Things You Must Know Before Using a CDN

Ashan Fernando Follow
Oct 13, 2020 · 6 min read

Image by Sumanley xulx from Pixabay

Over the past few years, web applications have crossed many notable hurdles. One such achievement is the speed of loading these applications. Today, the bar is so high that users expect any web application to load within a few seconds. This situation points to the fact that ensuring these speeds requires the right tools and practices to be in place.

In this context, one such practice is to use a Content Delivery Network (CDN) for the web application frontend. The need for a CDN is prominent for web applications that have users in multiple geographical locations,

> The challenge CDN solves, is the limit with Speed of Light.

In this article, I will be discussing the need for using a CDN, and different strategies you can use to set it up in your architecture, and the common pitfalls you should avoid.

The Need for CDN

In your professional life, you might have experienced different latencies when viewing web applications from different parts of the world. When the webserver distance is far away from you, it's likely to take more time to load the web application to your browser.

What is the reason for this delay? You are very much aware that when it comes to the frontend part of a web application, it requires loading the HTML, JavaScript, CSS, and Images to the browser at first for it to function. This shows the need for accelerating the speed of loading such assets.

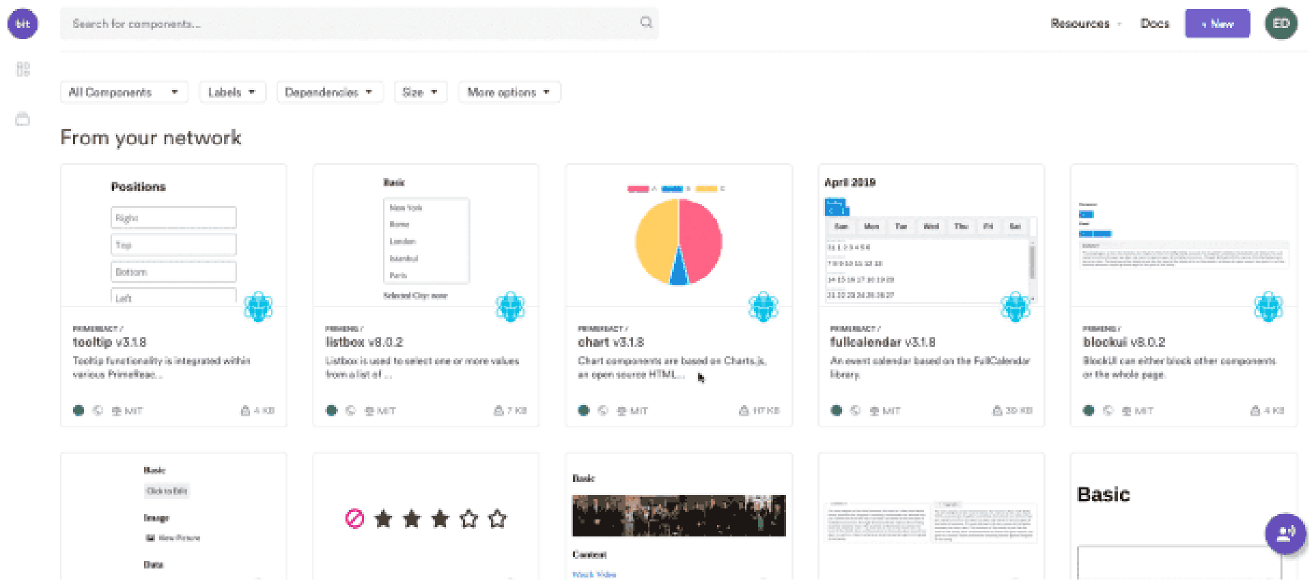
The only way to improve the speed of loading these assets is to bring the frontend content closer to the user. However, moving the server is not always possible. This is where the use of CDN comes in to play. We can use a CDN to cache the frontend content closer to the user so that the latency of loading the frontend becomes less.

. . .

Tip: **Share your reusable components** between projects using **Bit** (Github). Bit makes it simple to share, document, and organize independent components from any project**.**

Use it to maximize code reuse, collaborate on independent components, and build apps that scale.

Bit supports Node, TypeScript, React, Vue, Angular, and more.



PRIMEREACT /
tooltip v3.1.8
Tooltip functionality is integrated within various PrimeReac...
MIT 4 KB

PRIMENG /
listbox v8.0.2
Listbox is used to select one or more values from a list of ...
MIT 7 KB

PRIMEREACT /
chart v3.1.8
Chart components are based on Charts.js, an open source HTML...
MIT 117 KB

PRIMEREACT /
fullcalendar v3.1.8
An event calendar based on the FullCalendar library.
MIT 39 KB

PRIMENG /
blockui v8.0.2
BlockUI can either block other components or the whole page.
MIT 4 KB

Example: exploring reusable React components shared on Bit.dev

Different Strategies to Set Up CDN

If you decide to use a CDN for your web application, you have got to plan the strategy of using it.

One of the common approaches is to store and serve static content such as JavaScript, CSS, and Images using the CDN. In this approach, we might host the web application in a server where it delivers the index.html page. This page will contain the embed CDN links for these static assets.

Public CDN for JavaScript

You might have already witnessed that some JavaScript libraries at present provide the frontend CDN URLs that you can use to retrieve these files.

```
<script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
crossorigin="anonymous"></script>
```

As you can see here, it is the CDN hosted version of jQuery that you can directly embed into your web applications. The advantage here is that when many applications start to use this URL, you can find the library in your browser cache readily available for any application that uses the same URL. Because of this, it further improves the loading speed of your web applications.

Nonetheless, today this practice has changed mostly due to the number of JavaScript dependencies each application frontends uses. Since we use many JavaScript dependencies, it’s better to bundle them rather than loading them independently to reduce the number of requests needed to load the web application.

Need for the Private CDN

For this to work, we have to retrieve the bundle of static assets such as JavaScript from a private CDN. Here, we can further consider retrieving the CSS and Images from the CDN as well for optimum performance.

> *Note that in this case, the index.html is directly served from the webserver. Only the embed links (JavaScript links, CSS links) for static assets are served via the CDN as shown below.*

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>My Web App</title>

<link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="<Private CDN>/stylesheet.css">
  <link
    href="https://fonts.googleapis.com/css?
family=Material+Icons|Material+Icons+Sharp"
    rel="stylesheet">
</head>

<body>
    <!-- HTML Content -->
    <script src="<Private CDN>/runtime-es2015.b76e2.js"></script>
    <script src="<Private CDN>/vendor-es5.bafcac.js"></script>
  </body>

</html>
```

For this context, you can use CDN services like Akamai, Azure CDN, and AWS CloudFront. I hope you also note that we can use a mix of private and public CDN URLs in the index.html.

Use CDN as a Proxy and Entry Point to the Web Application

Another strategy you can employ is to serve the entire web application via the CDN service. Here the CDN also acts as a proxy and the entry point to the web application. For this to work, the CDN should be able to caches the frontend static content (HTML, CSS, and JavaScript) and bypass the dynamic content (API Calls) to the origin.

You can use this approach with AWS CloudFront, where you don’t need to define CDN URLs in your index.html explicitly. However, you need to determine the caching rules (TTL) to instruct the CDN proxy for the static and dynamic content based on path-based routing.

Common Pitfalls to Avoid when using a CDN

When using a CDN, there are common pitfalls that we fall into without knowing the future implications.

Avoid Caching of index.html.

One of the common mistakes we make is to cache everything, including the index.html. While it’s acceptable to cache the index.html for a website, it could have adverse effects on a web application. The problem is that if the user's browser or CDN caches the old version of the index.html, it could lead to incompatibility with the backend.

Therefore it's important not to cache the index.html at all. I would even recommend using the following meta tags to avoid caching the index.html in the browser.

```
<meta http-equiv="Cache-Control" content="no-cache, no-store, must-
revalidate">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Expires" content="0">
```

Selective Invalidation of CDN Cache

This step is a common part of the puzzle. When using the CDN, it doesn't make sense to invalidate the entire cache after a new release. You can classify the asset groups and set up rules to invalidate the asset group, depending on the changes you made in each release.

For example, since the frequent changes happen for JavaScripts and CSS in a frontend application, these two groups could be a part of the permanent invalidation. Still, if you have a separation between your custom JavaScripts and Vendor Scripts, you can further divide it into subgroups.

However, if you do major releases having modifications across a significant portion of the application, you can go for a complete invalidation since the corrective functionality of the application is more important than the added performance benefits from CDN.

Automate CDN Cache Invalidation

When we look at selective cache invalidations, it's crucial to remove the human error of missing an important asset group. Since it could directly affect the application functionality as is discussed above, you can set up rules in your continuous deployments to proceed with automated invalidations.

For example, you can detect the asset changes at the continuous deployment by comparing it with the artifacts at production. If the assets are different, we can automate the invalidation of the CDN cache.

Summary

I hope that this article has increased your awareness of what to look for when you set up a CDN for your web application.

As you have learned, setting up CDN comes with its complexity beyond just caching the content. As I've emphasized the need, it's essential to deliver the correct application behavior at the highest priority and avoid any potential conflicts from caching.

If you handle these challenges, I'm sure the caching with CDN will provide a pleasant user experience to the users with fast loading speeds.

With that note, I conclude this article. I hope you would feel free to raise any questions or enlighten me with your views in the comments below.

. . .

Learn more

10 Ways to Optimize Your React App's Performance

How to optimize performance to deliver an awesome user experience.

blog.bitsrc.io

Quantum Angular: Maximizing Performance by Removing Zone

Experiment: removing Zone from Angular with minimal effort, to boost runtime performance.

blog.bitsrc.io

Build Scalable React Apps by Sharing UIs and Hooks

How to build scalable React apps with independent and shareable UI components and hooks.

blog.bitsrc.io

Web Development   Content Delivery Network   JavaScript   Web App Performance   Frontend