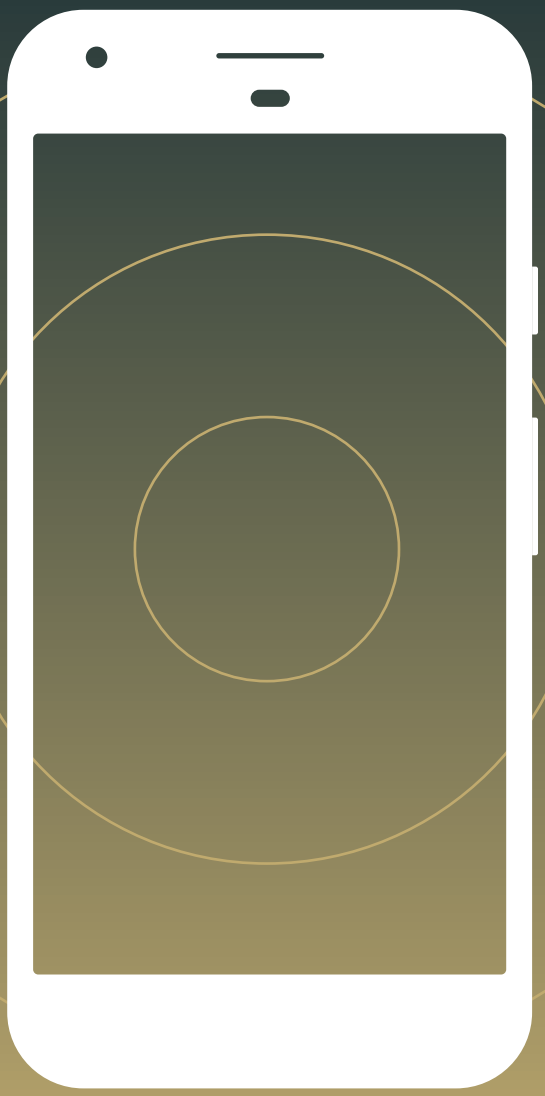
DIVANTE
eCOMMERCE SOFTWARE HOUSE

SEO FOR PWA

HOW TO OPTIMIZE YOUR JS-BASED PROGRESSIVE WEB APPS



REPORT BY

ELEPHATE

DIVANTE
eCOMMERCE SOFTWARE HOUSE

Introduction

The goal of this whitepaper is to help you better understand how search engines perceive JS-powered sites, what their limitations are, and how the potential threats can be prevented to make your PWA a great product in the world of mobile-first indexing where great user experiences take the lead.

Firstly, we introduce the advantages of Progressive Web Apps over native apps and what allows them to deliver engaging and fast user experiences. The process of crawling, rendering, and indexing JS-rich sites by Google is explained, alongside the limitations from its use of the dated version of the Chrome browser to render pages, and the extensive resources needed to execute JavaScript.

The general guidelines to mitigate the most important issues related to not only PWAs, but generally any sites built around the Single Page App model regardless of specific frameworks, as these problems are common across the board, are also included. Lastly, we outline JS rendering solutions that can be used to make any JS-powered site more SEO-friendly, just like any other HTML site.

AUTHOR

Paulina Piotrowicz

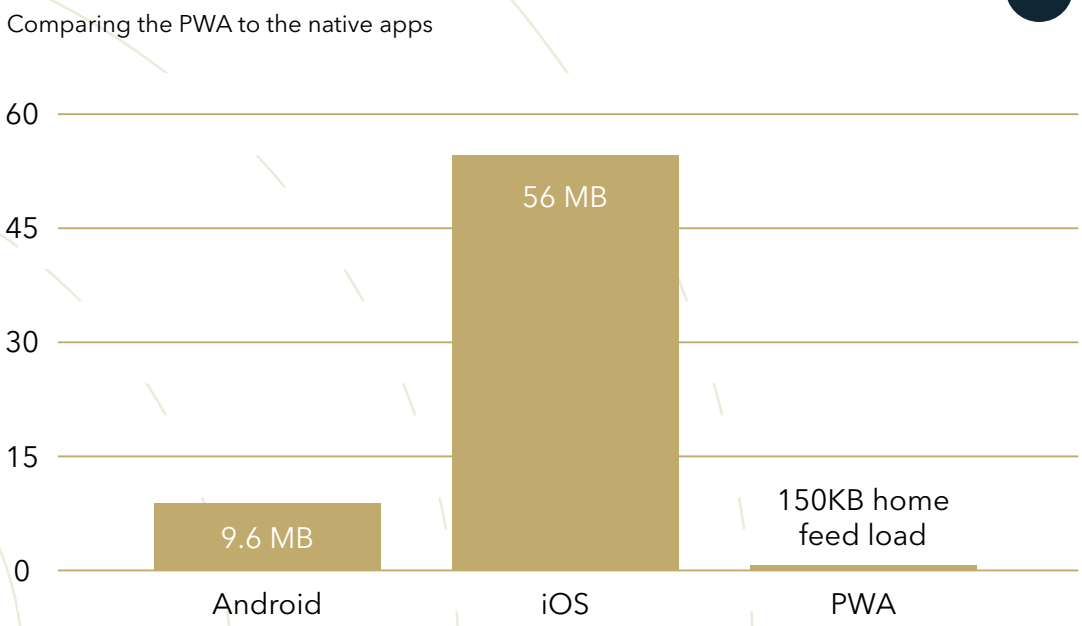
SEO Specialist
Elephate

Introduction to Progressive Web Apps

Progressive Web Apps provide engaging, native app-like experiences, without the excess of local device storage. Thanks to a set of APIs and service workers, PWAs are **reliable** on flabby networks or even **offline**, come with payment mechanisms, and push notifications to **re-engage previous website visitors**.

PWAs are **more accessible** than standard native apps - the distribution model doesn't require users to go to their OS store to download the app. On both Android and iOS, all a user needs to access a PWA is a browser. Once added to the homescreen, a PWA features a permanent icon shortcut. Native-like apps that live on the web are **crawlable** and indexable, enabling the launch and maintenance of one product that not only provides outstanding user experiences on all devices but also satisfies the crawlers. The responsive design of PWAs means the layout is automatically adjusted to fit the screen resolution - be it mobile, tablet or desktop. Great examples of efficient PWAs include Alibaba.com, Lancôme or Starbucks.

Progressive Web Apps are mobile-first oriented and **extremely lightweight compared to native apps**, providing the same level of interactivity. After the initial page renders, they act just like a normal Single Page Application (SPA), where the body contents are dynamically loaded with JavaScript. Even with the subsequent JS chunks transferred to refresh the content, the overall payloads of PWAs are unlikely to exceed the cost of an upfront download of a native app. In the case of Pinterest's PWA, the compressed payload of the app was only 150 KB compared to the staggering 56 MB of a native app required for the same experience on Apple devices.



Source: Addy Osmani - A Pinterest Progressive Web App Performance Case Study

Mobile sites need to load fast and their perceived load speed must feel fast - 53% percent of visits are abandoned if a mobile site takes longer than 3 seconds to load. PWAs utilize placeholders ready to be filled with downloaded content, giving users the sense that the app is loading instantly. Thanks to the Service Workers responsible for heavily caching the app shell containing all the main JS and CSS, users are gratified with the snappy interactiveness on the repeated visits.

JavaScript-based apps can be rendered in the user browser (**client-side rendering** - CSR) or rendered fully on the server (**server side rendered** - SSR), with mixed solutions in-between. We talk about these techniques in detail later on.

Depending on the rendering type you choose, SEO-friendliness, loading speed and the performance of the app will be affected. With mobile page loading speed now a ranking factor, if the JS-powered web app properly mitigates all crawling and indexability issues, its organic visibility can benefit from an outstanding user experience and an improved perceived page load. Even if Googlebot itself is not able to directly benefit from the heavy client-side caching, it's able to track real-world page performance as perceived by of your actual app users. Chrome User Experience Report (CrUX) gathers pagespeed statistics from Chrome browser users, and it's safe to assume that those metrics are then factored in when search rankings are computed.

Current state of JavaScript & SEO

Introduction

As of July 2018, the only search engine able to efficiently parse and execute JavaScript is Google. Although the other bigger player on the market, Bing, is capable of rendering JavaScript, it doesn't do it at scale.

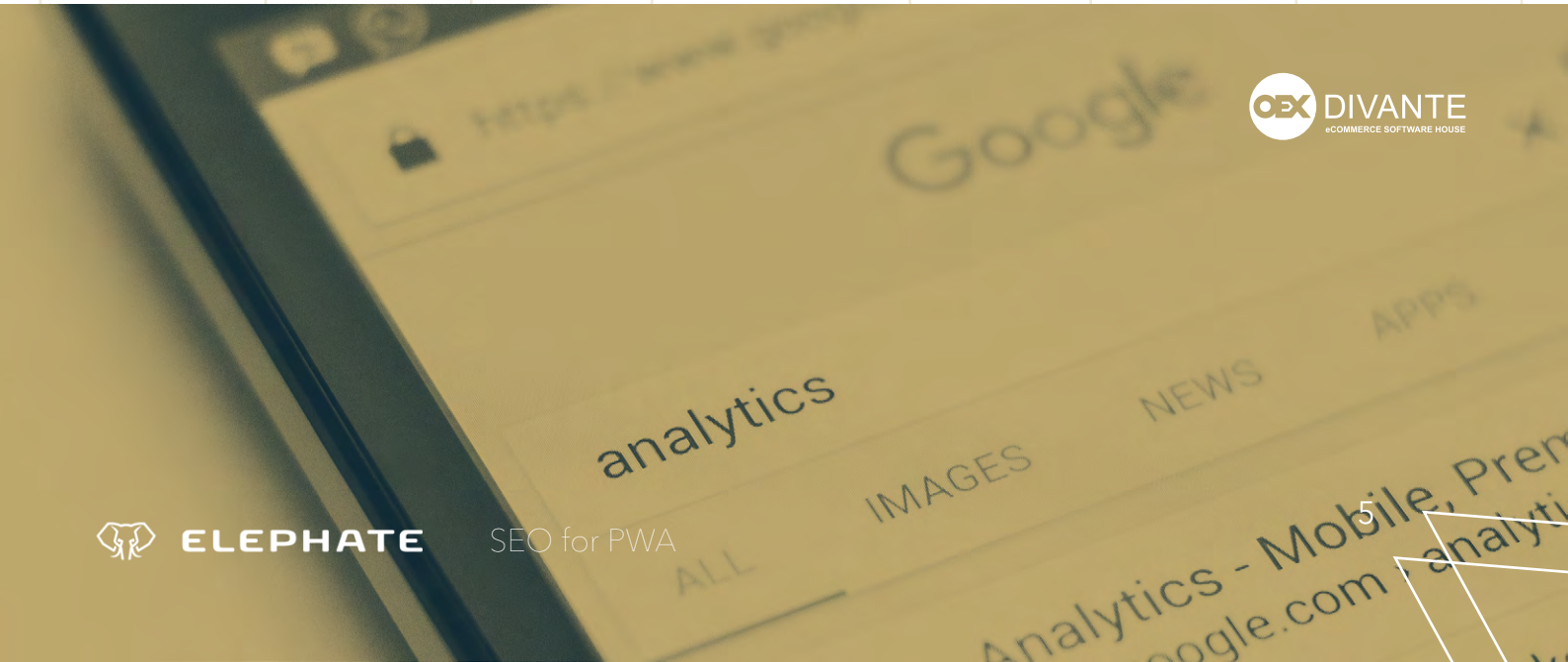
It was back in 2015 when Google first announced that they are generally able to render and understand JS-powered websites just like modern browsers:

> *Times have changed. Today, as long as you're not blocking Googlebot from crawling your JavaScript or CSS files, we are generally able to render and understand your web pages like modern browsers. To reflect this improvement, we recently updated our technical Webmaster Guidelines to recommend against disallowing Googlebot from crawling your site's CSS or JS files.*

Source: https://webmasters.googleblog.com

While it was a great step towards a JS-dominated web, and it's been 3 years since the initial announcement, there are still notable obstacles in maximizing the visibility potential of apps based on pure JS, where literally no content is visible before the scripts are executed.

John Mueller of Google admitted there are some JS-powered sites where Googlebot will be able to handle client-side JS ("some" being the keyword here). It indicates Google will struggle in all other cases:



DIVANTE
eCOMMERCE SOFTWARE HOUSE

Expensive JS execution

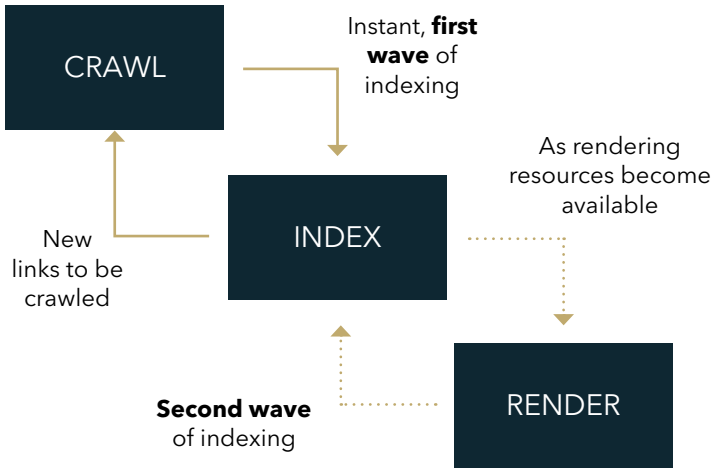
JS is heavy on CPU, time-consuming, and slower to parse than "raw" HTML. Because of limited CPU capabilities, according to a Web Apps Performance Study from Q4/16-Q1/17, JS parse times on mobile devices can be up to 36% longer than on desktop. The size of JS chunks, code quality, and complexity (not every 100 KB load of JavaScript will be equal in terms of the resources required to parse and compile) can noticeably affect battery usage, especially on low-end smartphones with both limited battery life and weaker processors. Longer JS execution times forced on the user's device **delay the first paint of the "hero" content** that the user is keen to see on the screen. This will be visible in metrics such as Time to First Paint and First Contentful Paint that aim to measure how quickly content is served to users - and both are part of the CrUX reports.

Two waves of indexing JavaScript-rich websites

In the case of heavy JS-dependant sites, Googlebot crawls and then queues only content and links discovered upon the first crawl of the server-side rendered response to be immediately indexed. Rendering the remaining client-side JS and extracting JS-injected links is deferred until Google's processing resources become available. Algorithms are used to determine whether a given external resource is essential for a valid render of the page. If its algorithm decides it's not crucial for rendering purposes, the file won't be fetched at this time.

When processing resources become available, Web Rendering Service (WRS) - part of the Caffeine indexer - parses, compiles and executes all JS scripts, including fetching data from databases and external APIs. At this point, Caffeine can index the JS-generated body contents. All new links discovered by WRS are then queued to be crawled by Googlebot and the process repeats.

Waves of Indexing for JS-powered websites



Source: Deliver search-friendly JavaScript-powered websites (Google I/O ,18)

Between the two waves of indexing, some key page elements **crucial for indexing** (such as **meta tags** or **rel canonical**) can be missed by Googlebot if they are not included in the initial server response, or the two versions can be sending mixed signals to Google on how it's supposed to treat your pages.

If your source code uses meta robots tag "index", clearly telling Googlebot the page is indexable, but JS scripts change it to "noindex", the page will initially be indexed anyway. To see the noindex directive (and consequently de-index the page), Google will need to wait until WRS takes over and renders the full Document Model Object (DOM). Dynamic HTML in DOM is what the browser formed after it received the static HTML document and then fetched all the linked assets, including CSS and JS - so DOM includes your JS-generated content.

Unfortunately, for some less-important pages, it can be days until the second wave of crawling kicks in after the initial indexing. Between those two events, the page may as well be found in the search results.

Limitations of Googlebot & the Web Rendering Service

The WRS responsible for rendering is in fact a headless Chrome 41 browser. Released in 2015, it comes with certain limitations when compared to more modern browsers:

- WRS does not support WebSocket protocol
- IndexedDB and WebSQL interfaces are disabled
- Chrome 41 supports ES5 syntax only (limited capabilities in the newer versions)
- Googlebot and WRS support only HTTP/1.x and FTP, with or without TLS
- All user permission requests are automatically denied
- 3D and VR contents are not indexable
- Session data storage & cookies is cleared across page loads - Google wants to see pages just like any first-time user would
- Service Workers are only partially supported for Googlebot (Chrome 41)
- There is no guarantee all your JS files will be downloaded upon the initial crawl of your web page

The general rule of thumb is that if your JS web pages are properly rendered both in Chrome 41 and Search Console using the Fetch as Google tool, the engine should technically be able to correctly see the pages as well.

Googlebot must wait until the page is rendered to discover new links

The rendering of CSR apps is deferred until Google has the resources to do so. But **in order to discover new JS-inserted links to follow, Googlebot must wait for the app to be fully rendered** anyway. It means that your **internal linking structure** will only be re-evaluated once WRS is done with executing JS. Internal links are valuable signals to search engines regarding topical relationships between pages, your site information architecture and the priority of certain page types.

If the initial server response is full of links to not-so-important pages, but the URLs to product categories are missing, you can imagine that those subpar pages will have more exposure (will be crawled more often). Google will prioritize them over your “sales pages” for all the wrong reasons.

Content added upon user interaction won’t be crawled

Content not present in the initial DOM and loaded only upon JavaScript events triggered by user interaction won’t be crawled or indexed by Google. Googlebot is a “lazy user” - it doesn’t click or scroll, so key body content loaded once a “show more” button is clicked, or lazy loading menu links via an AJAX call triggered by a mouseover event won’t be picked up by Google.

Pitfalls of Client-side rendering

In the case of **client-side rendering (CSR)** done in the client browser only (no pre-rendering), what the browser gets is usually a very simple HTML skeleton of a page. With JavaScript disabled, it renders blank or contains very little of the page layout or content, depending on how heavily it relies on JS. This HTML skeleton however contains hyperlinks to JS scripts, which then load all the contents of the page.

Client-side JS websites require additional computing resources from Google to fetch external scripts. Imagine an equivalent page is rendered on the server and served to Googlebot as “pure” HTML only - less of its resources will be wasted on fetching the appended JS files to extract contents and links. The saved computing power and remaining crawl demand can be then assigned for more frequent crawling of the top sales pages.

Client-side rendered JS apps can be made crawlable and indexable, but there are scenarios where Googlebot won’t be able to fully render the pages **due to their limited capabilities**

compared to modern browsers. Your contents may get indexed, but the app simply won't rank high. We talk about those challenges posed by crawlers later on.

Page speed impact on crawl demand

Despite Google's enormous crawling capabilities, **they aren't unlimited** - the crawl budget assigned to every domain is something that larger e-commerce sites shouldn't neglect. Crawl budget is defined as the number of your pages Googlebot is willing to visit in a given time period. It's assigned once a domain is first discovered and then re-evaluated throughout its lifecycle.

Googlebot adjusts the crawling speed to match server performance over time. If a **slow response time** is detected (Time to First Byte - TTFB), or Googlebot requests make **pages fetch noticeably longer** (Time to Last Byte - TTLB), the crawl rate will decrease as Googlebot doesn't want the server to crash and hamper the experiences of real visitors. By the same token, crawler won't be able to fetch as many pages in the same time window as it could if server performance didn't suffer.

PWA features that also impact SEO

Heavy PWA caching client-side

PWA Application Shell contains all HTML, inline CSS and JS required for an accurate page rendering. After the initial render, the Service Worker is responsible for caching the app shell for repeat pageviews, while the body contents will be asynchronously loaded via an API (History API). Users can benefit from an **improved performance on all the subsequent page views** and the app works even after they've gone offline. Thanks to CrUX gathering real-life data, the **improved loading speed from repeat visits will be taken into account for the site's rankings**.

Feature detection & older browser support

Google recommends that feature detection is used to determine what browsers support a given PWA feature, rather than assuming the compatibility based on a single user agent string. This allows to deliver similar

levels of user experiences cross-browser.

Analytic solutions provide statistics on what browsers the most frequent PWA users are on, so that developers are able to adjust all interactive functionalities to match the devices in their target market. If the site relies on pure client-side rendering, always test the browser and feature compliance prior to the app launch. There are some open-source PWA feature detectors to help with that.

Progressive enhancement and graceful degradation approaches aim to deliver a similar user experience to both the newest and older client browsers with limited support of some features. If a browser lacks service worker support, the content should be rendered server-side.

If a PWA uses new features like Fetch API, new syntax or methods, it should rely on polyfill - a piece of JavaScript used to provide modern functionality on older browsers that do not natively support it.

End devices that actually require polyfill code should only receive as much as needed.

Key Takeaway: SEO challenges for JS-powered websites

- If the CSR site goes heavy on the end device's hardware, by proxy it will also be inefficient to crawl. Google will crawl PWAs relying on client-side JS much slower than HTML pages.
- Rendered DOM contains executed JS which can make changes to key SEO elements such as meta or canonical tags. Their values found in the initial source code can contradict the values that are then served in the DOM. What does it mean for you? You may be sending misleading signals to Google and have problems with executing your SEO strategy.
- Dynamically loaded content and links may not be visible to search engines.
- Discrepancies between the server-side rendered and then client-side amends in the DOM can result in missing metadata, incorrect HTTP codes, missing canonicals.
- Typical SEO issues that aren't really JS-dependant and haunt static sites as well: content duplication, poor internal linking, redirect chains and redirect loops, missing pagination links, invalid canonical or hreflang tags.

Making your PWA more SEO-friendly

Let's look into the ways of helping search engines render JS and mitigate the aforementioned threats while retaining your initial PWA architecture. Our set of guidelines allows to mitigate the risks that are normally bound with client-side rendered JS, and present core SEO elements to Googlebot as you would on any other HTML site.

Fix all JS errors

Errors in HTML syntax, such as missing closing tags, are automatically fixed by browsers when parsed - the errors won't appear in the console or affect how the DOM is assembled.

However, JavaScript is more complex and no browser will auto fix JS errors for you. A single JS parsing error in Chrome 41 can result in Googlebot being unable to crawl and index the content. If any of the resources fail to download fully or contain JS errors, the links or content loaded by that script might not be discovered by Googlebot.

Although session data is cleared between page visits, Google

aggressively caches static files on their side in order to effectively render pages at scale. After fixing your JS script errors (i.e. according to the Chrome 41 console log), you need to specifically tell Googlebot the static file has changed (i.e. by cache busting, revalidation, Last-Modified/ETag headers). Otherwise the cached version of the file still containing the errors can be used for rendering instead.

Make your links easy to discover

Googlebot is able to extract and crawl links from **data attributes**, but they do not pass any link equity. We know that the internal linking structure is a strong signal to search engines about which pages are prioritized and should be crawled more frequently. Imagine that your app navigation are only JS links - no equity will be passed to subcategories from the most important "equity bucket" - the homepage.

How do you make your hyperlinks visible to search crawlers? Use standard HTML <a href=""> tags. If an <a href=""> link is appended by JS (it's present in the DOM, but not the source code) it will be sent to Googlebot for crawling only after the initial page has been rendered by WRS. As Google does not support user-like interactions such as clicking and scrolling, the content or links loaded only upon JS events will be invisible to the crawlers.

Transpile ES5+ modules to ES5 for Googlebot

JavaScript features from the ES5 specification are accessible on Chrome 41 and therefore should be executed by Google (it supports modern ES5+ only partially). Services such as Babel allow to transpile modern JavaScript statements to ES5 without worrying about how the page will be parsed by Googlebot or older client browsers. Examples include the "let" declaration, classes, and arrow functions.

This can change when Google decides to upgrade Googlebot & WRS to a later headless Chrome version.

Avoid JS redirects

Googlebot is able to follow JS redirects just as normal meta redirects and tests seem to prove JS redirects pass link equity. However, the recommended method is to still rely on server-side redirects.

JS redirects can be safely used on pages that require the user to login (they won't be accessible to Googlebot), and can be

a fallback solution when server-side redirects cannot be put in place. Also note that JavaScript redirects are slower than the server-side ones.

Represent URL changes with pushState

Deprecated escaped fragment identifiers in URLs („hashbangs" such as https://domain.com/#!page/123/) should be substituted by the use of pushState (part of the History API). PushState **updates the URL in the browser bar when the content changes**. PushState is supported by Googlebot and allows to utilize **SEO-friendly „clean" URLs**, while still fetching content asynchronously. It's good practice to always update to the canonical URL with pushState.

Note: # in URLs are used for anchoring to a certain fragment of the content on the same page and work only client-side - Googlebot ignores whatever is in the slug post-# character.

Make your pages linkable

Contrary to native apps, where the user can only be directed to a deep page after they have agreed to download the entire app, deep-linked PWA pages are quickly accessible to first-time visitors as well. Each screen must have a unique URL address that can be independently accessed in a new browser window, allowing you to integrate your PWA with other marketing channels (e-mail, SMS) through the use of deep links. Deep linking from your homepage to products or categories also allows Googlebot to efficiently crawl and index the content.

HTTP/1.x and HTTPS/2 protocol

PWA Service Workers require the use of secure HTTPS protocol, which entails additional SSL certificate handshakes and is thus initially slower than HTTP. Implementing HTTP/2 alongside HTTPS helps to mitigate the speed issue (one connection efficiently streams multiple responses simultaneously) and adds the HTTP/2 Server Push functionality. Thanks to **Server Push**, the server is able to proactively serve resources upfront, knowing that they will soon be requested by the client browser. Server Push decreases the number of round trips and shortens the time needed for server requests and responses, resulting in faster perceived page speed, comparing to the same resources sent over HTTPS only. Service Worker is then able to fetch and cache assets required for when the user goes online.

As Googlebot doesn't support HTTP/2 or server push, enabling HTTPS with HTTP/2 without ratined HTTP/1.x support will obscure crawling - the application will not be crawlable by search engine bots and may as well get de-indexed. Read more on this in an experiment where HTTP/1.1 was temporarily disabled: Does Googlebot Support HTTP/2? Challenging Google's Indexing Claims.

Make sure your images are optimized and indexable

The same way hyperlinks in data attributes are invisible to Google, images appended in data attributes won't be crawled by Googlebot. For product images to rank in

Image Search, you should use standard HTML <img src=””>.

The Srcset attribute allows to load various image resolutions based on the size of the end device - images served by the server should not be bigger than the resolutions the screen is able to display. If images are lazy loaded, they should be marked with <noscript> tags or structural data so that Googlebot can discover them.

Avoid timeouts that obscure rendering

Many asynchronous JS-based sites fail to render correctly because of the timeouts Googlebot encountered. Many tests have shown that Google waits for the maximum of 5 seconds for static files to download. Because of poor server performance, the spider may not be able to fetch crucial scripts required to fully render the page.

Avoid artificial delays (loaders)

One can argue that there are special cases when loaders can play a beneficial role to real visitors, but all will agree that the spinners shown on the screen for multiple seconds without any evident back-end processes will be a cause for frustration and an increased drop-off rate.

Assure content parity regardless of user agent

Dynamic rendering uses user agent detection to serve appropriate server-side rendered pages to search engine crawlers. Google wants to see your pages exactly as any normal user would, so it’s crucial to make sure all the key body content shown to users matches what is served to Googlebot.

Rendering for JS frameworks

Various JavaScript frameworks support server side rendering either out of the box or with 3rd party pre-rendering services.

Server-side rendering (SSR)

Even though Google is way ahead of other search engines when it comes to crawling and indexing JS-based sites, and is constantly getting better at it, **you still need server-side rendering if SEO is something that your business cares about**.

JS executed client-side (on the browser or by search engine) requires exhaustive memory and CPU capabilities, especially on middle-shelf mobile devices. When all JS is rendered on the server, the browser receives plain HTML which it can begin rendering immediately, without the extra time and resource costs

of external file downloads and script execution. The major benefits of full SSR are:

- relieving the end device from exhaustive memory/battery usage needed to parse and execute JS
- full content is sooner painted on the screen, resulting in faster perceived page load
- assuring Googlebot is served fully crawlable and indexable content - plain HTML is more efficient to "digest" by crawlers
- there is no risk of sending mixed signals to Google - the initial server response already contains the SEO-relevant page elements like meta robots, rel=prev/next and canonical tags, hreflang annotations and so on
- Google is able to immediately assess the internal link structure - all relevant hyperlinks to subpages can be instantly followed by Googlebot upon its first visit of an entry point page

Prerendering

This approach assumes a fully server-side render of the app shell and body contents. Full pre-renders can be served to gracefully degrade on older browsers that don't support service workers. There are several third-party rendering services such as prerender.io or PhantomJS and implementing pre-rendering is usually straightforward for developers.

Third party solutions initially assemble, cache, and then regularly refresh HTML snapshots waiting to be served when a crawler re-visits. Snapshots must always be 100% complete and reflect content changes in real-time, that's why it's recommended to regularly monitor how snapshots are taken.

Pre-rendering is based on serving a heavily cached static page, and hence it won't be optimal for websites that publish loads of new content or make updates on a daily basis. The pre-render may struggle to refresh the data in time and because of that, the HTML response might have gone stale hours or days ago.

Prerendering caches **one universal version of each URL** to be served upon request, you won't be able to implement any significant personalization of product listing pages based on the current user's behavior or click path.

On the other hand, pre-rendering can be an **efficient solution for small sites** that host **static and often stale contents** and **don't use any notable personalization** of the content.

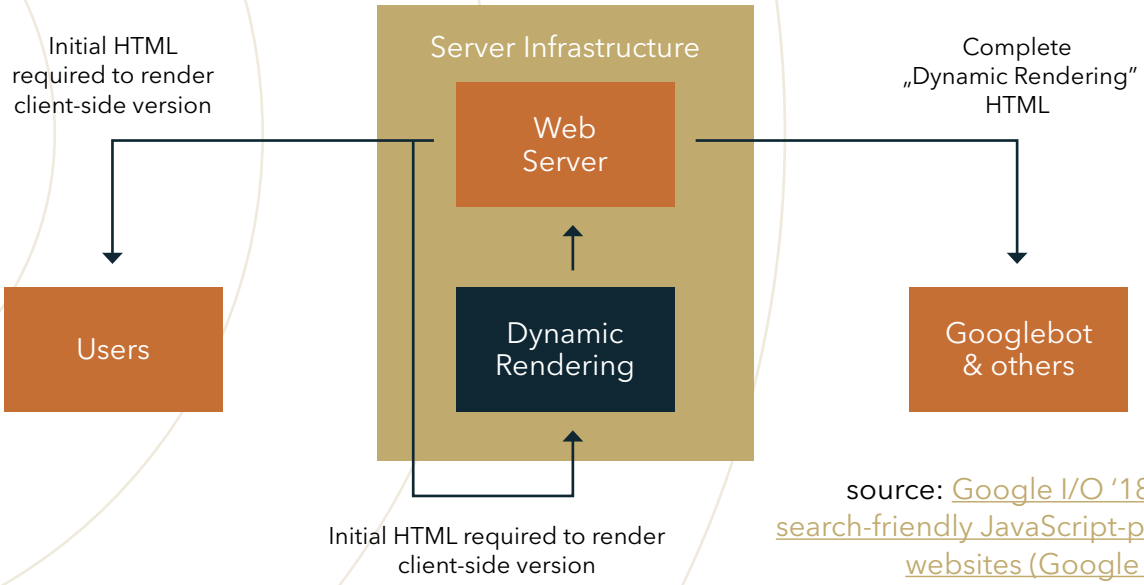
Some pre-rendering services don't offer device specific website snapshots that would match the device resolution - the page layout must be then again recalculated on the client browser, resulting in an additional rendering delay.

Dynamic rendering

In dynamic rendering, based on the detected user agent, the server response contains either a server-side assembled HTML snapshot of a JS-powered page (Googlebot, social media crawlers and others) or equivalent, but still JS-rich content (users). What the browsers get is the initial "untouched" JavaScript that needs to be rendered client-side.

In order to serve the correct version, the server must be able to correctly identify Googlebot by either the user agent string and/or reverse DNS lookup.

Diagram: Dynamic rendering architecture



source: Google I/O '18Deliver search-friendly JavaScript-powered websites (Google I/O ,18)

Ad-hoc SSR allows for strong **personalization** of content, assures it's always **up to date** and **gets indexed faster**. That's why **dynamic rendering is Google's preferred solution** for sites that meet any of the following criteria:

- news sites and publishers where content is ever changing or often updated
- merchants with advanced AI or rule-based product listing personalization
- publishers who heavily rely on a social media presence - social media bots must be able to create valid snippets.

The server-side tools recommended by Google for dynamic rendering include Headless Chrome with Puppeteer or Rendertron.

It's worth noting that it's **a significant change in Google's policy regarding serving content based on a detected user agent**. Before the dawn of JS-powered sites, the practice of a user agent sniffing and serving different source code to crawlers and client

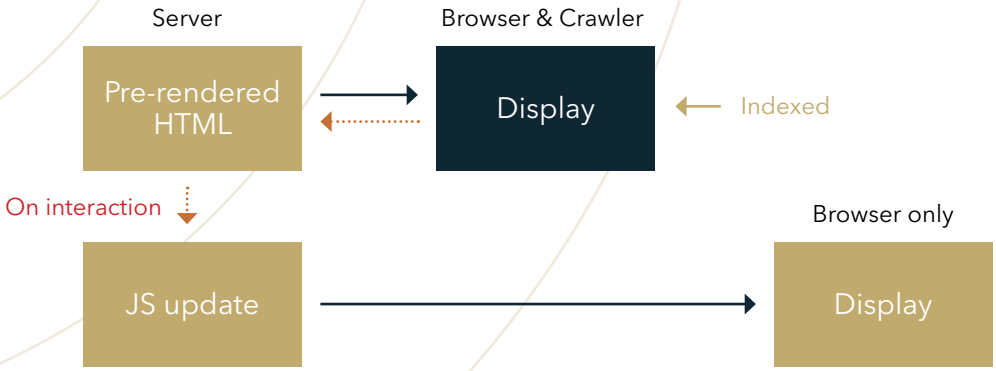
browsers could have been considered cloaking.

Contrary to prerendering, **dynamic rendering doesn't require precaching HTML snapshots** server-side, as they can be assembled on the fly - only when the page is actually requested. And to do that, dynamic rendering will constantly require significant server resources. You need to consider if the costs of dynamic rendering maintenance won't later outweigh the benefits. If your JS-powered site doesn't currently indicate any indexing issues, your content is mostly universal (or you're able to provide enough generic content under each URL for Googlebot to index), dynamic rendering may be overkill. Consider the fact that not all pages need to be crawlable - they can be rendered fully client-site. Even if Googlebot will be unable to extract all the content or follow all the links, it doesn't need to do so.

Hybrid rendering

This is a combined approach: only the **initial page view is rendered server-side** (both the application shell and the content), while **all the JS required to support user interactions and subsequent pageviews is rendered client-side**.

Hybrid rendering architecture



Hybrid rendering makes the initial page load snappier for users and mitigates the complexity of JS parsing for Googlebot. Session storage data is cleared across page views for Googlebot, so what a search engine bot sees upon each server request is the clean server-side rendered HTML - as if each of its page visits was its first. With mobile-first indexing, Googlebot smartphone is also the primary user agent used to crawl the sites - optimizing the app performance for the experience of users on the mobile automatically takes care of efficient content delivery to Googlebot. Hybrid rendering is **Google's long-term recommendation for JS-rich sites**.

How to create an SEO-friendly PWA?

Creating an SEO-friendly PWA with modern JavaScript frameworks is extremely easy. Thanks to tools like Nuxt.js (for Vue.js), Next.js (for React) and Angular Universal, you can enhance your app with Server Side Rendering without writing a single line of code. All of these solutions can deliver SEO-friendly Web Apps out of the box with zero config!

Thanks to the efforts of open source communities, all of the above can be enhanced with PWA capabilities in a few minutes just by installing a dedicated extension and, in most cases, without any knowledge of how this extensions works internally.

Of course, you don't have to believe me and this is why I'll show you how you can create an SEO-friendly PWA powered by Nuxt.js and it's PWA module with just a few lines of code.

AUTHOR

Filip Rakowski

Vue Storefront Front-End Developer Divante

Setting up Nuxt project

The first thing that we need to do is set up our Nuxt.js project. Nuxt is a web app framework built on top of Vue.js with a lot of useful features and plugins out of the box. One of them is excellent SSR support, which is crucial for SEO purposes.

Setting up a Nuxt.js project is a piece of cake and can be done in a few different ways (listed here). The easiest one relies on the Vue Command Line Interface and requires only one command (vue init nuxt-community/starter-template <project-name>). After installing the dependencies (npm install) our project is ready to use with SSR working out of the box. How cool is that?

Adding PWA capabilities

Adding PWA capabilities to an existing Nuxt.js app is as easy as setting it up thanks to the PWA module containing basically everything you'd ever need for a Progressive Web App.

In the modules documentation we can read that PWA capabilities can be added just by installing the package (yarn add @nuxtjs/pwa) and registering it in the nuxt.js config file:

These three lines of code are all you need to enhance your app with basic PWA capabilities.

```
modules: [
    '@nuxtjs/pwa',
],
```

Now the App Shell will be automatically cached which implies two very useful benefits:

- Our app will load instantly even on a slow network connection. Moreover, if we are relying only on server side generated data instead of asynchronous API calls on the client side, our app can work offline at this point without any additional configuration (adding offline support for REST API driven apps shouldn't take more than 5 minutes though.)
- Users can install the app on their homescreens after specifying the apps icon

Of course, we can do much more! The Nuxt.js PWA module is divided into five separate modules. Each of them is responsible for a different PWA functionality.

- **Workbox Module** - Workbox is a tool from Google allowing developers to enhance their applications with PWA capabilities much faster. This module adds Workbox with basic caching capabilities to your application, which

allows instant loads and offline capabilities. This module works out of the box without any additional configuration.

- **Manifest Module & icon Module** - This module automatically generates a Web App Manifest which, combined with the Icon Module, allows users to pin the App to their homescreens. This module works out of the box like the Worbox module. The only thing you need to do is specify the path to your app's icon and voila - you don't need to develop separate Android and iOS apps! The Icon Module will generate the appropriate icons for each device based on the SVG icon you provided.
- **Meta Module** - The Meta Module allows us to easily manage meta tags across the whole application, which can be very useful for SEO purposes. It works out of the box like the ones above.
- **OneSignal Module** - OneSignal is a Web Push client allowing you to send native push notifications to re-engage your users. They will receive these notifications even when the application is closed. Here you can find instructions on how to configure this module.

Bottom line

As you've just seen, setting up an SEO-friendly PWA is a piece of cake these days. Thanks to amazing frameworks and tools, we can achieve it within hours or sometimes even minutes ;)

Conclusion

PWA's are engaging, and deliver a native app-like experience. What's more, they are crawlable, indexable and are able to work offline. The list of advantages of PWAs is even longer. However, we should remember that there are still notable obstacles in maximizing visibility potential of apps based on pure JS. It's been only three years since Google announced indexing of PWA's, and JS-powered websites still come across many challenges. The challenges include, for example, the fact that Google crawls PWAs relying on client-side JS much more slowly than HTML pages or dynamically loaded content, and links may be not visible to search engines.

That being said, there are some ways you could make your PWA more SEO-friendly:

- fix all your JS errors - some of them may result in Googlebot being unable to crawl and index the content
- transpile ES5+ modules to ES5 for Googlebot, make your links easy to discover by using standard HTML tags
- avoid JS redirects and rely on server-side redirects
- represent URL change with pushState (supported by Googlebot and enabling the utilization of SEO-friendly „clean" URLs)
- make your pages linkable - Deep linking from your homepage to products or categories also allows Googlebot to efficiently crawl and index the contents.
- make sure your images are optimized and indexable
- avoid timeouts that obscure rendering
- avoid artificial delays (loaders)
- assure content parity regardless of user agent

We also discussed the different types of rendering: prerendering, dynamic rendering and hybrid rendering, which is a kind of combined approach: only the initial page view is rendered server-side (both the application shell and the contents), while all the JS required to support user interactions and subsequent pageviews is rendered client-side. Hybrid rendering is Google's long-term recommendation for JS-rich sites.

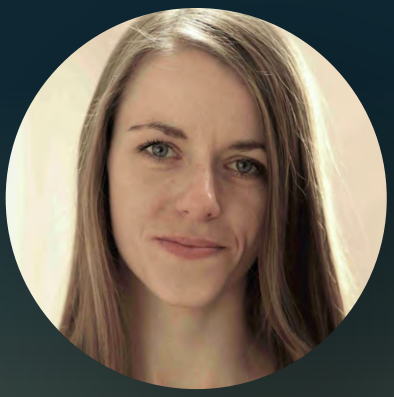
Finally, we showed that creating a SEO-friendly PWA powered by Nuxt.js is extremely easy and requires just a few lines of code!

BEST IMPLEMENTATIONS OF PROGRESSIVE WEB APPS

DOWNLOAD

ELEPHATE DIVANTE eCOMMERCE SOFTWARE HOUSE

If you want to know more about **SEO for PWA**, or **SEO or PWA** in general, contact us!



Maria Cieślak

Senior Technical SEO Specialist
Elephate

maria@elephate.com

Raymond Wojtala

Partnership Manager at Vue Storefront
Divante

ray@vuestorefront.io