Fast Playback with Video Preload

By François Beaufort
Dives into Chromium source code

Faster playback start means more people watching your video. That's a known fact. In this article I'll explore techniques you can use to accelerate your media playback by actively preloading resources depending on your use case.

Note: Unless specified otherwise, this article also applies to the audio element.

Credits: copyright Blender Foundation | www.blender.org .

TL;DR

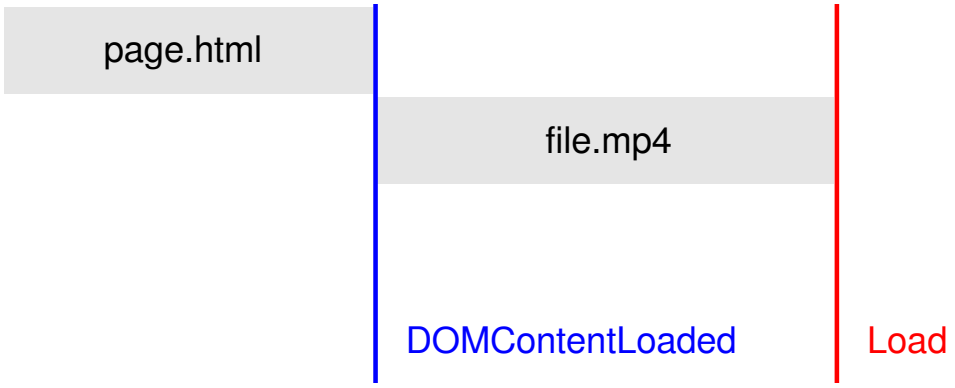
	It's great...	But...
Video preload attribute	Simple to use for a unique file hosted on a web server.	Browsers may completely ignore the attribute.
		Resource fetching starts when the HTML document has been completely loaded and parsed.

		MSE ignores the preload attribute on media elements because the app is responsible for providing media to MSE.
Link preload	Forces the browser to make a request for a video resource without blocking the document's **onload** event.	HTTP Range requests are not compatible.
	Compatible with MSE and file segments.	Should be used only for small media files (<5 MB) when fetching full resources.
Manual buffering	Full control	Complex error handling is the website's responsibility.

Video preload attribute

If the video source is a unique file hosted on a web server, you may want to use the video **preload** attribute to provide a hint to the browser as to how much information or content to preload. This means Media Source Extensions (MSE) is not compatible with **preload**.

Resource fetching will start only when the initial HTML document has been completely loaded and parsed (e.g. the **DOMContentLoaded** event has fired) while the very different **window.onload** event will be fired when resource has actually been fetched.



Setting the **preload** attribute to **metadata** indicates that the user is not expected to need the video, but that fetching its metadata (dimensions, track list, duration, and so on) is desirable. Note that starting in Chrome 64, the default value for **preload** is **metadata**. (It was **auto** previously).

```
<video id="video" preload="metadata" src="file.mp4" controls></video>

<script>
  video.addEventListener('loadedmetadata', function() {
    if (video.buffered.length === 0) return;
```

```
    var bufferedSeconds = video.buffered.end(0) - video.buffered.start(0);
    console.log(bufferedSeconds + ' seconds of video are ready to play!');
  });
</script>
```

Setting the **preload** attribute to **auto** indicates that the browser may cache enough data that complete playback is possible without requiring a stop for further buffering.

```
<video id="video" preload="auto" src="file.mp4" controls></video>

<script>
  video.addEventListener('loadedmetadata', function() {
    if (video.buffered.length === 0) return;

    var bufferedSeconds = video.buffered.end(0) - video.buffered.start(0);
    console.log(bufferedSeconds + ' seconds of video are ready to play!');
  });
</script>
```

There are some caveats though. As this is just a hint, the browser may completely ignore the **preload** attribute. At the time of writing, here are some rules applied in Chrome:

- When Data Saver is enabled, Chrome forces the **preload** value to **none**.
- In Android 4.3, Chrome forces the **preload** value to **none** due to an Android bug.
- On a cellular connection (2G, 3G, and 4G), Chrome forces the **preload** value to **metadata**.

Tips

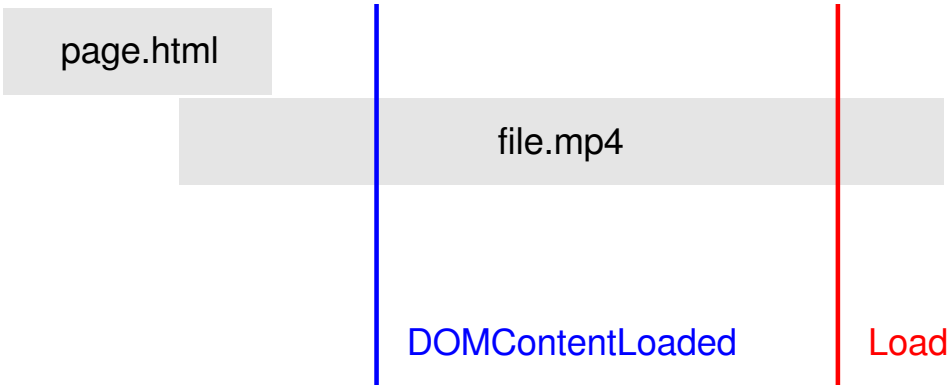
If your website contains many video resources on the same domain, I would recommend you set the **preload** value to **metadata** or define the **poster** attribute and set **preload** to **none**. That way, you would avoid hitting the maximum number of HTTP connections to the same domain (6 according to the HTTP 1.1 spec) which can hang loading of resources. Note that this may also improve page speed if videos aren't part of your core user experience.

Link preload

As covered in other articles, link preload is a declarative fetch that allows you to force the browser to make a request for a resource without blocking the **window.onload** event and while the page is downloading. Resources loaded via **<link rel="preload">** are stored

locally in the browser, and are effectively inert until they're explicitly referenced in the DOM, JavaScript, or CSS.

Preload is different from prefetch in that it focuses on current navigation and fetches resources with priority based on their type (script, style, font, video, audio, etc.). It should be used to warm up the browser cache for current sessions.



Preload full video

Here's how to preload a full video on your website so that when your JavaScript asks to fetch video content, it is read from cache as the resource may have already been cached by the browser. If the preload request hasn't finished yet, a regular network fetch will happen.

```
<link rel="preload" as="video" href="https://cdn.com/small-file.mp4">

<video id="video" controls></video>

<script>
  // Later on, after some condition has been met, set video source to the
  // preloaded video URL.
  video.src = 'https://cdn.com/small-file.mp4';
  video.play().then(_ => {
    // If preloaded video URL was already cached, playback started immediately.
  });
</script>
```

Note: I would recommend using this only for small media files (< 5MB).

Because the preloaded resource is going to be consumed by a video element in the example, the `as` preload link value is `video`. If it were an audio element, it would be `as="audio"`.

Preload the first segment

The example below shows how to preload the first segment of a video with `<link rel="preload">` and use it with Media Source Extensions. If you're not familiar with the MSE Javascript API, please read MSE basics.

For the sake of simplicity, let's assume the entire video has been split into smaller files like "file_1.webm", "file_2.webm", "file_3.webm", etc.

```
<link rel="preload" as="fetch" href="https://cdn.com/file_1.webm">

<video id="video" controls></video>

<script>
  const mediaSource = new MediaSource();
  video.src = URL.createObjectURL(mediaSource);
  mediaSource.addEventListener('sourceopen', sourceOpen, { once: true });

  function sourceOpen() {
    URL.revokeObjectURL(video.src);
    const sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vp09.00

    // If video is preloaded already, fetch will return immediately a response
    // from the browser cache (memory cache). Otherwise, it will perform a
    // regular network fetch.
    fetch('https://cdn.com/file_1.webm')
    .then(response => response.arrayBuffer())
    .then(data => {
      // Append the data into the new sourceBuffer.
      sourceBuffer.appendBuffer(data);
      // TODO: Fetch file_2.webm when user starts playing video.
    })
    .catch(error => {
      // TODO: Show "Video is not available" message to user.
    });
  }
</script>
```

Warning: For cross-origin resources, make sure your CORS headers are set properly. As we can't create an array buffer from an opaque response retrieved with `fetch(videoFileUrl, { mode: 'no-cors' })`, we won't be able to feed any video or audio element.

Support

Link preload is not supported in every browser yet. You may want to detect its availability with the snippets below to adjust your performance metrics.

```
function preloadFullVideoSupported() {
  const link = document.createElement('link');
  link.as = 'video';
  return (link.as === 'video');
}

function preloadFirstSegmentSupported() {
  const link = document.createElement('link');
  link.as = 'fetch';
  return (link.as === 'fetch');
}
```

Manual buffering

Before we dive into the Cache API and service workers, let's see how to manually buffer a video with MSE. The example below assumes that your web server supports HTTP Range requests but this would be pretty similar with file segments. Note that some middleware libraries such as Google's Shaka Player, JW Player, and Video.js are built to handle this for you.

```
<video id="video" controls></video>

<script>
  const mediaSource = new MediaSource();
  video.src = URL.createObjectURL(mediaSource);
  mediaSource.addEventListener('sourceopen', sourceOpen, { once: true });

  function sourceOpen() {
    URL.revokeObjectURL(video.src);
    const sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vp09.00

    // Fetch beginning of the video by setting the Range HTTP request header.
    fetch('file.webm', { headers: { range: 'bytes=0-567139' } })
    .then(response => response.arrayBuffer())
    .then(data => {
      sourceBuffer.appendBuffer(data);
      sourceBuffer.addEventListener('updateend', updateEnd, { once: true });
    });
  }

  function updateEnd() {
    // Video is now ready to play!
    var bufferedSeconds = video.buffered.end(0) - video.buffered.start(0);
    console.log(bufferedSeconds + ' seconds of video are ready to play!');
```

```
    // Fetch the next segment of video when user starts playing the video.
    video.addEventListener('playing', fetchNextSegment, { once: true });
  }

  function fetchNextSegment() {
    fetch('file.webm', { headers: { range: 'bytes=567140-1196488' } })
    .then(response => response.arrayBuffer())
    .then(data => {
      const sourceBuffer = mediaSource.sourceBuffers[0];
      sourceBuffer.appendBuffer(data);
      // TODO: Fetch further segment and append it.
    });
  }
</script>
```

Considerations

As you're now in control of the entire media buffering experience, I suggest you consider the device's battery level, the "Data-Saver Mode" user preference and network information when thinking about preloading.

Battery awareness

Please take into account the battery level of users' devices before thinking about preloading a video. This will preserve battery life when the power level is low.

Disable preload or at least preload a lower resolution video when the device is running out of battery.

```
if ('getBattery' in navigator) {
  navigator.getBattery()
  .then(battery => {
    // If battery is charging or battery level is high enough
    if (battery.charging || battery.level > 0.15) {
      // TODO: Preload the first segment of a video.
    }
  });
}
```

Detect "Data-Saver"

Use the `Save-Data` client hint request header to deliver fast and light applications to users who have opted-in to "data savings" mode in their browser. By identifying this request

header, your application can customize and deliver an optimized user experience to cost- and performance-constrained users.

Learn more by reading our complete Delivering Fast and Light Applications with Save-Data article.

Smart loading based on network information

You may want to check `navigator.connection.type` prior to preloading. When it's set to `cellular`, you could prevent preloading and advise users that their mobile network operator might be charging for the bandwidth, and only start automatic playback of previously cached content.

```
if ('connection' in navigator) {
  if (navigator.connection.type == 'cellular') {
    // TODO: Prompt user before preloading video
  } else {
    // TODO: Preload the first segment of a video.
  }
}
```

Checkout the Network Information sample to learn how to react to network changes as well.

Pre-cache multiple first segments

Now what if I want to speculatively pre-load some media content without knowing which piece of media the user will eventually pick. If the user is on a web page that contains 10 videos, we probably have enough memory to fetch one segment file from each but we should definitely not create 10 hidden video elements and 10 `MediaSource` objects and start feeding that data.

The two part example below shows you how to pre-cache multiple first segments of video using the powerful and easy-to-use Cache API. Note that something similar can be achieved with IndexedDB as well. We're not using service workers yet as the Cache API is also accessible from the Window object.

Fetch and cache

```
const videoFileUrls = [
  'bat_video_file_1.webm',
  'cow_video_file_1.webm',
  'dog_video_file_1.webm',
```

```
  'fox_video_file_1.webm',
];

// Let's create a video pre-cache and store all first segments of videos inside.
window.caches.open('video-pre-cache')
.then(cache => Promise.all(videoFileUrls.map(videoFileUrl => fetchAndCache(videoF

function fetchAndCache(videoFileUrl, cache) {
  // Check first if video is in the cache.
  return cache.match(videoFileUrl)
  .then(cacheResponse => {
    // Let's return cached response if video is already in the cache.
    if (cacheResponse) {
      return cacheResponse;
    }
    // Otherwise, fetch the video from the network.
    return fetch(videoFileUrl)
    .then(networkResponse => {
      // Add the response to the cache and return network response in parallel.
      cache.put(videoFileUrl, networkResponse.clone());
      return networkResponse;
    });
  });
}
```

Note that if I were to use HTTP Range requests, I would have to manually recreate a **Response** object as the Cache API doesn't support Range responses yet. Be mindful that calling **`networkResponse.arrayBuffer()`** fetches the whole content of the response at once into renderer memory, which is why you may want to use small ranges.

For reference, I've modified part of the example above to save HTTP Range requests to the video pre-cache.

```
    ...
    return fetch(videoFileUrl, { headers: { range: 'bytes=0-567139' } })
    .then(networkResponse => networkResponse.arrayBuffer())
    .then(data => {
      const response = new Response(data);
      // Add the response to the cache and return network response in parallel.
      cache.put(videoFileUrl, response.clone());
      return response;
    });
```

Play video

When a user clicks a play button, we'll fetch the first segment of video available in the Cache API so that playback starts immediately if available. Otherwise, we'll simply fetch it from the network. Keep in mind that browsers and users may decide to clear the Cache.

As seen before, we use MSE to feed that first segment of video to the video element.

```
function onPlayButtonClick(videoFileUrl) {
  video.load(); // Used to be able to play video later.

  window.caches.open('video-pre-cache')
  .then(cache => fetchAndCache(videoFileUrl, cache)) // Defined above.
  .then(response => response.arrayBuffer())
  .then(data => {
    const mediaSource = new MediaSource();
    video.src = URL.createObjectURL(mediaSource);
    mediaSource.addEventListener('sourceopen', sourceOpen, { once: true });

    function sourceOpen() {
      URL.revokeObjectURL(video.src);

      const sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vp09.
      sourceBuffer.appendBuffer(data);

      video.play().then(_ => {
        // TODO: Fetch the rest of the video when user starts playing video.
      });
    }
  });
}
```

Warning: For cross-origin resources, make sure your CORS headers are set properly. As we can't create an array buffer from an opaque response retrieved with `fetch(videoFileUrl, { mode: 'no-cors' })`, we won't be able to feed any video or audio element.

Create Range responses with a service worker

Now what if you have fetched an entire video file and saved it in the Cache API. When the browser sends an HTTP Range request, you certainly don't want to bring the entire video into renderer memory as the Cache API doesn't support Range responses yet.

So let me show how to intercept these requests and return a customized Range response from a service worker.

```
addEventListener('fetch', event => {
  event.respondWith(loadFromCacheOrFetch(event.request));
});

function loadFromCacheOrFetch(request) {
  // Search through all available caches for this request.
  return caches.match(request)
  .then(response => {

    // Fetch from network if it's not already in the cache.
    if (!response) {
      return fetch(request);
      // Note that we may want to add the response to the cache and return
      // network response in parallel as well.
    }

    // Browser sends a HTTP Range request. Let's provide one reconstructed
    // manually from the cache.
    if (request.headers.has('range')) {
      return response.blob()
      .then(data => {

        // Get start position from Range request header.
        const pos = Number(/^bytes\=(\d+)\-/g.exec(request.headers.get('range'))[
        const options = {
          status: 206,
          statusText: 'Partial Content',
          headers: response.headers
        }
        const slicedResponse = new Response(data.slice(pos), options);
        slicedResponse.setHeaders('Content-Range': 'bytes ' + pos + '-' +
            (data.size - 1) + '/' + data.size);
        slicedResponse.setHeaders('X-From-Cache': 'true');

        return slicedResponse;
      });
    }

    return response;
  }
}
```

It is important to note that I used **`response.blob()`** to recreate this sliced response as this simply gives me a handle to the file (in Chrome) while **`response.arrayBuffer()`** brings the entire file into renderer memory.

My custom `X-From-Cache` HTTP header can be used to know whether this request came from the cache or from the network. It can be used by a player such as ShakaPlayer to ignore the response time as an indicator of network speed.

Have a look at the official Sample Media App and in particular its ranged-response.js file for a complete solution for how to handle Range requests.

Last updated July 2, 2018.