Public

Native lazy-loading in CMS platforms

Considerations, recommendations, and learnings

Status: Approved
Authors: felixarntz@, igrigorik@, scottfriesen@
Last Updated: 2020-07-14

This document outlines several takeaways from implementing support for native lazy-loading in the WordPress platform core, potentially helpful for adoption in other CMSs.

Standardization & adoption

Attribute specification is a standard

The `loading` attribute was incubated in the WICG and is now part of the WHATWG HTML Standard (see lazy loading attributes in general and the `img loading` attribute). More recently, support for the attribute was also added to iframes (see the `iframe loading` attribute). While this document currently focuses primarily on adding the attribute to images, the majority of learnings from there apply to providing the attribute on iframes as well.

Attribute has cross-browser support

The `loading` attribute on `img` elements is supported by most popular Chromium-powered browsers (Chrome, Edge, Opera), Firefox, and the implementation for WebKit / Safari is in progress. Detailed information on cross-browser support can be found on caniuse.com. It should be noted that even browsers that do not support the attribute simply ignore the attribute with no adverse effects.

User experience & performance

Conservative thresholds for elements scrolling into view

With lazy-loading mechanisms it is important that they use well-tuned heuristics so that images that are not within the initial viewport (“below the fold”) are still loaded early enough so that the chance of them not having finished loading by the time the user scrolls to them is as minimal as it would be without lazy-loading. The browser-native implementation of the `loading` attribute automatically prefetches nearby resources before they become visible in the viewport, which is

tuned based on environmental signals of the user. This helps maximize the likelihood that the resource is already loaded by the time it becomes visible.

Based on Chrome's implementation, the difference compared to images which are not lazy-loaded is minor: Experiments using Chrome on Android indicated that on 4G networks, 97.5% of below-the-fold lazy-loaded images were fully loaded within 10ms of becoming visible, compared to 97.6% for non lazy-loaded images. And even on a slow 2G network, still 92.6% of below-the-fold images were fully loaded within 10ms. As such, native lazy-loading provides a stable experience regarding visibility of elements scrolling into view.

Considerations when used on elements in first visible viewport

At the moment it is recommended to only add `loading="lazy"` attributes to images which are positioned below the fold, if possible. Images that are marked as candidates for lazy-loading require the browser to resolve where the image is positioned on the page, which relies on the IntersectionObserver to be available and thus delays their fetching. The impact of images in the initial viewport being marked with `loading="lazy"` on the Largest Contentful Paint metric is fairly small, with a regression of <1% at the 75th and 99th percentiles compared to eagerly loaded images. Furthermore the implementation may be improved in the future. Yet, if you are able to know which images will be above the fold, it is currently recommended to omit the `loading` attribute on those.

Automatically detecting in the content authoring phase whether an image is likely to be in the viewport or not is complex though, and even manually it is hard to reliably assess due to different viewport formats. It is recommended to make this decision depending on the audience of the CMS as well as the effort needed to implement a detection mechanism.

Elements should include dimension attributes

While an image file is being loaded by the browser, it does not immediately know the image's dimension, unless specified otherwise. In order for the browser to reserve the necessary space on a page, it is strongly recommended that all `img` tags include `width` and `height` attributes. If those are not present, layout shifting will occur which becomes especially apparent to a user if an image takes a while to load. This guidance applies to `img` tags regardless of whether or not they are being loaded lazily, but with lazy-loading it is even more important to reduce risk of additional layout shifts. For CMSs that provide dimension attributes `width` and `height` on their images, this is not a concern.

If a CMS is unable to provide dimension attributes for its images, lazy-loading them is a trade-off decision between saving network resources and assessing a *slightly* higher risk of layout shifts: While native lazy-loading is implemented in a way that images are likely to be loaded once they become visible, it should be acknowledged that there is a minimally greater chance of them not being loaded yet, in which case missing `width` and `height` attributes on

such images increase their impact on cumulative layout shifting. Since the likelihood of an image not having finished loading once visible is low and since this problem is technically not related to lazy-loading, this should be a secondary concern here.

Graceful degradation

In browsers that do not (yet) support the `loading` attribute, its presence will be ignored. While these browsers will not get the benefits of lazy-loading, there is no negative impact whatsoever for such browsers.

Technical implementation

Using “lazy” attribute value by default on content images

In order to benefit the web and its users, the **implementation should add the `loading` attribute as “lazy” by default on all content images**. While there are certain minor UX trade-offs, the majority of those images should not affect the Largest Contentful Paint metric. CMSs should provide mechanisms to opt out of lazy-loading for certain images though, at least for developers so that the default behavior can be fine tuned. Whether to introduce a UI or an API to allow for the `loading` attribute value to be changed largely depends on the audience of the CMS.

For WordPress specifically, it was decided to make the behavior customizable programmatically, but not through a user interface. Furthermore, in WordPress all images, including template images e.g. in theme header or footer, are lazy-loaded by default, acknowledging the trade-off with the current Largest Contentful Paint impact.

Regardless of the detailed approach taken, on a high level lazy-loading should be implemented with an opt-out mechanism, rather than an opt-in mechanism, since it is a technical term and many CMS users would not know what to make of it; if they had to enable lazy-loading in general or per image, the potential performance gains at a large scale would likely be lost. If a certain set of images are likely to be in the initial viewport, it is recommended to not load those lazily. For such images, the `loading` attribute should be omitted altogether, rather than setting it as “eager”.

Avoiding JavaScript-based fallback

While adding a JavaScript-based fallback to the native `loading` attribute is possible, it may negatively impact all the browsers that do support the attribute. Furthermore, with traditional JavaScript approaches being more error-prone as explained above, rolling out lazy-loading at a large scale would not go as smoothly. Another consideration is that any JavaScript solution would not be standardized, leading to further discussions on which one to pick.

Therefore it is recommended to not provide a fallback and have browsers without support simply ignore the attribute. It is worth adding that, even if there was no negative impact on browsers that support the attribute, the behavior and complexity of a JavaScript solution would be a much greater surface area for potential issues if enabled by default - which is likely one of the main reasons lazy-loading has not found its way into any major CMS's core before now.

Retrofitting existing content

For maximum performance impact it is recommended that `loading` attributes are retroactively added to images in existing content as well, rather than only for new content. The attribute's existence should preferably be decoupled from database storage and be handled on the code level. Unless it is truly desirable to provide a UI for controlling the `loading` attribute on a *per-image* basis, the recommendation is to refrain from integrating the feature with the CMSs editor. Using the attribute in reusable templates or dynamically modifying `img` tags in user-generated content are two recommended alternatives.

Optimizing server-side addition of the attribute

Based on the above recommendation of retrofitting existing content, it may be necessary to run additional server-side logic on pageload. For template-based CMSs with e.g. a template for `img` tags this should not be much of a consideration if the template can be modified directly. However, CMSs that work with a string of arbitrary HTML markup that comes directly from the database storage should optimize the logic to inject the `loading` attribute for performance. To take WordPress as an example, it was already running a regular expression on the user-generated content to dynamically inject responsive image attributes, so expanding and slightly modifying this existing regular expression resulted in better server-side performance than adding another one (specifically, `wp_make_content_images_responsive` was replaced with a more generic `wp_filter_content_tags`).

Another consideration for dynamically injecting the attribute into user-generated content is that the logic should preferably account for the case where a user or a plugin already added a loading attribute to an image, to avoid duplicate attributes.

Adding the attribute where reasonably possible

While the goal is to lazy-load images throughout, there may be cases where adding the attribute would result in other problems or simply be too cumbersome. The recommendation is to add the `loading` attribute to in-content `img` tags that the CMS typically has control over, but it should not need to go as far as e.g. running the entire page markup through an output buffer to modify all `img` tags. If a user or developer manually hard-coded an `img` tag in a way that is not recommended anyway, it is perfectly fine to not cover such an edge-case.