the browsers supports DASH then we initialize the player by calling `initPlayer()` otherwise we log the error to console.

```
var manifestUri = 'media/example.mpd';

function initApp() {
  // Install built-in polyfills to patch browser incompatibilities.
  shaka.polyfill.installAll();

  // Check to see if the browser supports the basic APIs Shaka needs.
  if (shaka.Player.isBrowserSupported()) {
    // Everything looks good!
    initPlayer();
  } else {
    // This browser does not have the minimum set of APIs we need.
    console.error('Browser not supported!');
  }
}
'Browser not supported!'
```

Initializing the player is the meat of the process and will take serval different steps.

We create variables to capture the video element using `getElementById` and the player by assigning a new instance of `Shaka.Player` and attach it to the video element.

We then attach the player to the window object to make it easier to access the console.

Next we attach the error event handler to the `onErrorEvent` function defined later in the script. Positioning doesn't matter as far as Javascript is concerned.

The last step in this function is to try and load a manifest using a promise. If the promise succeeds then we log it to console otherwise the `catch` tree of the promise chain is executed and runs the `onError` function (which is different than `onErrorEvent` discussed earlier).

```
function initPlayer() {
  // Create a Player instance.
  var video = document.getElementById('video');
  var player = new shaka.Player(video);

  'video'// Attach player to the window to make it easy to access in the
  window.player = player;

  // Listen for error events.er.addEventListener('error', onErrorEvent);

  // Try 'error'// Try to load a manifest.
  // This is an asynchronous process.
  player.load(manifestUri).then(function() {
    // This runs if the asynchronous load is successful.og('The video has
  }).catch(onError);  // onError is 'The video has now been loaded!'// onl
}
```

The last part of the script is to create the functions for errors (`onErrorEvent` and `onError`)

Finally we attach the `initApp()` function to the `DOMContentLoaded` event.

```
function onErrorEvent(event) {
  // Extract the shaka.util.Error object from the event.
  onError(event.detail);
}

function onError(error) {
  // Log the error.
  console.error('Error code', error.code, 'object', error);
}

document.addEventListener('DOMContentLoaded', initApp);
'Error code'
```

If everything works out OK we should have a video playing on screen.

There is a full example available to show how the player works. We've covered only the player's basic functionality; there's additional capabilities like casting to an Android Play device and playing your content on your TV... I'm more concerned with getting the video working.

Playing content: Dash.js

Figure 5: Dash.js is the reference implementation for MPEG-DASH

Dash.js is the reference DASH implementation, meaning this is the technology that they use to validate and demonstrate the different part of the specification and what they offer developers and implementers to use as the basis of their own player software.

The first way to use Dash.js is to manually initialize the player and attach it to a video element already in the page. It is possible to also create the video element programmatically and then assign it to the player.

The standard setup method uses javascript to initialize and provide video details to dash.js. MediaPlayerFactory provides an alternative declarative setup

syntax.

Standard Setup

Using the same files that we used to create the Shaka demo we create the Dash.js video using code like the one below. In this page the script initializes the player and attaches it to the element with the id of video2 (#video2)

```
<!DOCTYPE html>
<html lang="en"<html lang="en">ta charset="UTF-8">
    <title>Title</title>
    <style><title>
      video {
        width: 640px;
        height: 360px;
      }
    >
</head>
<body>
  <div>
    <video id="video2" poster="media/SavingLight.jpg" controls></video>
  </div>

  <script src="http://cdn.dashjs.org/latest/dash.all.min.js"></script>
  <scr</head>  (function(){
      var url = "media/example.mpd";
      var player = dashjs.MediaPlayer().create();
      player.initialize(document.querySelector("#video2"), url, true);
    })();
  </script>

</body>
</html>
</script>
```

MediaPlayerFactory

An alternative way to build a Dash.js player in your web page is to use the

`MediaPlayerFactory`. The MediaPlayerFactory will automatically instantiate and initialize the MediaPlayer module on appropriately tagged video elements.

Create a `video` element somewhere in your html and provide the path to your mpd file as src. Also ensure that your video element has the `data-dashjs-player` attribute on it. An example using the `MediaPlayerFactory` looks like this:

```
<!DOCTYPE html>
<html lang="en"<html lang="en">ta charset="UTF-8">
    <title>Title</title>
    <script src="http://cd<title>s.org/latest/dash.all.min.js"></script>
    <</script>style>
      video {
        width: 640px;
        height: 360px;
      }
    </style>
</head>
<body>
  <div>
    <video data-dashjs-player src="media/example.mpd" controls></video>
  </div>

</body>
</html>
```

Conclusion

Dash works but it requires a lot of work upfront to make the technology work as intended for the use cases use the technology for. The demos cover some of the most basic use cases for video on demand; we have not considered live streams or encrypted video.

As with many things on the web there is no ‘one size fits all’ solution. DASH works and it provides awesome capabilties but with those capabilities come additional cost for storage and delivery. In the example I used for this project used three streams each for audio and video and the weigh between 79 and 115MB for the video stream and betwee 5 and 7MB per audio stream. The more bitrates you

add the more you have to consider storage costs.

Video is an awesome tool but one that requires a lot of prep work up front for it to be an effective tool.

Notes about the demo repository

All the code examples in this post are available in the dash-demo Github repository. To store the mp4 content, some of which is over 100mb in size, we've set the repo with GIT LFS to handle the large files; this will get around Github's file size limitation.

A brief summary of the files:

- html5-video.html is a traditional HTML5 video tag using MP4, WebM and OGG video
- index.html uses the Shaka Player to play DASH video
- dashjs.html uses Dash.js's MediaPlayerFactory method
- dashjs2.html uses Dash.js's traditional method