SUCCESS AT SCALE

Case studies from the web's finest products

Curated by Addy Osmani

SUCCESS AT SCALE

Case studies from the web's finest products

Curated by

Addy Osmani

Published 2024 by Smashing Media AG, Freiburg, Germany.
All rights reserved.
ISBN: 978-910835-00-9

Copyediting: Owen Gregory
Proofreading: Owen Gregory and Geoff Graham
Cover and interior illustration: Espen Brunborg
Book design: Ari Stiles
Ebook production: Cosima Mielke
Typefaces: Elena by Nicole Dotin and Mija by Miguel Hernández.

Success at Scale was curated by Addy Osmani
Research and editing: Leena Sohoni-Kasture
Case study authors: Nabeel Al-Shamma, Zack Argyle, Benji Bear, Christopher Chedeau, Gareth Clubb, Glenn Conner, Eyal Eizenberg, Ilknur Eren, Darren Hebner, Catherine Houle, Roderick Hsiao, Xuan Huang, Daniel Husar, Renato Iwashima, Ankit Jain, Tomoki Kiraku, Natasha Kosoglov, Sriram Krishan, Kiko Lam, Ohad Laufer, Nolan Lawson, Andrew Lee, Milica Mihajlija, Thomas Nattestad, Addy Osmani, Rob Palmer, José M. Pérez, Barry Pollard, Aaron Shekey, Shunya Shishido, Thomas Steiner, Melanie Sumner, Stacey Tay, Charis Theodoulou, and Oliver Tse.

This book is printed with material from FSC® certified forests, recycled material and other controlled sources.

MIX
Paper from responsible sources
FSC® C083411

Please send errors to: errata@smashingmagazine.com

Dedicated to the brilliant minds moving the web forward.

You inspire us every day.

PERFORMANCE

CAPABILITIES

CONTENTS

Case studies and interviews
with the people who made it happen

Introduction

The success of a web product or service hinges on several factors that encompass more than flashy design. Website performance, capabilities, accessibility, and developer experience ultimately determine the fate of a web development project. In this series, we look at why each of four factors are crucial for success and how they play a critical role in achieving not just functional websites but exceptional ones. Here's a very brief definition for each of the four factors.

- **Performance** is an important factor because a slow-loading website will frustrate users and lead to high bounce rates, making it a critical factor for project success.

- **Capabilities** refers to the app-like features and functionality that a website provides.

- **Accessibility** is the ability of all users to access and use a website, regardless of their disabilities. This includes users with visual impairments, hearing impairments, and motor impairments.

- **Developer experience** refers to the ease with which developers can build and maintain a website. A good developer experience will make it easier for developers to be productive and to create high-quality websites.

This series of case studies will explore how these four factors can contribute to the success of a web development project.

They will also discuss the challenges that developers face in ensuring that their websites are performant, capable, accessible, and developer-friendly.

Join in then, to take a look at these real-world examples, that discuss the pivotal decisions, challenges, and triumphs that helped to shape some of the digital experiences we encounter daily. I hope that these case studies will serve as a roadmap for developers, businesses, and enthusiasts alike, guiding them towards the path of success in our user-centric online world.

Glossary

A list of common terms and references you will find across the book.

API: Application programming interface, which is a software intermediary that allows two applications to talk to each other and serves as a contract of service between them.

ARIA: Accessible Rich Internet Applications, a set of roles and attributes that help to make web content more accessible to people with disabilities (https://smashed.by/ariadocs).

BundlePhobia: Tool that provides the cost (in MBs) of adding an npm package to your bundle. (https://bundlephobia.com/).

Chrome user experience report (CrUX): Get user experience metrics for how real-world Chrome users experience popular destinations on the web. (https://smashed.by/cruxdocs).
Also see CrUX dashboard (https://smashed.by/cruxlauncher).

CI/CD: Continuous integration/continuous delivery, a modern software development practice where frequent code changes are introduced, integrated, and deployed, often using automation.

GraphQL: A query language for APIS and a runtime for fulfilling those queries with your existing data (https://graphql.org/).

Lazy loading: A strategy to load noncritical resources only when they are actually needed.

Lighthouse: An open-source diagnostic tool from Google Chrome for improving the performance, quality, and correctness of your web apps (https://smashed.by/lighthouse).

Next.js: A React framework for creating full-stack web applications at scale (https://nextjs.org/).

npm: Package manager for Node.js (https://www.npmjs.com/).

Performance tools: Tools such as Lighthouse used to measure the performance of web sites. Other common tools: WebPageTest, Page-Speed Insights, Google Search Console.

PWA: Progressive web applications that can provide an app-like experience on the web (https://smashed.by/pwasguide).

React: Popular library for building web and native user interfaces (https://reactjs.org/). React Native (https://reactnative.dev/) is used for developing native applications for Android/iOS devices.

Service workers: Proxy servers that sit between web applications, the browser, and the network (when available) (https://smashed.by/serviceworkersprimer).

SPA: Single page app or a web app where most of the interaction takes place on a single page by dynamically rewriting it with fresh data.

TypeScript: A language that extends JavaScript by adding types to the language (https://www.typescriptlang.org/).

WCAG: Web Content Accessibility Guidelines 2.2 (https://smashed.by/wcagquickref).

Web app manifest: A JSON file where developers can specify the behavior for the PWA after it's installed on a device.

Web vitals: Google's unified guidance for quality signals that are essential to delivering a great user experience on the web (https://web.dev/vitals/). This includes core web vitals such as largest contentful paint (LCP), first input delay (FID), and cumulative layout shift (CLS). Other important metrics that may be used are time to first byte (TTFB), first contentful paint (FCP), time to interactive (TTI), total blocking time (TBT), and interaction to next paint (INP).

Workbox: Production-ready service worker libraries and tooling. (https://smashed.by/workbox).

PERFORMANCE
1420
400
100
1
2
3

PERFORMANCE

Introduction

Performance is one of the most important factors that affect the long-term success of a web application. Suboptimal performance results in frustrated users – and frustrated users are easily tempted to leave. Competing products are available in the market, and if one is slow, users can readily switch to another. This is true for all kinds of apps, from news and social media, to shopping and dating.

Most users lack the patience to wait for pages to load and may even experience some stress while waiting. Moreover, poor performance (due to large page sizes) can also add to the usage cost for users who are accessing the site through a limited data connection. As such, apps with better performance are more likely to retain their users.[1] This automatically translates to better user engagement and resulting business gains.

User-Centric Performance Metrics and Perceived Speed

While fast page loads are crucial, it is important to note that speed is relative. The perceived speed depends on how the user accesses the app; that is, their device and network type. Also, a site that loads progressively may seem faster than one which loads all at once, even if the total time taken to load is the same for both. A fast initial load with a slow response to interactions may also result in a lower perceived speed.

To quantify perceived speed, Google's web vitals initiative aims to provide unified guidance on performance reporting.[2] Metrics like

1 https://smashed.by/retainingusers
2 https://smashed.by/vitals

first contentful paint (FCP), largest contentful paint (LCP), time to interactive (TTI), cumulative layout shift (CLS), and others are used to measure how users experience the performance of a web page. Core web vitals are a subset of these metrics that are essential to capture three key aspects of user experience – loading measured by the LCP, interactivity measured by the first input delay (FID), and visual stability measured by CLS. (Note: This is the current list of core web vitals and may evolve with time.)

As indicated in some of the case studies discussed later, core web vitals can directly correlate with key business metrics. The science[3] and business impact[4] of web vitals have also been discussed in detail on Google and Chrome developers blogs. Optimizing these metrics guarantees the delivery of a fast and seamless navigation experience that delights users, thereby turning them into loyal and returning customers. Thus, they align with the interests of both technical and business stakeholders.

Measuring Performance

Performance metrics allow us to measure and compare performance and its effect on user experience. Teams can decide what metrics to use for measurement based on the user experience they aim to provide. In some situations, they may even go for custom metrics.

There are various tools available that compute metric values using data captured during actual page loads. This data can be captured in a controlled lab environment (lab data) or from the real-life use of the app (field data). Field data is how fast your real users experience your site. Lab data is how fast your site could be (often for a user on a slow to median power device or network). It's impossible to sim-

3 https://smashed.by/vitalsscience
4 https://smashed.by/vitalsimpact

ulate every scenario (combinations of browsers, devices, networks, and operating systems) in a lab environment. Real-user monitoring (RUM) helps to fill this gap in customer experience visibility.

Lighthouse is the recommended option for testing with lab data. Given a page, Lighthouse can run audits against it, assess its performance and generate reports that are easy to understand. Chrome user experience report (CrUX) is a RUM option that can be used to get metrics based on field data from real-world Chrome users' experience.

Budgeting Performance

High performance is rarely a coincidence. Instead, it needs to be planned for. The ability to measure performance metrics allows us to set goals corresponding to the desired metrics for our websites. These goals can be described using performance budgets that are allocated when developing or re-engineering an app for high performance. The team will then work within this budget when choosing image sizes, external JavaScript libraries, and CSS and fonts.

Apps with better performance are more likely to retain their users. This automatically translates to better user engagement and resulting business gains.

A budget should be defined for different types of pages on the site and the device or connection type. They can be set for milestone timings like FCP and TTI mentioned earlier; for example:

- *TTI should be less than 5 seconds for the homepage on a slow 3G connection.*

Budgets can also be based on the quantity of a resource, like the maximum size of images or scripts. An example of a quantity-based budget would be:

- *Under 170 KB of critical path resources (compressed/minified) for the product page on mobile.*

Optimizing Performance

Website performance optimization is a vast topic that could cover multiple books. Here, I would just like to mention that there is probably a better way to serve every resource on a website. You just need to find what works best to help you achieve your budget goals. A summary of optimization techniques used for key resource types is as follows.

1. **Images**: Optimize based on image format, compression type, image size; choose responsive images, lazy-load below the fold images, and evaluate if using image CDNs works.

2. **JavaScript**: serve minified and compressed scripts, code-split where applicable, remove unused code, and serve modern JavaScript to modern browsers.

3. **CSS and fonts**: Inline critical CSS and fonts, defer non-critical CSS, preload web fonts.

4. **Third-party resources**: Evaluate the size of resource versus value-added and check for low-size alternatives. Use async, defer and preconnect where applicable.

Monitoring Performance

We can test which of our optimizations work within the set budgets using Lighthouse.[5] Once the site is out there with real users, Chrome UX reports can help to understand the on-field performance of the site.[6] This public dataset aggregates real user-experience data across millions of websites from users who have opted-in for the tests.

These reports help to gather the field values for metrics like LCP, FID, time to first byte (TTFB), FCP, and CLS. This makes it easy to monitor what is already working and what needs further optimization and allows for continuous improvement based on real data.

Building a Performance Culture

A robust performance culture recognizes speed as one of the core features of the website. Optimization should not be a one-off task but a continuous commitment, especially in the face of new feature requests, which may tend to derail performance. Performance issues should be made known to developers, product managers, and decision-makers alike so that teams can work together to address them.

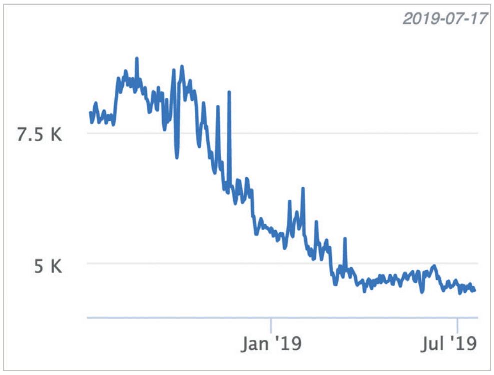
Let us now look at some organizations and products that achieved significant performance improvements by changing the way they measure, budget, optimize, and monitor performance. These organizations have indeed inculcated a strong performance culture.

5 https://smashed.by/lighthouse
6 https://smashed.by/crux

Making Instagram.com Faster

by Glenn Conner

In recent years, Instagram.com has seen a lot of changes – we've launched stories, filters, creation tools, notifications, and direct messaging, as well as myriad other features and enhancements.[1] However, as the product grew, one unfortunate side effect was that our web performance began to suffer. Over the last year, we made a conscious effort to improve this. Our ongoing efforts have thus far resulted in almost 50% cumulative improvement to our feed page load time. This case study will outline some of the work we've done that led to these improvements.



Performance improvements over the last year for feed page display done times (milliseconds).

Correctly prioritizing resource download and execution, and reducing browser downtime during the page load is one of the main levers for improving web application performance. In our case, many of these types of optimizations proved to be more immediately impactful than code size reductions, which tended to be individually small and only began to add up after many incremental improvements.

1 The original version of this case study was published in August 2019. It is the first in a four-part series. Read the entire case study at https://smashed.by/fasterinstagram1 , https://smashed.by/fasterinstagram2 https://smashed.by/fasterinstagram3 & https://smashed.by/fasterinstagram4

They were also less disruptive to product development, requiring less code change and refactoring. So initially we focused our efforts in this area, beginning with resource prefetching.

JavaScript, XHR, and Image Prefetching (and How You Need to Be Careful)

As a general principle, we want to inform the browser as early as possible about what resources are required to load the page. We often know what resources we are going to need ahead of time, but the browser might not become aware of those until late in the page-loading process. These resources mainly include those that are dynamically fetched by JavaScript (other scripts, images, XHR requests, etc.) since the browser is unable to discover these dependent resources until it has parsed and executed some other JavaScript first.

Instead of waiting for the browser to discover these resources itself, we can provide a hint to the browser that it should start working on fetching those resources immediately. The way we do this is by using HTML preload tags. They look something like this:

```
<link rel="preload" href="my-js-file.js" as="script"
type="text/javascript" />
```

At Instagram, we use these preload hints for two types of resources on the critical page-loading path: dynamically loaded JavaScript, and preloading XHR GraphQL requests for data. Dynamically loaded scripts are those that are loaded via `import('...')` for a particular client-side route. We maintain a list of mappings between server-side entry points and client-side route scripts – so when we receive a page request on the server side, we know which client-side route scripts will be required for a particular server-side entry point, and we can add a preload for these route-specific scripts as we render the initial page HTML.

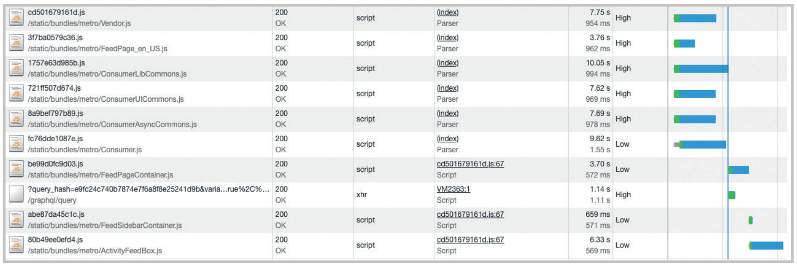
For example, for the FeedPage entry point, we know that our client-side router will eventually make a request for `FeedPageContainer.js`, so we can add the following:

```
<link rel="preload" href="/static/FeedPageContainer.js"
as="script" type="text/javascript" />
```

Similarly, if we know that a particular GraphQL request is going to be made for a particular page entry point, then we should preload that XHR request. This is particularly important as these GraphQL queries can sometimes take a long time, and the page can't render until this data is available. Because of this, we want to get the server working on generating the response as early as possible in the page life cycle.

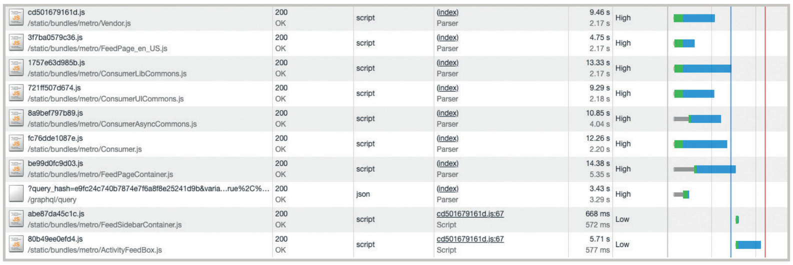
```
<link rel="preload" href="/graphql/query?id=12345" as="fetch"
type="application/json" />
```

The changes to the page load behavior are more obvious on slower connections. With a simulated fast 3G connection (the first waterfall below – without any preloading), we see that *FeedPageContainer.js* and its associated GraphQL query only begin once *Consumer.js* has finished loading. However, in the case of preloading, both *FeedPageContainer.js* and its GraphQL query can begin loading as soon as the page HTML is available. This also reduces the time to load any non-critical lazy-loaded scripts, which can be seen in the second waterfall. Here *FeedSidebarContainer.js* and *ActivityFeedBox.js* (which depend on *FeedPageContainer.js*) begin loading almost immediately after *Consumer.js* has completed.



cd501679161d.js /static/bundles/metro/Vendor.js	200 OK	script	(index) Parser	7.75 s 954 ms	High
3f7ba0579c36.js /static/bundles/metro/FeedPage_en_US.js	200 OK	script	(index) Parser	3.76 s 962 ms	High
1757e63d985b.js /static/bundles/metro/ConsumerLibCommons.js	200 OK	script	(index) Parser	10.05 s 994 ms	High
721ff507d674.js /static/bundles/metro/ConsumerUICommons.js	200 OK	script	(index) Parser	7.62 s 969 ms	High
8a9bef797b89.js /static/bundles/metro/ConsumerAsyncCommons.js	200 OK	script	(index) Parser	7.69 s 978 ms	High
fc76dde1087e.js /static/bundles/metro/Consumer.js	200 OK	script	(index) Parser	9.62 s 1.55 s	Low
be99d0fc9d03.js /static/bundles/metro/FeedPageContainer.js	200 OK	script	cd501679161d.js:67 Script	3.70 s 572 ms	Low
?query_hash=e9fc24c740b7874e7f6a8f8e25241d9b&varia…rue%2C%… /graphql/query	200 OK	xhr	VM2363:1 Script	1.14 s 1.11 s	High
abe87da45c1c.js /static/bundles/metro/FeedSidebarContainer.js	200 OK	script	cd501679161d.js:67 Script	659 ms 571 ms	Low
80b49ee0efd4.js /static/bundles/metro/ActivityFeedBox.js	200 OK	script	cd501679161d.js:67 Script	6.33 s 569 ms	Low

Without preloading.



cd501679161d.js /static/bundles/metro/Vendor.js	200 OK	script	(index) Parser	9.46 s 2.17 s	High
3f7ba0579c36.js /static/bundles/metro/FeedPage_en_US.js	200 OK	script	(index) Parser	4.75 s 2.17 s	High
1757e63d985b.js /static/bundles/metro/ConsumerLibCommons.js	200 OK	script	(index) Parser	13.33 s 2.17 s	High
721ff507d674.js /static/bundles/metro/ConsumerUICommons.js	200 OK	script	(index) Parser	9.29 s 2.18 s	High
8a9bef797b89.js /static/bundles/metro/ConsumerAsyncCommons.js	200 OK	script	(index) Parser	10.85 s 4.04 s	High
fc76dde1087e.js /static/bundles/metro/Consumer.js	200 OK	script	(index) Parser	12.26 s 2.20 s	High
be99d0fc9d03.js /static/bundles/metro/FeedPageContainer.js	200 OK	script	(index) Parser	14.38 s 5.35 s	High
?query_hash=e9fc24c740b7874e7f6a8f8e25241d9b&varia…rue%2C%… /graphql/query	200 OK	json	(index) Parser	3.43 s 3.29 s	High
abe87da45c1c.js /static/bundles/metro/FeedSidebarContainer.js	200 OK	script	cd501679161d.js:67 Script	668 ms 572 ms	Low
80b49ee0efd4.js /static/bundles/metro/ActivityFeedBox.js	200 OK	script	cd501679161d.js:67 Script	5.71 s 577 ms	Low

With preloading.

Benefits of Preload Prioritization

In addition to starting the download of resources sooner, link preloads also have the additional benefit of increasing the network priority of async script downloads. This becomes important for async scripts on the critical loading path because the default priority for these is low. This means that XHR requests and images in the viewport will have higher network priority, and images outside the viewport will have the same network priority. This can cause situations where critical scripts required for rendering the page are blocked or have to share bandwidth with other resources. (If you're interested, see Addy Osmani's "Preload, Prefetch, and Priorities in Chrome" on Medium.)[2] Careful use (more on that in a minute) of preloads gives an important level of control over how we want the browser to prioritize content during initial loads in cases where we know which resources should be prioritized.

2 https://smashed.by/resourcepriorities

Problems with Preload Prioritization

The problem with preloads is that with the extra control it provides comes the extra responsibility of correctly setting the resource priorities. For example, when testing in regions with very slow overall mobile and Wi-Fi networks and significant packet loss, we noticed that `<link rel="preload" as="script">` network requests for scripts were being prioritized over the `<script />` tags of the JavaScript bundles on the critical page rendering path, resulting in an increase in overall page load time.

This stemmed from how we were laying out the preload tags on our pages. We were only putting preload hints for bundles that were going to be downloaded asynchronously as part of the current page by the client-side router.

```
<!-- preloaded async route bundles -->
<link rel="preload" href="SomeConsumerRoute.js" as="script" />
<link rel="preload" href=""..." as="script" />
...
<!-- critical path scripts to load the initial page -->
<script src="Common.js" type="text/javascript"></script>
<script src="Consumer.js" type="text/javascript"></script>
```

Preloading just async route JavaScript bundles.

In the example for the logged out page, we were prematurely downloading (preloading) *SomeConsumerRoute.js* before *Common.js* and *Consumer.js*, and since preloaded resources are downloaded with the highest priority but are not parsed/compiled, they blocked *Common* and *Consumer* from being able to start parsing/compiling. The Chrome data saver team also found similar issues with preloads and wrote about their solution.[3] In their case, they opted to always put

3 https://smashed.by/relpreload

preloads for async resources after the script tag of the resource that requests them. In our case we opted for a slightly different approach. We decided to have a preload tag for all script resources and to place them in the order they would be needed. This ensured that we were able to start preloading all script resources as early as possible in the page (including synchronous script tags that couldn't be rendered into the HTML until after certain server-side data was added to the page), and ensured that we could control the ordering of script resource loading.

```
<!-- preloaded critical path scripts -->
<link rel="preload" href="Common.js" as="script" />
<link rel="preload" href="Consumer.js" as="script" />
<!-- preloaded async route bundles -->
<link rel="preload" href="SomeConsumerRoute.js" as="script" />
...
<!-- critical path scripts to load the initial page -->
<script src="Common.js" type="text/javascript"></script>
<script src="Consumer.js" type="text/javascript"></script>
<script src="SomeConsumerRoute.js" type="text/javascript"
async></script>
```

Preloading all JavaScript bundles.

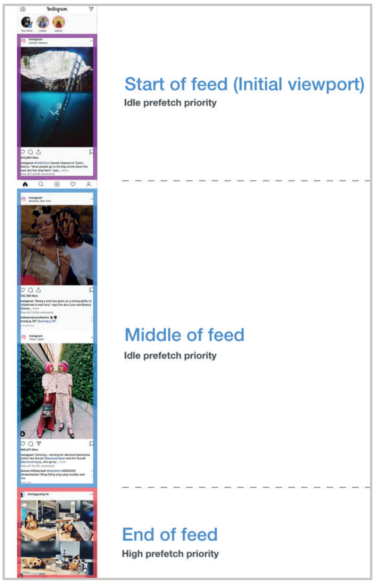
Image Prefetching

One of the main surfaces on *instagram.com* is the feed, consisting of an infinite scrolling feed of images and videos. We implement this by loading an initial batch of posts and then loading additional batches as the user scrolls down the feed. However, we don't want the user to wait every time they get to the bottom of the feed (while we load a new batch of posts), so it's very important for the user experience that we load in new batches before the user hits the end of their current feed.

This is quite tricky to do in practice for a few reasons:

- We don't want off-screen batch loading to take CPU and bandwidth priority away from parts of the feed the user is currently viewing.
- We don't want to waste user bandwidth by being over-eager with preloading posts the user might not ever bother scrolling down to see; but on the other hand, if we don't preload enough, the user will frequently hit the end of feed.
- *Instagram.com* is designed to work on a variety of screen sizes and devices, so we display feed images using the `img srcset` attribute (which lets the browser decide which image resolution to use based on the user's screen size). This means it's not easy to determine which image resolution we should preload and risks preloading images the browser will never use.

The approach we used to solve the problem was to build a prioritized task abstraction that handles queueing of asynchronous work (in this case, a prefetch for the next batch of feed posts). This prefetch task is initially queued at an idle priority (using `requestIdleCallback`), so it won't begin unless the browser is not doing any other important work. However, if the user scrolls close enough to the end of the current feed, we increase the priority of this prefetch task to high by canceling the pending idle callback and thus firing off the prefetch immediately.



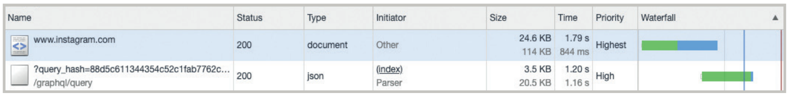
Types of prefetch priorities.

Once the JSON data for the next batch of posts arrives, we queue a sequential background prefetch of all the images in that preloaded batch of posts. We prefetch these sequentially in the order the posts are displayed in the feed rather than in parallel, so that we can prioritize the download and display of images in posts closest to the user's viewport. In order to ensure we actually prefetch the correct size of the image, we use a hidden media prefetch component whose dimensions match the current feed. Inside this component is an `<img>` that uses a `srcset` attribute, the same as for a real feed post. This means that we can leave the selection of the image to prefetch up to the browser, ensuring it will use the same logic for selecting the correct image to display on the media prefetch component as it does when rendering the real feed post. It also means that we can prefetch images on other surfaces on the site – such as profile pages – as long as we use a media prefetch component set to the same dimensions as the target display component.

The combined effect of this was a 25% reduction in time taken to load photos (i.e. the time between a feed post being added to the DOM and the image in that post actually loading and becoming visible), as well as a 56% reduction in the amount of time users spent waiting at the end of their feed for the next page.

Pushing Data Using Early Flushing and Progressive HTML

We've shown how using link preloads allows us to start dynamic queries earlier in the page load; that is, before the script that will initiate the request has even loaded. With that said, issuing these requests as a preload still means that the query will not begin until the HTML page has begun rendering on the client, which means the query cannot start until two network round trips have completed (plus however long it takes to generate the HTML response on the server). As we can see below for a preloaded GraphQL query, even though it's one of the first things we preload in the HTML `head`, it can still be a significant amount of time before the query actually begins.



Name	Status	Type	Initiator	Size	Time	Priority	Waterfall
www.instagram.com	200	document	Other	24.6 KB 114 KB	1.79 s 844 ms	Highest	
?query_hash=88d5c611344354c52c1fab7762c… /graphql/query	200	json	(index) Parser	3.5 KB 20.5 KB	1.20 s 1.16 s	High	

Execution of a preloaded GraphQL query.

The theoretical ideal is that we would want a preloaded query to begin execution as soon as the request for the page hits the server. But how can you get the browser to request something before it has even received any HTML back from the server? The answer is to push the resource from the server to the browser. While it might look like HTTP/2 push is the solution here, there is actually a very old (and often overlooked) technique for doing this that has universal browser support and doesn't have any of the infrastructural complexities of implementing HTTP/2 push. Facebook has been using this successfully since 2010 (see BigPipe),[4] as have other sites in various forms such as eBay – but this technique seems to be largely ignored or unused by developers of JavaScript SPAs. It goes by a few names – early flush, head flushing, progressive HTML – and it works by combining two things:

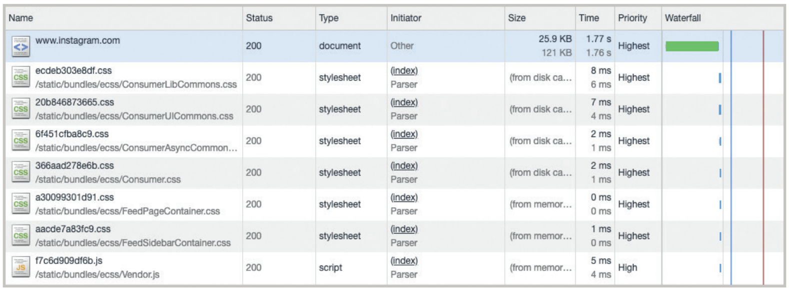
- HTTP chunked transfer encoding
- progressive HTML rendering in the browser

Chunked transfer encoding[5] was added as part of HTTP/1.1, and essentially it allows an HTTP network response to be broken up into multiple chunks that can be streamed to the browser.

4 https://smashed.by/bigpipe
5 https://smashed.by/chunkedtransferencoding

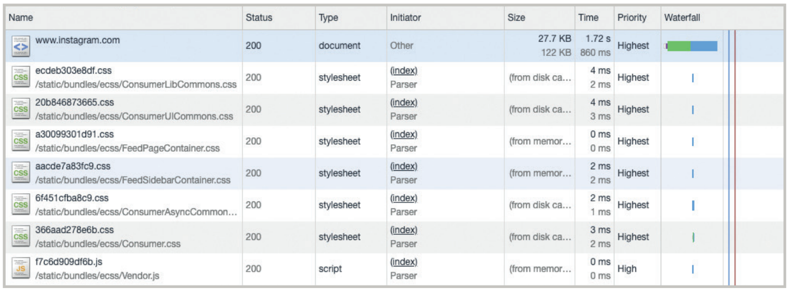
The browser then stitches these chunks together as they arrive into a final completed response. While this does involve a fairly significant change to how pages are rendered on the server side, most languages and frameworks have support for rendering chunked responses (in the case of Instagram we use Django on our web front ends, so we use the `StreamingHttpResponse` object).[6]

The reason this is useful is that it allows us to stream the contents of an HTML page to the browser as each part of the page completes, rather than having to wait for the whole response. This means we can flush the HTML head to the browser almost immediately (hence the term "early flush") as it doesn't involve much server-side processing. This allows the browser to start downloading scripts and style sheets while the server is busy generating the dynamic data in the rest of the page. You can see the effect of this below.



Name	Status	Type	Initiator	Size	Time	Priority	Waterfall
www.instagram.com	200	document	Other	25.9 KB 121 KB	1.77 s 1.76 s	Highest	
ecdeb303e8df.css /static/bundles/ecss/ConsumerLibCommons.css	200	stylesheet	(index) Parser	(from disk ca…	8 ms 6 ms	Highest	
20b846873665.css /static/bundles/ecss/ConsumerUICommons.css	200	stylesheet	(index) Parser	(from disk ca…	7 ms 4 ms	Highest	
6f451cfba8c9.css /static/bundles/ecss/ConsumerAsyncCommon…	200	stylesheet	(index) Parser	(from disk ca…	2 ms 1 ms	Highest	
366aad278e6b.css /static/bundles/ecss/Consumer.css	200	stylesheet	(index) Parser	(from disk ca…	2 ms 1 ms	Highest	
a30099301d91.css /static/bundles/ecss/FeedPageContainer.css	200	stylesheet	(index) Parser	(from memor…	0 ms 0 ms	Highest	
aacde7a83fc9.css /static/bundles/ecss/FeedSidebarContainer.css	200	stylesheet	(index) Parser	(from memor…	1 ms 0 ms	Highest	
f7c6d909df6b.js /static/bundles/ecss/Vendor.js	200	script	(index) Parser	(from memor…	5 ms 4 ms	High	

Without early flush: no resources load until the HTML is fully downloaded.

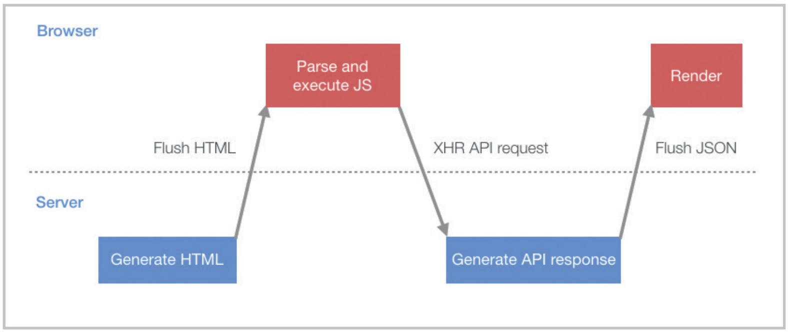


Name	Status	Type	Initiator	Size	Time	Priority	Waterfall
www.instagram.com	200	document	Other	27.7 KB 122 KB	1.72 s 860 ms	Highest	
ecdeb303e8df.css /static/bundles/ecss/ConsumerLibCommons.css	200	stylesheet	(index) Parser	(from disk ca…	4 ms 2 ms	Highest	
20b846873665.css /static/bundles/ecss/ConsumerUICommons.css	200	stylesheet	(index) Parser	(from disk ca…	4 ms 3 ms	Highest	
a30099301d91.css /static/bundles/ecss/FeedPageContainer.css	200	stylesheet	(index) Parser	(from memor…	0 ms 0 ms	Highest	
aacde7a83fc9.css /static/bundles/ecss/FeedSidebarContainer.css	200	stylesheet	(index) Parser	(from memor…	2 ms 2 ms	Highest	
6f451cfba8c9.css /static/bundles/ecss/ConsumerAsyncCommon…	200	stylesheet	(index) Parser	(from disk ca…	2 ms 1 ms	Highest	
366aad278e6b.css /static/bundles/ecss/Consumer.css	200	stylesheet	(index) Parser	(from disk ca…	3 ms 2 ms	Highest	
f7c6d909df6b.js /static/bundles/ecss/Vendor.js	200	script	(index) Parser	(from memor…	0 ms 0 ms	High	

With early flush: resources start loading as soon as the HTML tags are flushed to the browser.

6 https://smashed.by/streaminghttpresponse

Additionally, we can use chunked encoding to send data to the client as it completes. In the case of server-side rendered applications this could be in the form of HTML, but we can push JSON data to the browser in the case of single page apps like *instagram.com*. To see how this works, let's look at the simple case of a single page app starting up. First, the initial HTML containing the JavaScript required to render the page is flushed to the browser. Once that script parses and executes, it will then execute an XHR query which fetches the initial data needed to bootstrap the page.



A Single page app starting up.

This process involves multiple roundtrips between the server and client, and introduces periods where both the server and client are sitting idle. Rather than have the server wait for the client to request the API response, a more efficient approach would be for the server to start working on generating the API response immediately after the HTML has been generated and to push it to the client. This would mean that by the time the client has started up, the data would likely be ready without having to wait for another round trip.

The first step in making this change was to create a JSON cache to store the server responses. We implemented this by using a small inline script block in the page HTML that acts as a cache and lists the queries that will be added to this cache by the server (this is shown in a simplified form on the next page).

```
<script type="text/javascript">
  // the server will write out the paths of any API calls it
plans to
  // run server-side so the client knows to wait for the
server, rather
  // than doing its own XHR request for the data
  window.__data = {
    '/my/api/path': {
        waiting: [],
    }
  };
  window.__dataLoaded = function(path, data) {
    const cacheEntry = window.__data[path];
    if (cacheEntry) {
      cacheEntry.data = data;
      for (var i = 0;i < cacheEntry.waiting.length; ++i) {
        cacheEntry.waiting[i].resolve(cacheEntry.data);
      }
      cacheEntry.waiting = [];
    }
  };
</script>
```

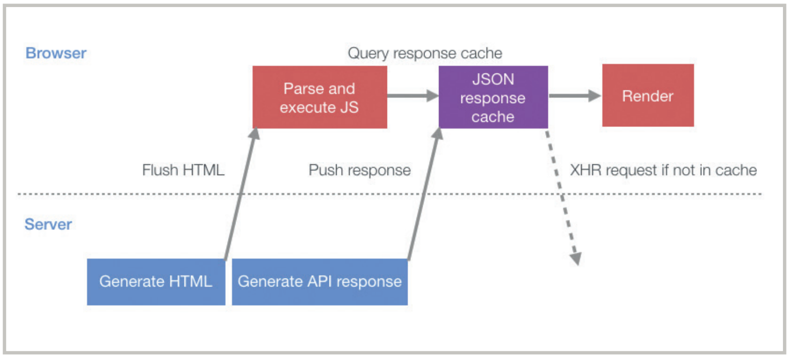
After flushing the HTML to the browser, the server can execute the API query itself and when it completes, flush the JSON data to the page as a `script` tag containing the data. When this HTML response chunk is received and parsed by the browser, it will result in the data being inserted into the JSON cache. A key thing to note with this approach is that the browser will render progressively as it receives response chunks (that is, they will execute complete script blocks as they are streamed in). So you could potentially generate lots of data in parallel on the server and flush each response in its own script

A key thing to note with this approach is that the browser will render progressively as it receives response chunks (that is, they will execute complete script blocks as they are streamed in)

block as it becomes ready for immediate execution on the client. This is the basic idea behind Facebook's BigPipe system where multiple independent pagelets are loaded in parallel on the server and pushed to the client in the order they complete.

When the client script is ready to request its data, instead of issuing an XHR request, it first checks the JSON cache. If a response is present (or pending) it either responds immediately, or waits for the pending response.

This has the effect of changing the page load behavior to this:



Facebook's big pipe system.

Compared to the loading approach discussed previously, the server and client can now do more work in parallel, reducing idle periods where the server and client are waiting on each other. The impact of this was significant: desktop users experienced a 14% improvement in page display completion time, while mobile users (with higher network latencies) experienced a more pronounced 23% improvement.

Instagram Key Takeaways

Correct prioritization of resources to reduce browser downtime during page load is one of the main levers for improving performance.

With new features being introduced frequently, the Instagram engineering team soon recognized the side effects on performance and took steps to improve it. Some of the key enhancements implemented were related to resource prioritization as summarized here.

- Use of preloads to inform the browser about resources that would be required. These were used especially for dynamically loaded JavaScript and XHR GraphQL requests for data.
- Reduce code size before compression as this is what gets parsed and executed on the user's device.
- Push critical dependencies early to the browser using HTTP chunked transfer encoding and progressive HTML rendering.
- Use a cache-first approach with staging which ensures that any interactions performed by the user on the cached state (likes/comments) are applied to the new state from the server.
- Prioritized prefetching of images with higher priority assigned to the next batch of images on the feed as the user scrolls to the end of the current view.
- Serve modern JavaScript bundles (ES2017) to modern browsers.

Interview

Glenn Conner

Former Front-End Engineer at Instagram
Author of “Making Instagram.com Faster”

What excited you or your team the most about the work in the case study?

I was really interested in implementing early/progressive flushing of data from the server. I still find it fascinating how this technique is relatively obscure even today, but that it’s been available since the HTTP 1.1 days; and with the deprecation of HTTP 2 push is again basically the only way to effectively push data to the client during page load.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

After a couple of successful performance experiments I started to get a good sense that there was usually some impact on user metrics, but it was sometimes surprising which metrics were affected. It wasn’t the case that every performance optimization would provide growth or time-spent wins. Optimizing some parts of start-up would sometimes affect lots of metrics, and other times it would just affect a few, and it wasn’t always predictable, so the lesson was to test and measure. We had to spend a lot of time building out the infrastructure to be able to effectively test the impact of these changes and A/B test because testing optimizations is surprisingly hard to do. For example, for transpiling optimizations as you can’t just put some feature flags in your bundle, you need to be able to build a full alternate version of the build and then the server-side logic to serve one set of bundles over another. You also need to take into account other stuff, like more users will have the control bundle already in their cache, which will skew results, so you need to include cache breakers for control group participants, and so on.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

One big difference would be that I would take server-side rendering more seriously. I had kind of given up on it as a concept after it not delivering on previous projects, but I've done a 180 on that after seeing how it's been a key part of the architecture in the new Facebook front end. And the combination of streaming HTML along with new React features like selective hydration has been a game changer for performance.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

One critical decision was to concentrate on RUM-based metrics and not optimize for synthetic benchmarks like Lighthouse scores or JS bundle size. It's certainly easier to test and make optimizations to a synthetic metric, but they are simplifications and proxies to the real performance that your users are experiencing. Depending on the assumptions that go into a synthetic score or metric, if any of those assumptions don't hold true for your application, you're going to be optimizing for things that make no improvements. Every application has its own performance quirks and it's super important that you understand the critical path and nature of those quirks in order to be able to make real improvements.

What came next after the case study was published?

One of the improvements was extracting CSS into its own static bundles. Previously the CSS was inlined into the JavaScript bundles as a string in each JS module that was then inserted into the page on module load. This was terrible for performance owing to double parsing the CSS first as JS and then as CSS plus all the DOM thrash from injecting hundreds of style tags. We knew this was a problem for a long time, but there was some complexity involved in refactoring our build system to support standalone CSS bundles, as well as CSS bundle splitting and loading for dynamically imported JS bundles. We eventually ended up fixing this and it resulted in a decent reduction in JS bundle size and gave an overall page-load improvement, as well as improving cache hit rates, as CSS changes wouldn't invalidate JS bundle hashes anymore (or vice versa).

Do you have any advice for teams that would like to follow in your footsteps?

The one piece of optimization I never got to really make progress on was effective bundle splitting. Instagram still eagerly parses and loads way too much JS. This is fixable, but would require some pretty big changes to the build infrastructure as well as the product architecture. Some of the changes that the Facebook team have shared about the new FB front end point toward some of the approaches that are important for optimizing the delivery of a large JS app.

Has the site changed significantly since the case study was published?

The general architecture is pretty similar, but there's a lot more that has been added in terms of functionality to bring the website up to feature parity with the native apps – things like direct messaging, for example.

Shopify: Want to Improve UI Performance? Start by Understanding Your User.

By Darren Hebner

In 2019,[1] my team at Shopify did a deep dive into the performance of the Marketing section in the Shopify admin.[2] Our focus was to improve the UI performance. This included a mix of improvements that affected load time and perceived load time, as well as any interactions that happen after the merchant has landed in our section.

It's important to take the time to ask yourself what the user (in our case, the merchant) is likely trying to accomplish when they visit a page. Once you understand this, you can try to unblock them as quickly as possible. UI developers can look for opportunities to optimize for common flows and interactions the merchant is likely going to take. This helps us focus on improvements that are user-centric instead of just trying to make our graphs and metrics look good.

I'll dive into a few key areas that we found made the biggest impact on UI performance:

- How to assess your current situation and spot areas that could be improved.
- Prioritizing the loading of components and data.
- Improving the perceived loading performance by taking a look at how the design of loading states can influence the way users experience load time.

1 This case study was originally published in September 2019: https://smashed.by/shopifyperf
2 https://smashed.by/shopifyadmin

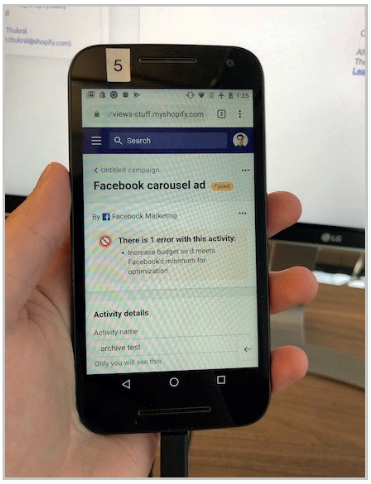
Our team has always kept performance[3] top of mind. We follow industry best practices like route-based bundle splitting[4] and are careful not to include any large external dependencies. Nevertheless, it was still clear that we had a lot of room for improvement.

The front end of our application is built using React, GraphQL, and Apollo.[5] The advice in this case study aims to be framework-agnostic, but there are some references to React-specific tooling.

Assess Your Current Situation

DEVELOP MERCHANT EMPATHY BY TESTING ON REAL-WORLD DEVICES

In order to understand what needed to be improved, we had to first put ourselves in the shoes of the merchant. We wanted to understand exactly what the merchant is experiencing when they use the Marketing section. We should be able to offer merchants a quality experience no matter what device they access the Shopify admin from.



Moto G3 device.

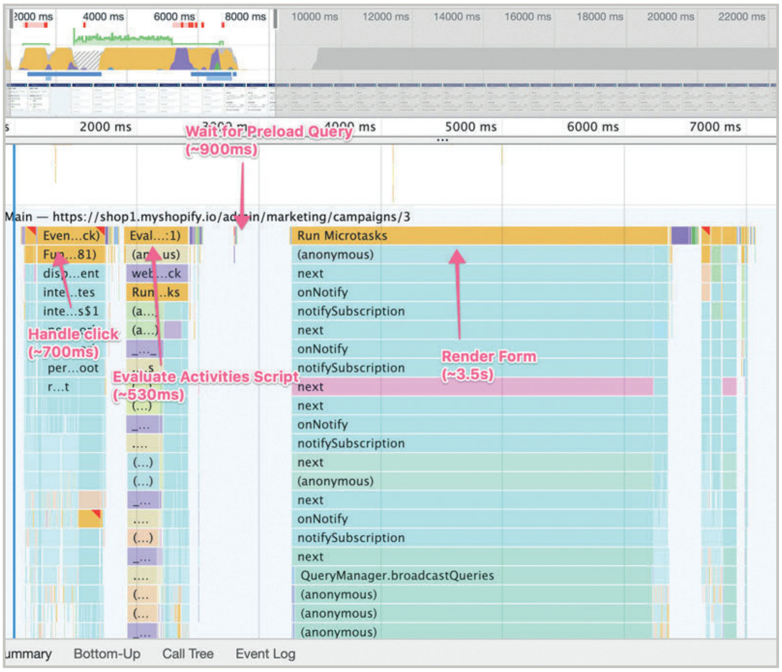
We think testing using real, low-end devices is important. Testing on a low-end device allows us to ensure that our application performs well enough for users who may not have the latest iPhone or Macbook Pro.[6]

We grabbed a Moto G3 and connected the device to Chrome developer tools via the remote devices feature. If you don't have

3 https://smashed.by/shopifyperf
4 https://smashed.by/bundlesplitting
5 https://www.apollographql.com/
6 https://smashed.by/slightlylate

access to a real device to test with, you can make use of *WebPageTest.org* to run your application on a real device remotely.

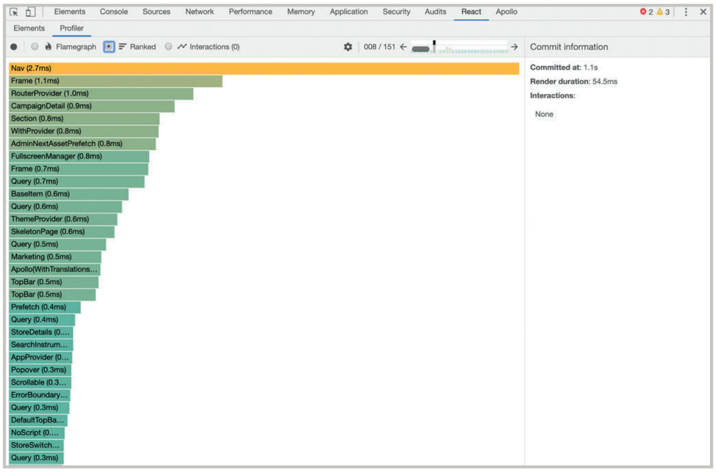
CAPTURE AN INITIAL PROFILE


Wait for Preload Query (~900ms)
Handle Click (~700ms)
Evaluate Activities Script (~530ms)
Render Form (~3.5s)
Main — https://shop1.myshopify.io/admin/marketing/campaigns/3
Run Microtasks
Summary
Bottom-Up
Call Tree
Event Log

Our initial performance profile was captured using Chrome DevTools.

After capturing our initial profile using the performance profiler included in the Chrome developer tools, we needed to break it down. This profile gives us a detailed timeline of every network request, JavaScript execution, and event that happens during our recording, plus much, much more. We wanted to understand exactly what happens when a merchant interacts with our section.

We ran the audit with React in development mode so we could take advantage of the user timings they provide. Running the application with React in production mode would have performed better, but having the user timings made it much easier to identify which components we need to investigate.



Nav (2.7ms)
Frame (1.1ms)
RouterProvider (1.0ms)
CampaignDetail (0.9ms)
Section (0.8ms)
WithProvider (0.8ms)
AdminNextAssetPrefetch (0.8ms)
FullscreenManager (0.8ms)
Frame (0.7ms)
Query (0.7ms)
BaseItem (0.6ms)
Query (0.6ms)
ThemeProvider (0.6ms)
SkeletonPage (0.6ms)
Query (0.5ms)
Marketing (0.5ms)
Apollo(WithTranslations...
TopBar (0.5ms)
TopBar (0.5ms)
Prefetch (0.4ms)
Query (0.4ms)
StoreDetails (0....
SearchInstrum...
AppProvider (0...
Popover (0.3ms)
Scrollable (0.3...
ErrorBoundary...
Query (0.3ms)
DefaultTopBa...
NoScript (0....
StoreSwitch...
Query (0.3ms)

Committed at: 1.1s
Render duration: 54.5ms
Interactions:
None

React profiler from React DevTools.

We also took the time to capture a profile using the profiler provided by React DevTools. This tool allowed us to see React-specific details, like how long it took to render a component, or how many times that component has been updated. The React profiler was particularly useful when we sorted our components from slowest to fastest.

Get Your Priorities in Order

After reviewing both of these profiles, we were able to take a step back and gain some perspective. It became clear that our priorities were out of order.

We found that the components and data most crucial to merchants were being delayed by components that could have been loaded at a later time. There was a big opportunity here to rearrange the order of operations in our favor, with the ultimate goal of making the page useful as soon as possible.

We know that the majority of visits to the Marketing section are incremental. This means that the merchant navigated to the Marketing section from another page in the admin. Because the admin is a single page app, these incremental navigations are all handled client-side (in our case using React Router). This means that traditional performance metrics like time to first byte or first meaningful paint may not be applicable. We instead make use of the Navigation Timing API[7] to track navigations within the admin.

When a merchant visits the Marketing section, the following events happen:

- JavaScript required to render the page is fetched.
- A GraphQL query is made for the data required for the page.
- The JavaScript is executed and our view is rendered with our data.

Any optimizations we do will be to improve one of those events. This could mean fetching less data and JavaScript, or making the execution of the JavaScript faster.

DEPRIORITIZE NON-ESSENTIAL COMPONENTS AND CODE EXECUTION

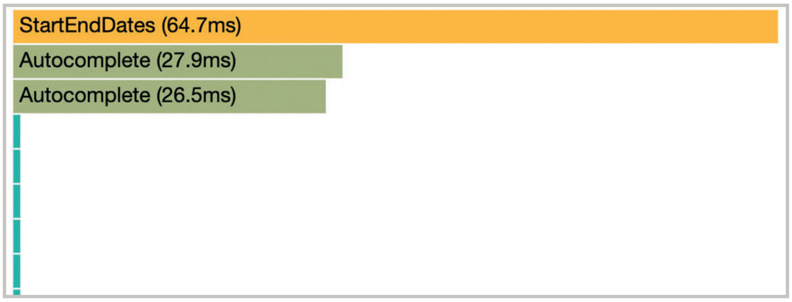
We wanted the browser to do the least amount of work necessary to render our page. In our case, we were asking the browser to do work that did not immediately benefit the merchant. This low-priority work was getting in the way of more important tasks. We took two approaches to reducing the amount of work that needed to be done:

- Identifying expensive tasks that are being run repeatedly and memoize (~cache) them.
- Identifying components that are not immediately required and deferring them.

7 https://smashed.by/navigationtimingapi

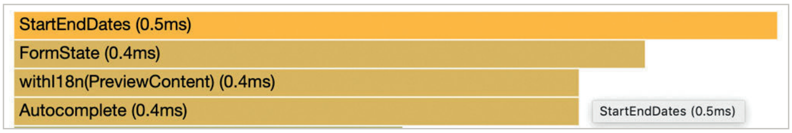
MEMOIZE REPETITIVE AND EXPENSIVE TASKS

One of the first wins here was around date formatting. The React profiler was able to identify one component that was significantly slower than the rest of the components on the page.



React profiler identifying `<StartEndDates />` *component is significantly slower.*

The `<StartEndDates />` component stood out. This component renders a calendar that allows merchants to select start and end dates. After digging into this component, we discovered that we were repeating a lot of the same tasks over and over. We found that we were constructing a new `Intl.DateTimeFormat` object every time we needed to format a date. By creating a single `Intl.DateTimeFormat` object and referencing it every time we needed to format a date, we were able to reduce the amount of work the browser needed to do in order to render this component.



`<StartEndDates />` *after memoization of two other date formatting utilities.*

This, in combination with the memoization of two other date formatting utilities, resulted in a drastic improvement in this component's render time, taking it from ~64.7 ms down to ~0.5 ms.

DEFER NON-ESSENTIAL COMPONENTS

Async loading allows us to load only the minimum amount of JavaScript required to render our view. It is important to keep the JavaScript we load small and fast as it contributes to how quickly we can render the page on navigation.

One example of a component that we decided to defer was our `<ImagePicker />`. This component is a modal that is not visible until the merchant clicks a **Select image** button. Since this component is not needed on the initial load, it is a perfect candidate for deferred loading.

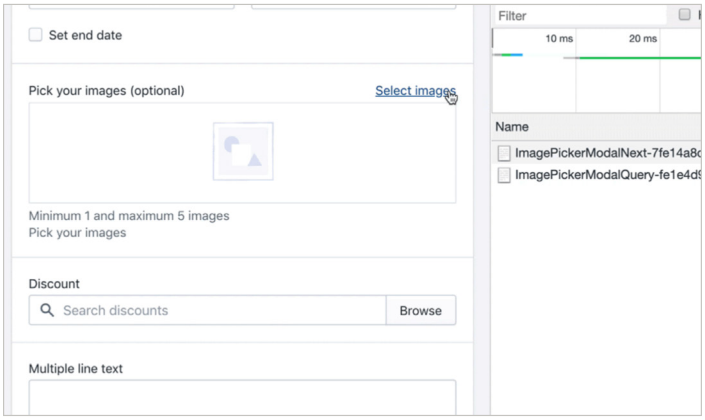
By moving the JavaScript required for this component into a separate bundle that is loaded asynchronously, we were able to reduce the size of the bundle that contained the JavaScript that is critical to rendering our initial view.

GET A HEAD START

Deferring the loading of components is only half the battle. Even though the component is deferred, it may still be needed later on. If we have the component and its data ready when the merchant needs it, we can provide an experience that really feels instant.

Knowing what a merchant is going to need before they explicitly request it is not an easy task. We do this by looking for hints the merchant provides along the way. This could be a hover, scrolling an element into the viewport, or common navigation flows within the Shopify admin.

In the case of our `<ImagePicker />` modal, we do not need the modal until the **Select image** button is clicked. If the merchant hovers over the button, it's a pretty clear hint that they will likely click. We start prefetching the `<ImagePicker />` and its data so by the time the merchant clicks we have everything we need to display the modal.


Set end date
Pick your images (optional)
Select images
Minimum 1 and maximum 5 images
Pick your images
Discount
Search discounts
Browse
Multiple line text
Filter
10 ms
20 ms
Name
ImagePickerModalNext-7fe14a8c
ImagePickerModalQuery-fe1e4d9

Prefetching the image picker when the merchant hovers over the activator button makes it feel like the modal instantly loads.

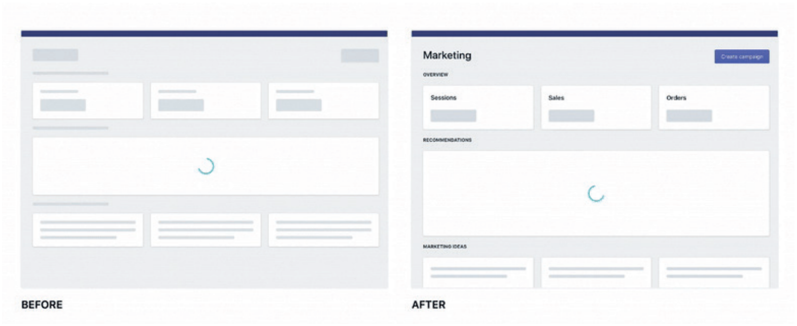
Improve the Loading Experience

In a perfect world, we would never need to show a loading state. In cases where we are unable to prefetch, or the data hasn't finished downloading, we fallback to the best possible loading state by using a spinner or skeleton content. We typically choose to use a skeleton if we have an idea what the final content would look like.

USE SKELETONS

Skeleton content has emerged as a best practice for loading states. When done correctly, skeletons can make the merchant feel like they have "arrived" at the next state before the page has finished loading.

Skeletons are often not as effective as they could be. We found that it's not enough to put up a skeleton and call it a day. By including static content that does not rely on data from our API, the page will feel a lot more stable as data arrives from the server. The merchant feels like they have arrived instead of being stuck in an in-between loading state.

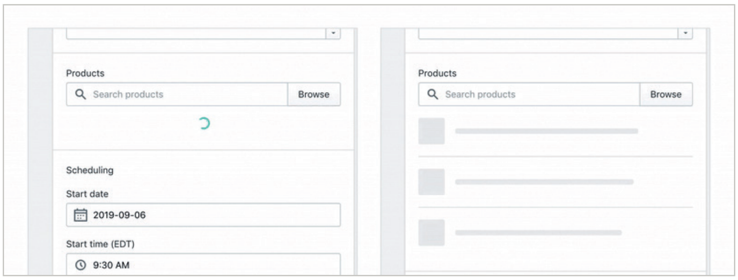


These images show how adding headings helps the merchant understand what content they can expect as the page loads.

Small tweaks like adding headings to the skeleton go a long way. These changes give the merchant a chance to scan the page and get a feel for what they can expect once the page finishes loading. They also have the added benefit of reducing the amount of layout shift that happens as data arrives.

IMPROVE STABILITY

When navigating between pages, there are often going to be several loading stages. This may be caused by data being fetched from multiple sources, or the loading of resources such as images or fonts.

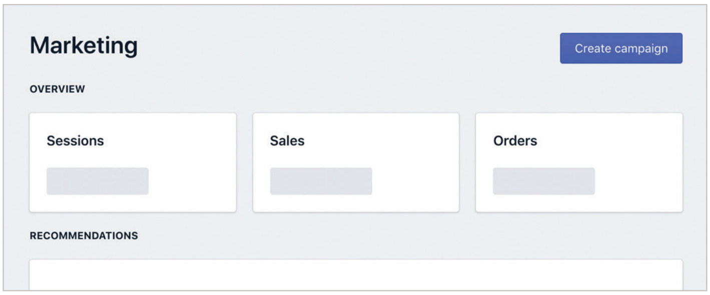


Using a skeleton to help improve stability by matching the height of the skeleton to the height of the final content as closely as possible.

As we move through these loading stages, we want the page to feel as stable as possible. Drastic changes to page layout are disorienting and can even cause the user to make mistakes.

MAKE THE PAGE USEFUL AS QUICKLY AS POSSIBLE

In this example, you can see that we are rendering the **Create campaign** button while we are still in the loading state. We know this button is always going to be rendered, so there's no sense in hiding it while we are waiting for unrelated data to arrive. By showing this button while still in the loading state, we unblock the merchant.



Rendering the ***Create campaign*** *button while we are still in the loading state.*

No Such Thing as Too Fast

The deep dive helped our team develop best practices that we are able to apply to our work going forward. It also helped us refine a performance mindset that encourages exploration. As we develop new features, we can apply what we've learned while always trying to improve

The deep dive helped our team develop best practices that we are able to apply to our work going forward.
It also helped us refine a performance mindset that encourages exploration.

on these techniques. Our focus on performance has spread to other disciplines like design and research. We are able to work together to build up a clearer picture of the merchant's intent so we can optimize for this flow.

Shopify Key Takeaways

Make the page useful as quickly as possible to enhance the user experience.

To identify performance gaps in their Marketing section, the Shopify team decided to determine a user's perspective: in this case, their merchants. They developed merchant empathy by testing the site on real-world devices. This led to the discovery that crucial components and data were being delayed by components that could have been loaded at a later time. They addressed this issue by:

- Identifying expensive tasks that run repeatedly and caching them.
- Identifying components that are not immediately required and deferring them.
- Using skeletons to load content to improve the perceived loading speed.
- Improving stability between transitions during page load.

These changes helped the Shopify team to improve their overall user-centric performance metrics.

Interview

Darren Hebner

Senior Front-End Developer at Shopify
Author of "Shopify: Want to improve UI Performance? Start by Understanding Your User."

What excited you or your team the most about the work in the case study?

For me, the exciting thing about web performance, and the UI side of things in particular, is how so much of it is sleight of hand. You get to be creative with your solutions. Lazy-loading offscreen content and preparing for the user's next interaction by preloading are examples of this.

If everything goes right, the interface starts to disappear. When a merchant is using our admin interface, they are trying to focus on their business. If the merchant is stuck waiting for a page to load or thrown off by a large visual shift on the page, it's distracting them from accomplishing their goal. That's why it's so important to focus on these details.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

I was particularly pleased with how this work opened up conversations with other teams at Shopify. The blog post originally started as an internal document that was shared with other teams working in Shopify's admin. I later recycled some of the content into a short presentation that I gave at other Shopify offices.

If everything goes right, the interface starts to disappear.

There were, of course, already many people at Shopify who cared about the topics I touched on in the blog post, and this was a great way for us to connect and amplify the conversation around UI performance.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

If I were to do the same project today, I'd have put more focus on tracking the progress of our UX improvements over time. With UX changes, we often judged a change by how it felt instead of how much it affected the metrics.

We did pay attention to metrics that reflected the timing of various page-load events, but we didn't have a metric for the visual loading experience. The cumulative layout shift metric would have been an excellent way for us to track this. I'm looking forward to using it on future projects.

Improving Third-Party Web Performance at the *Telegraph*

by Gareth Clubb

At the *Telegraph* we're currently going through a process of rebuilding our public-facing website.[1] This gives us the opportunity to take learnings from the existing build but also write some of the code from scratch. You can see our progress so far on pages such as Gaming[2] and Culture.[3] We can monitor the impact this has on metrics across usability, performance, and accessibility.

We're incredibly passionate about the web, particularly performance, and regularly attend local meet-ups such as LDNWebPerf[4] to learn from others in the industry. We knew web performance was crucial to the success of our new website, but didn't know how to introduce the topic into an organisation with more than 160 years of history and in excess of 1,000 employees. We recognised that with millions of visits per month, performance improvements would be of huge value to our users.

We started slowly, fixing the immediate problems within our technical control: this brought good results but only accounted for 5% of the requests. We knew we would have to start working with the wider business to tackle the rest.

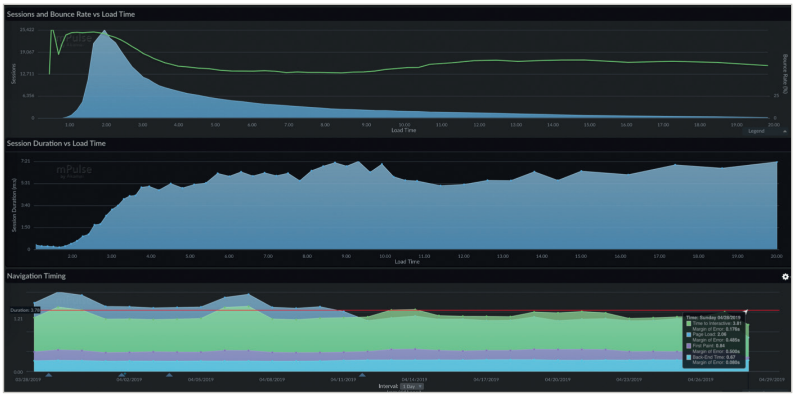
Improving the performance impact of third-party scripts on a website takes time; results won't come overnight, but by being patient and chipping away slowly, eventually these efforts will be rewarded.

1 This case study was originally published online in April 2019: https://smashed.by/telegraphengineering
2 https://smashed.by/gaming
3 https://smashed.by/culture
4 https://ldnwebperf.org/

Creating a Performance Culture

The most important and hardest thing to achieve when tackling web performance at the *Telegraph* was trying to instill a performance culture. A lot of the performance challenges we have faced have not been technical but have been "organizational." Everyone wants "that tag" on a page which will make the organisation money, so it was very important that we get the right individuals in a room to educate, challenge, and work together.

We set up a web performance working group and invited people from across the company – covering advertising, marketing, commercial and, of course, technology. The meetings are run fortnightly, and we use them as an opportunity to review third-party tags, discuss current challenges, and work as a cross-organizational team to try to make our web pages as fast as possible.



Akamai's mPulse showing important front-end metrics for a subset of pages.

Making web performance visible to non-technical teams will keep it at the forefront of their minds when they consider any additional requests, so we're now working to try to build different real user monitoring (RUM) dashboards for different teams. For example: one built around key advertising metrics that is on display in the advertising area; and one built for marketing so that if any regressions happen from a third party, we can help show them where that came from.

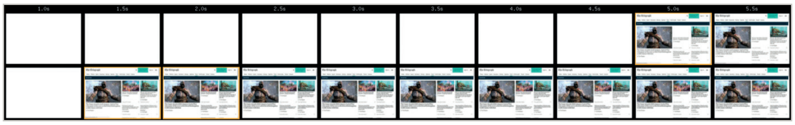
Breaking Down the Silos (Build a Rapport)

There are a lot of different teams at the *Telegraph*, some with an entry level onto the website – through tag managers, embeds, and advertising. As the core technical team, we found it imperative to work with these teams to show support. We would never dismiss anyone or anything but simply be on hand to offer best-practice advice.

Over time we built up a good rapport across the organization, and that gave us very early insight into what was potentially being added onto the site, when it was being added (so we could keep a record) and, most importantly, we could offer advice for the benefit of our end users.

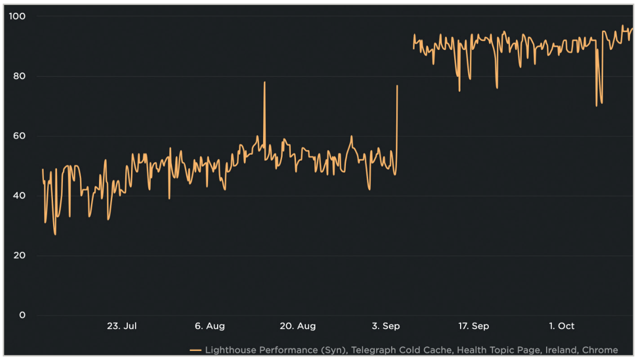
Deferring All JavaScript

The single biggest improvement (and easiest to implement technically) came from deferring all JavaScript, including our own, by adding the `defer` attribute to each script tag.

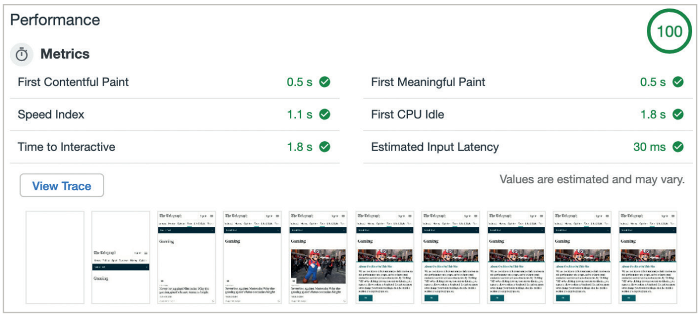

WebPageTest filmstrip comparing before and after deferred scripts.

Mention you're going to delay JavaScript execution to any marketing, advertising, or analytics team and you may see hesitation and reluctance in their reply. They worry about lost revenue if adverts don't load fast enough, or worry that analytics will be skewed, which won't reflect true visitor numbers. Both of which could have an effect on the entire organization's income.

We created a deferred JavaScript test on our QA environments and watched the statistics over a number of days. The results were promising:



Nearly 100% improvement in Lighthouse performance score (unthrottled) when scripts were deferred.



Performance

100

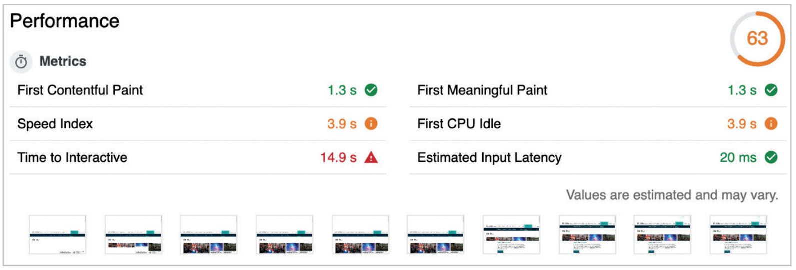
Metrics

First Contentful Paint	0.5 s ✓	First Meaningful Paint	0.5 s ✓
Speed Index	1.1 s ✓	First CPU Idle	1.8 s ✓
Time to Interactive	1.8 s ✓	Estimated Input Latency	30 ms ✓

Values are estimated and may vary.

Unthrottled Lighthouse performance score (https://www.telegraph.co.uk/gaming/)

We’ve made huge strides in the right direction to improve our overall performance scores, but we’re well aware that there is still a huge amount to be done, especially around mobile devices and slower connections.


Performance

63

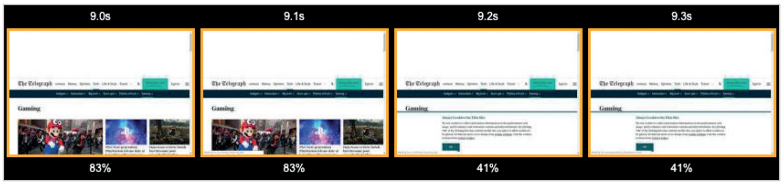
Metrics

First Contentful Paint	1.3 s ✓	First Meaningful Paint	1.3 s ✓
Speed Index	3.9 s ●	First CPU Idle	3.9 s ●
Time to Interactive	14.9 s ▲	Estimated Input Latency	20 ms ✓

Values are estimated and may vary.

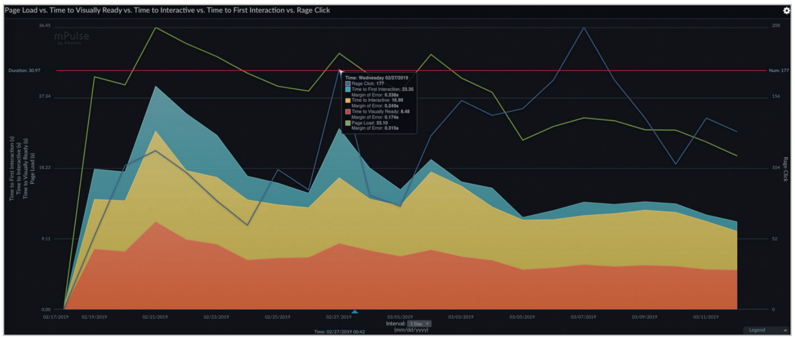
Using WebPageTest ‘Fast 3G’ simulated connection (https://www.telegraph.co.uk/gaming/)

The speed index score is high because of a third-party cookie notification banner – we’re going to move that into our core codebase as soon as possible.



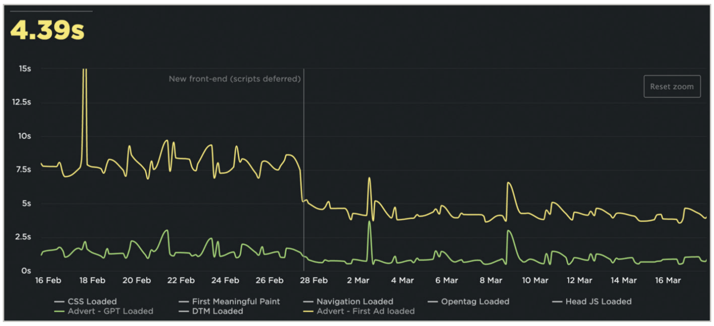
WebPageTest filmstrip showing the cookie notification banner.

Time to interactive (TTI) and first CPU idle measure how soon the page might be interactive to a user. The high values shown above can be attributed to third-party JavaScript. We’re working hard with teams across the organization to educate, help, and question any JavaScript that is executed on site. As a team, we know we’ll get there – it just takes time.



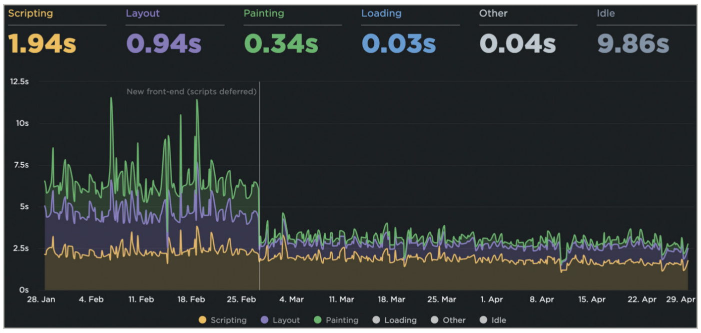
Akamai's mPulse tracking key performance metrics vs. rage clicks on a subset of pages.

We can say with confidence though that deferring our JavaScript hasn't skewed any existing analytics and it certainly hasn't delayed any advertising. By using custom performance marks in the advertising code, deferring JavaScript and reducing bundle sizes, the first ad loaded metric improved by an average of four seconds.



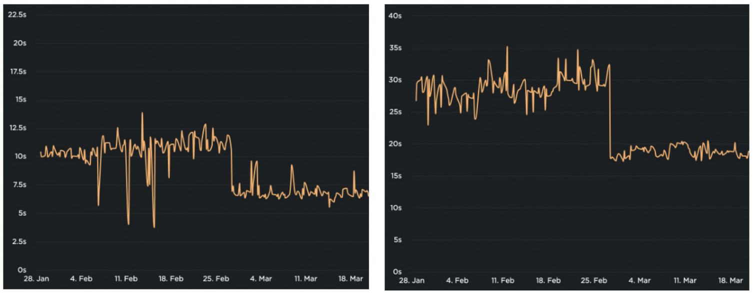
Custom first ad loaded metric improving by an average of four seconds.

Noticeable improvements have also been recorded in other areas, particularly where the CPU spends most of its time.



Over 2-second improvement in layout and painting (https://www.telegraph.co.uk/culture/).

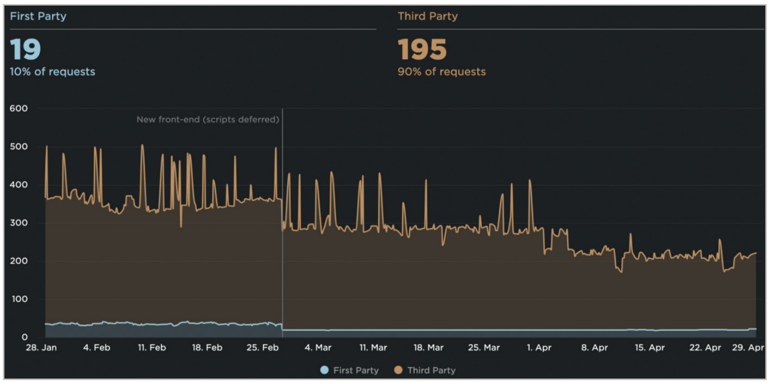
We also saw that by reducing our JavaScript bundle sizes (both first- and third-party), we would end up with a reduction in time to interactive.



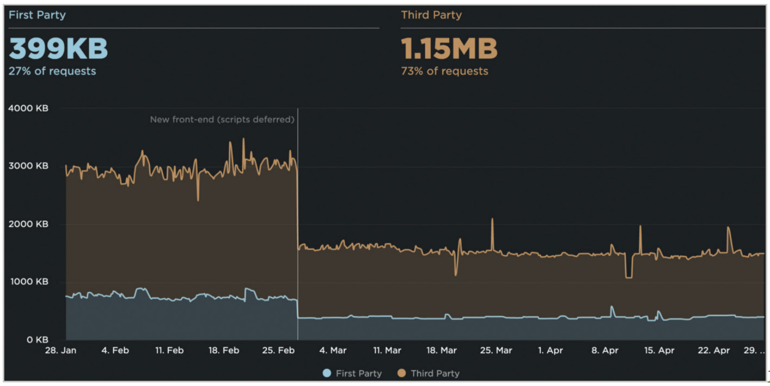
6-second improvement in TTI for Chrome and a 15-second improvement for iPhone 6 (https://www.telegraph.co.uk/culture/).

Regular Audits of Our Tag Managers

Third-party tags can be added by different teams across the organization and they are often forgotten about over time. People move on, contracts expire, or the results are yielded but the teams never get back in touch to have the scripts removed. That's why it's incredibly important to audit any tag manager and to do so often.



Number of third-party requests gradually declining over time from regular audits (https://www.telegraph.co.uk/culture/).



MB reduction in third-party payload (https://www.telegraph.co.uk/culture/).

At the *Telegraph*, we review our tag manager entries at least every quarter. We will reach out to the individual or team that made the original request (for it to be added) to check if they still require it to be present on the website.

When we started this process, we had a collection of very old scripts and couldn't track the original requester. We removed those on the premise that, if they were important, people would get back in touch – no one did.

Testing Each Additional Tag Request

Our main tag manager is currently controlled by our internal technical support team. Teams must send them a written request for the addition of any new script. This made it very easy for us to work with the team to add a performance testing phase to each and every request.

Testing an individual script can be difficult. Scripts have impacts on other scripts, and when you have in excess of 200 third-party requests, the fluctuations in readings can make it challenging to get any meaningful data.

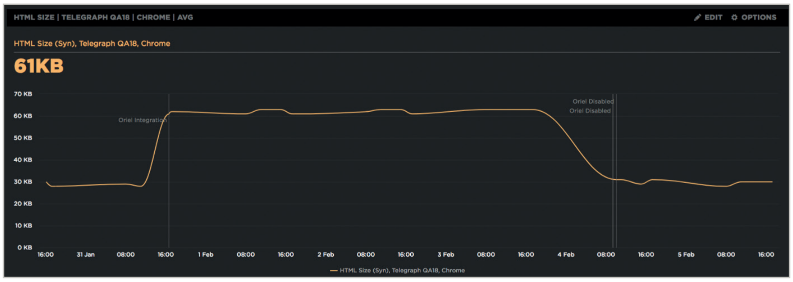
For this reason, we've started testing in isolation. We have a blank page with some dummy text on it and a single, synchronous, tag manager. We would add the third-party script there and run the page through WebPageTest after the initial benchmarking.

Some of the key metrics we would try to monitor:

METRIC	TARGET
Number of requests	< 3
First contentful paint	< 5% increase
Bundle size	< 50 KB
Evaluate events	< 50 ms

These targets aren't set in stone and there is a percentage of give and take. We try to use them as best we can; it really just boils down to being sensible. If there is a large amount of degradation in any of our monitored metrics then it's an obvious rejection with a clear explanation of why.

For larger pieces of work, we would spin them up on a QA environment and let SpeedCurve monitor it over a few days. Some of the results weren't impressive and thus didn't make the cut.

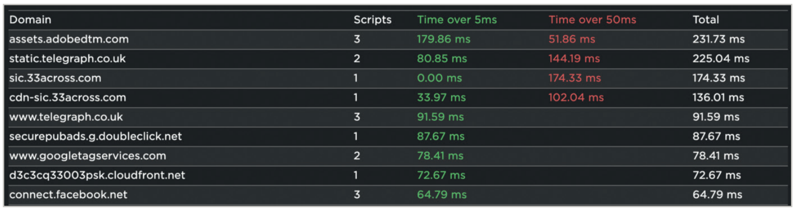


This script was added via ESI and doubled the HTML size from 30 KB to 60 KB.

Monitoring

SYNTHETIC

There are a large number of tools on the market to monitor a website's performance synthetically – many are built around WebPageTest APIs. At the *Telegraph* we use SpeedCurve[5] and have found it invaluable to help pinpoint potential problems along the way. The dashboards they offer around JavaScript usage and third-parties remind us to keep a record of where we've come from and where we need to be.



Domain	Scripts	Time over 5ms	Time over 50ms	Total
assets.adobedtm.com	3	179.86 ms	51.86 ms	231.73 ms
static.telegraph.co.uk	2	80.85 ms	144.19 ms	225.04 ms
sic.33across.com	1	0.00 ms	174.33 ms	174.33 ms
cdn-sic.33across.com	1	33.97 ms	102.04 ms	136.01 ms
www.telegraph.co.uk	3	91.59 ms		91.59 ms
securepubads.g.doubleclick.net	1	87.67 ms		87.67 ms
www.googletagservices.com	2	78.41 ms		78.41 ms
d3c3cq33003psk.cloudfront.net	1	72.67 ms		72.67 ms
connect.facebook.net	3	64.79 ms		64.79 ms

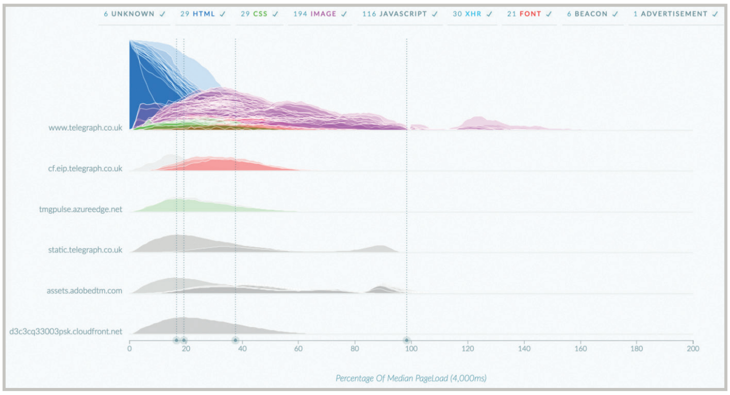
SpeedCurve can easily highlight offending domains that delay the first CPU idle.

REAL USER MONITORING (RUM)

It goes without saying that synthetic monitoring will only get you so far, and that is why we're currently exploring many options when it comes to real user monitoring. We've toyed with the idea of beacon-

5 https://speedcurve.com/

ing and collecting our own data as well as using an external service like Akamai's mPulse or SpeedCurve's LUX.



mPulse dashboard showing at which point resources are called for users during the page's loading process.

METRICS WE MONITOR

- Lighthouse performance, accessibility and SEO scores
- Asset sizes (HTML, CSS, JavaScript, images, number of DOM elements)
- First CPU idle
- Custom metrics such as first ad loaded and our definition of first meaningful paint

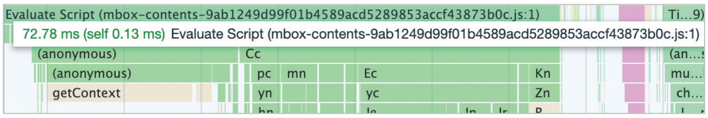
Ideas in the Backlog

BUSINESS METRICS

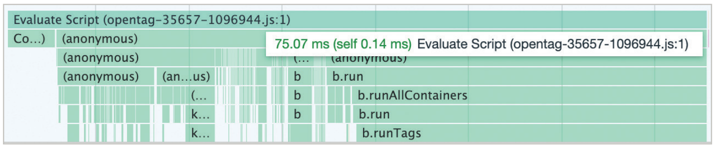
One correlation we haven't made yet is the one between our core business metrics and our web performance improvements. As a team, we would love to tell the business how much it will cost when a third-party request comes in . We're not quite there yet. In hindsight, we would have prioritized this correlation earlier in the process as it can be a powerful tool when negotiating performance improvements with stakeholders and management.

AIM FOR SMALLER BUNDLES AND REDUCE PARSE/COMPILE TIMES

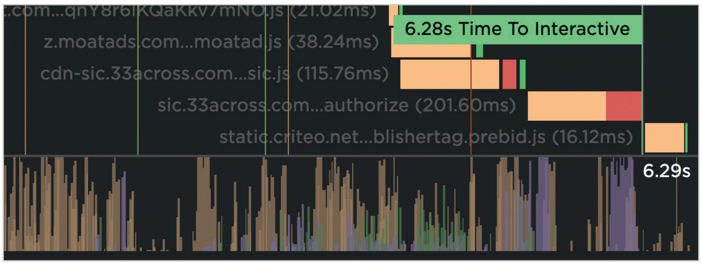
When we've spotted potential problems with third-parties, we always try to get in touch with them and, more often than not, they're willing to listen. We're now getting to the stage where we can start to actively try to pursue smaller bundle sizes and dig deeper into what individual scripts are doing to see if we can offer suggestions on reducing the time it takes to parse and compile their scripts.


Evaluate Script (mbox-contents-9ab1249d99f01b4589acd5289853accf43873b0c.js:1)
Ti…9)
72.78 ms (self 0.13 ms) Evaluate Script (mbox-contents-9ab1249d99f01b4589acd5289853accf43873b0c.js:1)
(anonymous)
Cc
(an…
(anonymous)
pc
mn
Ec
Kn
mu…
getContext
yn
yc
Zn
ch…

Adobe's Target (used for A/B testing) takes ~70 ms to be evaluated.


Evaluate Script (opentag-35657-1096944.js:1)
Co…)
(anonymous)
75.07 ms (self 0.14 ms) Evaluate Script (opentag-35657-1096944.js:1)
(anonymous)
(anonymous)
(an…us)
b
b.run
(…
b
b.runAllContainers
k…
b
b.run
k…
b.runTags

Qubit's Opentag (tag manager) takes ~75 ms to be evaluated.


.com…qnY8r6lKQaKkV7mNO.js (21.02ms)
6.28s Time To Interactive
z.moatads.com…moatad.js (38.24ms)
cdn-sic.33across.com…sic.js (115.76ms)
sic.33across.com…authorize (201.60ms)
static.criteo.net…blishertag.prebid.js (16.12ms)
6.29s

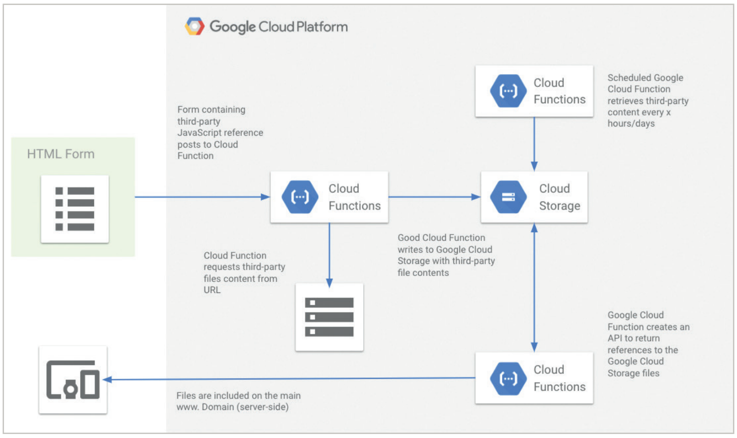
One third-party (native advert supplier) is blocking TTI *by nearly 1,000 ms, 300 ms of which is spent evaluating the two scripts.*

SERVER-SIDE TAG MANAGER OR CDN PROXY

Organizations can often turn to their CDN providers to proxy third-party requests; however, for some this isn't an option. The *Telegraph* is currently investigating a proxy via Akamai but also potentially building a server-side tag manager via serverless architecture.

The general idea here is that an HTML form which contains references to the third-party files would POST to a Cloud function. This Cloud function can download the data from those scripts and save them to Cloud storage.

The references to these files can be included server-side via the main *www* domain. The third-parties would have to be called every *x* hours/days to get relevant updates.


Google Cloud Platform
HTML Form
Form containing third-party JavaScript reference posts to Cloud Function
Cloud Functions
Cloud Storage
Cloud Functions
Scheduled Google Cloud Function retrieves third-party content every x hours/days
Cloud Function requests third-party files content from URL
Good Cloud Function writes to Google Cloud Storage with third-party file contents
Google Cloud Function creates an API to return references to the Google Cloud Storage files
Cloud Functions
Files are included on the main www. Domain (server-side)

Architecture: General > Server-side tag manager.

Benefits:

- Removes need for client-side tag manager (excessive JavaScript)
- Efficient cache policy for repeat visits

- No additional DNS lookups or SSL handshakes
- HTTP/2 multiplexing

By following best practice advice for web performance, involving teams from across the organization, and making our changes visible – the business and, most importantly, our users can see the real benefits. No matter where a person lives, what device or connection they use, we should all have the same access to news and information.

We need to try to be as inclusive as possible – web performance and accessibility is imperative to making that happen. Resolving third-party issues, our biggest performance challenge to date, will help us get there.

Telegraph Key Takeaways

Audit the tag managers regularly to remove unwanted tags.

The *Telegraph* team realized that their performance challenges were mostly organizational. This was especially applicable to their tag managers, where third-party tags were indiscriminately added by different teams over time. The scripts were never removed.

Now the team reviews tag manager entries with representatives from different departments every quarter. This helps to get rid of entries that are no longer required. They also introduced better control over tag managers so the performance impact of every new tag request is tested in isolation before it is included.

Interview

Gareth Clubb

Former *Telegraph* Principal Software Engineer
Author of "Improving Third-Party Performance at the *Telegraph*"

What excited you or your team the most about the work in the case study?

It's safe to say I'm quite passionate about web performance and improving a site's page speed. Why? It comes down to accessibility and inclusivity. I'm a firm believer in the "open web." It's what it was created for – to share information. I always strive to make sure that no matter a user's device, location, or network speed, they should have access to news and information – it's what makes the web great.

The publishing sector often gets a bad reputation for not caring about the end user and only wanting to make money through paywalls and advertising, and I wanted that to change. I joined the industry and that sector in particular to try and make a difference. I joined the *Telegraph* towards the start of 2016, after it had gone through a large replatform from one content management system to another. It was obvious to me that site speed was never really a consideration but was a problem – a news article would take nearly 12.5 seconds to show the user its headline when competitors were doing it in under a second. If loading times are not important to a business during a rebuild and don't make the list of non-functional requirements, you ultimately end up with fantastic tooling and a great developer experience, but you have fallen short in helping the end user. That is why we do what we do – we build fantastic experiences for end users.

Those loading times scared me, but it was also the thing that excited me the most as I knew the smallest changes could have the biggest impact. I was really excited to find out my new team was as passionate as I was about site speed but just hadn't had the opportunity to try and put things right. This meant we could go on this web performance journey together, as trying to tackle something as crucial as loading times within a medium-to-large organization would have been an incredibly daunting task to do alone. We could really demonstrate how important site speed is to the business and sector.

After getting monitoring in place, the biggest challenge we faced – and the one many still face to this day – was the performance of third-party code, particularly third-party JavaScript. We knew that if we could tackle this problem head-on together then we'd better our own website but also many others through working directly with third-party providers on improvements. Also, sharing our knowledge among the web community was an exciting prospect.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

I've changed roles and changed companies since the original case study went live. I've become an engineering lead at PropTech Zoopla and do you know what? We are facing the exact same challenges I went through with the *Telegraph*. I know third-party JavaScript is renowned for tracking users on the web but it was following me around in my real-life career, and that highlights how big of an issue third-party code is for all our websites.

I'm pleased to say that the things my new talented team are trying are the same things I highlighted in the case study. So to answer the question, would I do anything differently? I don't think so. I believe the approach is the same but the order in which they are done could be improved. The very first thing we've done is spin up a web performance working group. This is a group of like-minded individuals who care about performance, want to learn more about performance, and ultimately want to build more performant user experiences. It's a group of software engineers, both front-end and back-end; there are members who care about search engine optimization and those who are responsible for advertising across our products. This group of people can really help instill a performance culture across the different engineering and marketing teams, so this is definitely the first thing I would organize when tackling web performance challenges.

The next most important thing to highlight is monitoring. We did a lot of work at the *Telegraph* to improve site speed but we never really tracked it from the start. This meant we could never demonstrate to the business or any stakeholders the incredible work that we had done. At Zoopla we're trying to learn from those mistakes and gather the important data up front to make more informed decisions. Make sure you have the monitoring in place before you go on this journey, and it needs to be monitoring that non-technical individuals can easily understand, have access to, and share. Whether it's running manual Lighthouse checks in a browser, automating them through CI/CD pipelines, or setting up real user or synthetic monitoring, do something and do it from the start. It's going to make it easier to demonstrate how important site speed is within your organization. One of the hardest parts of web performance is linking site speed metrics to a business conversion rate, but I would thoroughly recommend having a go at it to be able to understand what impact a third-party script or new feature might have on your website.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

We knew as a team that if we wanted to make serious strides in improving the performance of the website then we had to get buy-in from the leadership team, particularly in a medium-large company. It would have been naive of us to assume we could tackle the problems as a small group without proper business backing. This is why monitoring is so important, because we could show the impact the site speed was having for our users, and we could show the gains made from simple improvements. What also helped was competitor analysis as everyone has a competitive streak.

Getting the *Telegraph*'s chief technology officer (CTO), Toby Wright, behind us enabled us to confidently work with different teams when performance was not considered. It enabled us to break down silos across marketing and advertising teams, and it enabled us to roll out a performance-first mindset where we could stop any code releases from happening if there were any degradations in our loading time metrics. We also managed to work closely with our product managers and stakeholders to make sure any site speed-related tasks were prioritized properly and not seen as the small amount of technical debt we should tackle with each iteration – we managed to make web performance the next most important thing for the teams to work on.

One individual can make small technical improvements over a period of time, one cross-functional team can start to make bigger improvements and enable a web performance culture, but ultimately one CTO can help get an entire organization behind your cause.

What came next after the case study was published?

Not all journeys have to come to an end and that's true of the *Telegraph*'s web performance journey, even after I left. I'm in touch with the super smart developers I once worked alongside, and I hear of the challenges they're tackling on a daily basis, and web performance is still not forgotten about. The website now delivers a consistently faster experience, with first paint being under one second, and it's on a par with its competitors in the sector, but there is always room for improvement. The team is now trying to work out what the next most important thing is. Server-side third-party tag management?

Caching third parties on their content delivery network (CDN)? A more performant consent management platform? We will just have to wait and see!

The one thing I am most proud of on my journey, though, is the culture and mindset we've managed to instill across the company and within many individuals. Getting people to care is one of the most important aspects when you want to push for change and make a difference. Still seeing those people care today makes me realize we've done a job to be proud of.

The people I am lucky enough to work with today at Zoopla are now starting out on their journey, and I know through experience that it's going to be challenging, but it's going to be fun and it's going to be incredibly rewarding once we get to where we need to be.

Do you have any advice for teams that would like to follow in your footsteps?

Web performance isn't just a technical problem to solve, it's mostly about people. Find people in your team and your company who care as much as you do. Surround yourself with those people and work together to build a web performance culture and mindset in your peers and your leadership team. Make sure you meet regularly with a structured agenda or problem to solve and invite people from across the organization to discuss the problems they are facing. When the web performance working groups started at the *Telegraph* and Zoopla, respectively, one of the first things we spoke about was trying to define standards, monitoring, performance budgets (the maximum size or number of assets), and working out how to make site speed a non-functional requirement for new features.

From a technical point of view, I've already called out monitoring, but I would work out how best to use the monitoring effectively with performance budgets and alerting. If someone adds a new web font, a large JavaScript bundle, or just hasn't properly optimized a particular image, then it would be ideal if the engineering team knew about this sooner rather than later. Early feedback is very important, so make sure you don't just monitor what's happening in your production or live environments – monitor any development environment and stop that regression from affecting your users.

If you're just starting out on a web performance journey at your company, or maybe you've been at this game as long as I have, my advice would be the same – keep trying, be open to learning new things, and embrace failure, but ultimately just don't give up. Your users and the web are counting on you.

Has the site changed significantly since the case study was published?

The team have done some great work around improving time to first byte (TTFB), largest contentful paint (LCP) and fully loaded metrics. In terms of third parties, it's in a similar place to when we first went live. It's still using a small JavaScript image lazy-loader with the Intersection Observer API, and I'd love to see this now using the `loading` attribute for supported browsers. All of the JavaScript tags are still deferred, helping that first paint, but again: could it use Java-Script modules for browsers that support them? Even conditionally load enhancements as and when they are scrolled into view?

Ultimately, my point here is that yes, the website is as similar as when it first went live. It even won UK News Website of the Year 2019. But web performance and improving site speed has to be a continuous journey: it's not something you do once a year or once a rebuild. Even though broadband speeds and processing power are getting faster, not everyone has that privilege. But everyone has the right to the news and information about the world they're in, so make sure you enable web performance as part of your engineering and organization culture and give everyone that right.

Wix: Trim the Fat from Your Bundles Using Webpack Analyzer and React Lazy/Suspense

by Eyal Eizenberg

As client-side applications become more complex, their bundle sizes become bigger and bigger. Devices and regions with slower connections suffer the most from increasing bundle sizes, and it's just getting worse every day. In this article[1] I will go over a real-world example from my work at Wix where I was able to trim my bundle size by about 80% using Webpack Bundle Analyzer and React Lazy/Suspense.

How Early Should I Optimize?

If you are just getting started with your new and shiny web application, you are probably trying to focus on getting off the ground and making your product come to life. You are probably not focusing on performance or bundle sizes too much. I can relate to this. However, in my experience, this is something you should think about right from the start. Good architecture and trying to think about the future of your app will save you a lot of time and tech debt in the long run. Obviously, it's hard to guess everything ahead of time, but you should try to do your best.

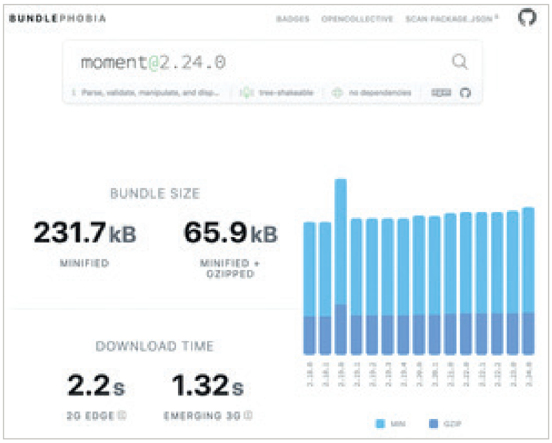
There are two great tools which I think you should use right from the start. These tools will help you recognize problematic NPM packages even before you rely on them in your app.

1 This original version of this case study was published in September 2019: https://smashed.by/wixengineering

BUNDLEPHOBIA

Bundlephobia shows you how much an NPM package will increase your bundle size.[2] This is a great tool which might help you make better choices with regards to picking a third-party package you might need, or how to design your architecture so your app doesn't become bloated. In the screenshot below, I checked the popular time-parsing library "moment." You can see that it's big. Almost 66 KB gzipped.

For a lot of people with blazing internet speeds, it's nothing. However, look how long the download time increases for 2G/3G networks: 2.2 s and 1.32 s respectively, and that's just for one package.



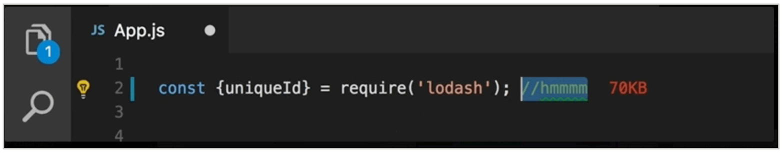
Bundlephobia's result for the "moment" package.

IMPORT COST EXTENSION

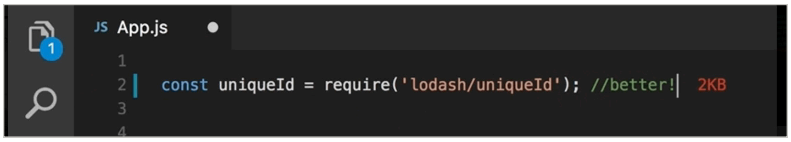
This is a very cool extension for various popular editors (1 million+ downloads for VS Code), which shows you how much importing a package will cost.[3] What I really like about this extension is that it helps identify specific problematic areas on the fly. This following images (taken from Import Cost's GitHub page) shows a perfect example of how importing the `uniqueId` property from Lodash brings in the entire Lodash package (70 KB) as opposed to importing just

2 https://bundlephobia.com/
3 https://smashed.by/vscodeimportcost

the `uniqueId` function directly, which adds just 2 KB. The article "Get Slim Bundles with the Import Cost Extension" by Yair Haimovitch has more information about Import Cost.[4]



```
const {uniqueId} = require('lodash'); //hmmmm  70KB
```



```
const uniqueId = require('lodash/uniqueId'); //better!  2KB
```

Cost of importing all of Lodash vs just a specific function.

Bloated Bundles: A Case Study

So you've built your amazing app. It works great on your high-speed internet connection and your superpowered, ultra-fast with extra RAM dev computer. Then after a little while, you start getting complaints from your users or from your analytics team that your app's load time is not so great. This recently happened to me after we released a new feature I was working on here at Wix.

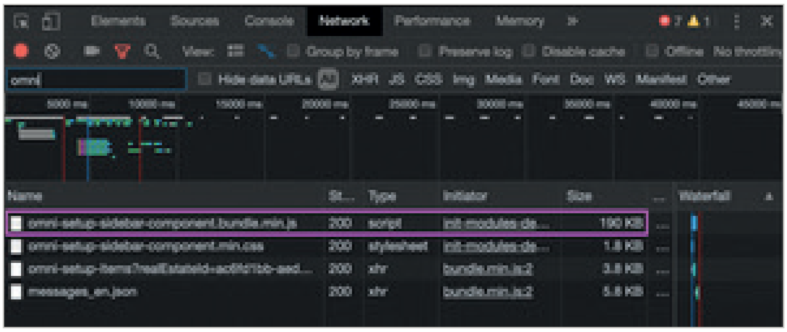
To give you some perspective, let's first look at the new feature. The feature is a new progress bar at the top of your sidebar. The goal is to expose various steps you should take in order to have a better chance of succeeding with your business (connect SEO, add shipping regions, add your first product, etc).

The progress bar updates automatically by connecting to the server via websockets. When the user completes all the recommended steps, a tooltip with a "happy moment" is shown in order to cele-

4 https://smashed.by/gitimportcost

brate your achievement. After the "happy moment" is closed, the progress bar is hidden and will never be shown again for this site.

So what was happening? Why was I getting complaints from our analytics team saying that the load time for the page has increased? Looking at the **Network** tab in Chrome's DevTools, it quickly became apparent that my bundle was big: 190 KB big.



Name	St...	Type	Initiator	Size
omni-setup-sidebar-component.bundle.min.js	200	script	init-modules-de...	190 KB
omni-setup-sidebar-component.min.css	200	stylesheet	init-modules-de...	1.8 KB
omni-setup-items?realEstateId=ac6fd1bb-aed...	200	xhr	bundle.min.js:2	3.8 KB
messages_en.json	200	xhr	bundle.min.js:2	5.8 KB

My bundle size according to Chrome's DevTools.

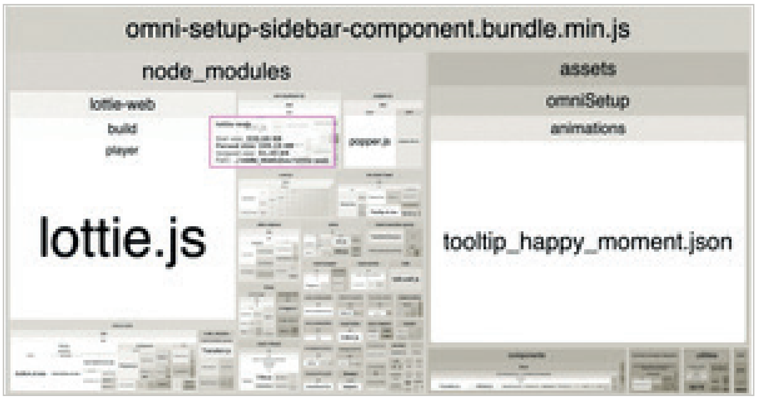
I thought to myself, why should this small feature have such a (relatively) big bundle?! Why indeed…

FINDING THE PROBLEMATIC AREAS IN YOUR BUNDLE

After realizing the bundle size was too big, it was time to find out why. A great tool to help find problematic areas in your bundle is Webpack Bundle Analyzer.[5] This tool will open a new tab in your browser and it will visualize all of your dependencies.

Using the analyzer, I was able to find the culprit. I was using lottie-web, which added 61.45 KB to my bundle. Lottie is a really cool JavaScript library that renders After Effects animations natively. In my specific app, our designer/animator wanted a nice animation when the "happy moment" appeared. He designed it and gave me a JSON

5 https://smashed.by/slimbundles



Result of Webpack Bundle Analyzer.

file, which I passed to the Lottie package and voilà – a great-looking animation appeared before my eyes. In addition to the lottie-web package, the JSON file I had to pass to Lottie was 26 KB. So Lottie + the JSON file + additional small dependencies was costing me about 94 KB. Just for that animation in the "happy moment". This for me, was actually a sad moment...

REACT LAZY/SUSPENSE TO THE RESCUE

After I dusted myself off, it was time to fix the problem. It was obvious that there is no need to bring in everything that was needed for the animation right from the start. In fact, there was even a very good chance that the "happy moment" won't be shown at all during the current user's session. I read up on React Lazy/Suspense which came out recently, and I thought that this might be a great chance to test it out.

If you are not familiar with the concept of lazy components, the idea is that you split your app into smaller pieces and then fetch the relevant pieces only when you need them. So in my case, I wanted to break apart the component that was responsible for rendering the "happy moment" and fetch it only when the user completes all the recommended steps.

React 16.6.0 (or higher) provides a simple API which helps render lazy components called `React.lazy` and `React.Suspense`.[6] Let's look at this simple example:

```
const OtherComponent = React.lazy(() => import('./
OtherComponent'));

function MyComponent() {
 return (
   <div>
     <React.Suspense fallback={<div>Loading...</div>}>
       <OtherComponent />
     </React.Suspense>
   </div>
 );
}
```

We have a component here which renders a `div` and in it the `Suspense` component, which wraps the `OtherComponent`. If you look at line 1 you will see that `OtherComponent` is not brought directly. Usually it will look like this: `import OtherComponent from ‘./OtherComponent’;`

Instead, the `import` command is used as a function which receives the path of the file. This works because Webpack has built-in code splitting, and when used in this specific way returns a promise which will resolve with the content of the file once it's fetched. This import is wrapped in the `React.lazy` function.

In our render function of `MyComponent` the `OtherComponent` is wrapped in `React.Suspense` which has a prop called `fallback`. This means that only when the render function "gets" to the `OtherComponent` (line 7) it will begin fetching it. In the meantime, it will render whatever is rendered in the `fallback` prop: in this example, a `div` with the text "Loading..." That's it. It just works...

6 https://reactjs.org/docs/code-splitting.html

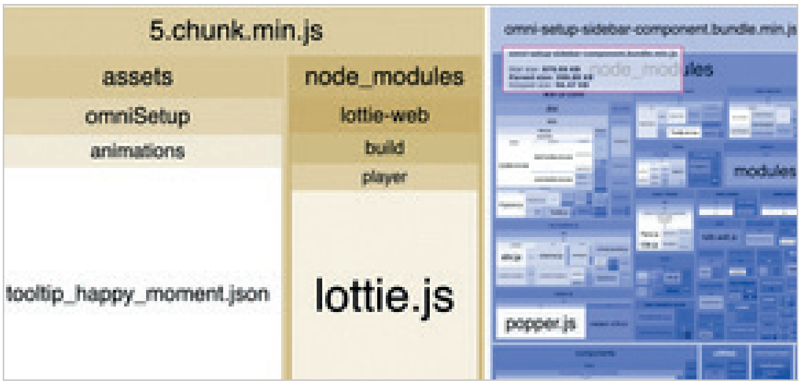
There are two gotchas you should take into consideration:

1. The component which is brought in lazily has to have a default export and that will be the entry point of your component. You can't use a named export.

2. You have to wrap the `React.lazy` component with the `React.Suspense` component, and you have to provide it with the `fallback` prop, otherwise an error will be thrown. But don't worry: in case you don't want to render anything until the lazy component arrives, you can just pass `null` as the `fallback` prop.

DID IT WORK OUT FOR ME?

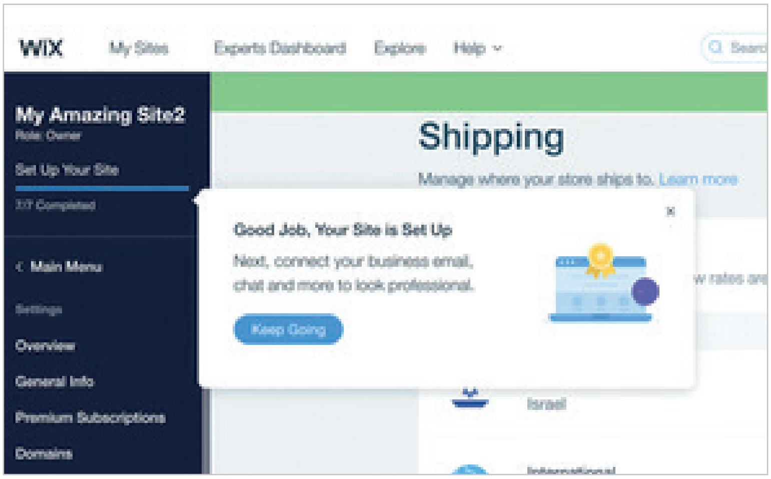
It did! Well, kind of… The part that worked marvelously was the code splitting. Let's look at the Webpack analysis after splitting the code:



Result of Webpack Bundle Analyzer after splitting.

As you can see in the image above, my bundle has been cut down by about 50% to 96 KB. Yay!

So what didn't work? The positioning of my tooltip was now off:



Misplaced tooltip.

The problem was that I told the tooltip to open by setting a state in the React component. In the meantime, I rendered null (nothing) using the `React.Suspense` component. Once the content arrived lazily, it was rendered into the DOM. However, the positioning of the tooltip was already done beforehand, and because the props of the tooltip component did not change, it didn't know that it needed to check if it needed to reposition the content. If I changed Chrome's window size, the tooltip popped into the right position because the tooltip was listening to prop changes and window resizes in order to initiate repositioning.

So what was the solution here? Cut out the middleman.

I needed to first fetch the lazy component and only then set the state which told the tooltip to open. I was able to do this by using the same Webpack code-splitting ability but without wrapping it in `React.lazy`:

```
async loadAndSetHappyMoment() {
const component = await import(
  '../SidebarHappyMoment/SidebarHappyMoment.component'
);
this.SidebarHappyMoment = component.SidebarHappyMoment;
this.setState({
  tooltipLevel: TooltipLevel.happyMoment,
});
}
```

This function is called after my component gets triggered via websockets that it needs to show the “happy moment”. I am using Webpack’s `import` function (lines 2–4). If you remember what I wrote earlier, it returns a promise so I can use the `async/await` syntax.

Once the component arrives, I am setting it to the instance of my component (line 5) so I will be able to use it in the render function later. Notice also how I can use named exports now. I am using the one called `SidebarHappyMoment` (line 5). Last but not least, I am telling the tooltip to open by setting the state after I know my component is ready (lines 6–8).

My render function now looks like this:

```
renderTooltip() {
      if (this.state.tooltipLevel === TooltipLevel.happyMoment)
{
         return <this.SidebarHappyMoment />;
       }
      //  ...
      }
```

Notice how in line 3 I am rendering `this.SidebarHappyMoment` which I’ve set on my instance earlier. This is now a normal synchronous render function like you’ve used a million times before.

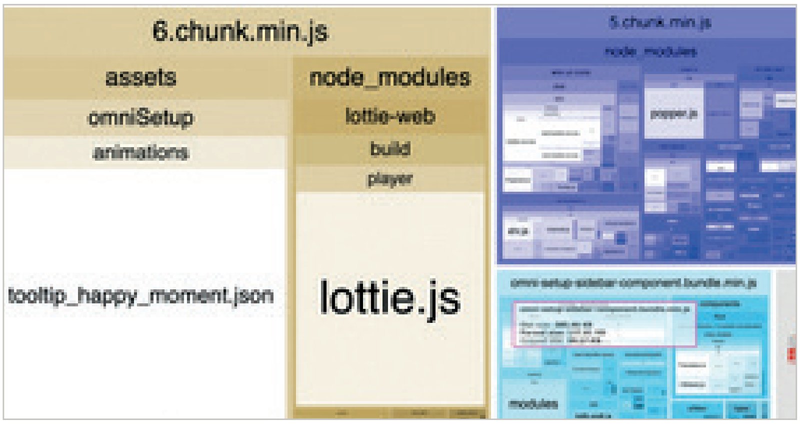
And now, my “happy moment” tooltip rendered exactly where it should have because the tooltip was opened only after its content was ready.

THE PRODUCT DEFINES THE ARCHITECTURE

Wait, what?! Yes, exactly!

The product defines what needs to be visible and interactive when the component first renders. This will help you as a developer figure out what you can break apart and bring in later as needed. I gave my specific use case more thought and remembered that once the user completes the setup steps or if they are not the site’s admin, we don’t want to render the progress bar at all. Using this information, I was able to split my bundle even more, and now it looks like this:

So what was the solution here? Cut out the middleman.



Three-way split of the bundle.

As you can see, the bundle size is now only 38 KB. Remember we started with 190 KB? An 80% reduction. I have already recognized more things I can extract, and I am eager to trim the bundle even more.

Conclusion

Developers tend to stay in their comfort zone and not delve beyond the code and its functionality. However, using these tools, some creative thinking, and working closely with your product manager, you could probably enhance your app's performance by making your bundle size much smaller.

Wix Key Takeaways

Trim bloated bundles and redesign to meet requirements with improved speed.

When the introduction of a new feature led to degradation in performance, Wix engineer Eyal Eizenberg was able to pinpoint the root cause of the issue using the webpack bundle analyzer. He identified that the use of a third-party library introduced to implement the new feature added significantly to the bundle size.

The new feature was a "happy moment" tooltip that was only shown to users when they completed all steps in the set-up process. Since it was not required for all users, through intelligent code-splitting combined with the use of React Lazy/Suspense, he was able to significantly reduce the bundle size from 190 KB to 38 KB and improve the load time.

Interview

Eyal Eizenberg

Head of R&D, Dashboard Group, Wix
Author of "Wix: Trimming the Fat From JS Bundles With Webpack Bundle Analyzer"

What excited you or your team the most about the work in the case study?

My team's main focus is the dashboard at Wix. This page has a lot of traffic, and performance is always on our minds. My area of expertise is front-end development, and when it comes to performance I love trying to find new ways of trimming bundle sizes. I use Webpack Bundle Analyzer all the time to make sure everything is bundled correctly and I am not bundling things I don't really need. I analyzed this specific project and noticed there was a problem. Right around that time, React Suspense was rolling out. I was excited to see how I could use React Suspense and webpack lazy loading to fix the problem I was facing.

What I really loved about this case study was that usually performance issues are solved solely by developers. However, in this case the solution came by working closely with the product manager and realizing we could lazy-load parts of the application and trim the bundle size dramatically.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

I was very surprised with the results. Usually when analyzing big JavaScript bundles you are able to make things better by shaving a few KBs here and there, you switch a third-party library for a smaller one, or by realizing a project was bundling too many things erroneously. In this case, not only was I able to reduce the bundle size by 80%, but I was also able to do it without the users noticing anything, or replacing libraries. It was just by some creative thinking and collaborating with the product manager to better understand the product.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

The biggest part of the bundle was Lottie and the JSON provided to it. I am much more aware of it now, and if it is required in a project I apply the methodologies I learned in this case study. However, since there are many other big libraries out there, I try to use tools such as Import Cost and Bundlephobia before bringing in a new npm package. And lastly, I work very closely with product managers in order to really understand the applications I need to develop before diving in so I could recognize potential optimizations.

In this case, not only was I able to reduce the bundle size by 80%, but I was also able to do it without the users noticing anything, or replacing libraries.

What came next after the case study was published?

The case study had a big ripple effect with many developers reaching out to discuss it, blog posts, several related talks at international conferences, and this book.

Do you have any advice for teams that would like to follow in your footsteps?

Developers tend to work with the best internet connection available. However, most of the time our customers don't have that, whether it be developing countries, out-of-reach geographical locations, or just someone using their phone while commuting to work on a train. Try slowing down your connection in Chrome's DevTools and ask yourself: is this acceptable to me? If the answer is no, then you have to step up your performance game.

Work with your product managers and try to see how you can break apart your application so the crucial things load first and quickly, and the less important ones load in the background or if possible not at all.

Has the site changed significantly since the case study was published?

We are now in the process of writing the third version of the dashboard. The motivation for that is not bad performance but, rather, deep research and the will to understand our customers better, provide them with the best experience and the most relevant information for their specific needs. Naturally, as Wix is growing so rapidly, making sure our performance is top-notch is crucial.

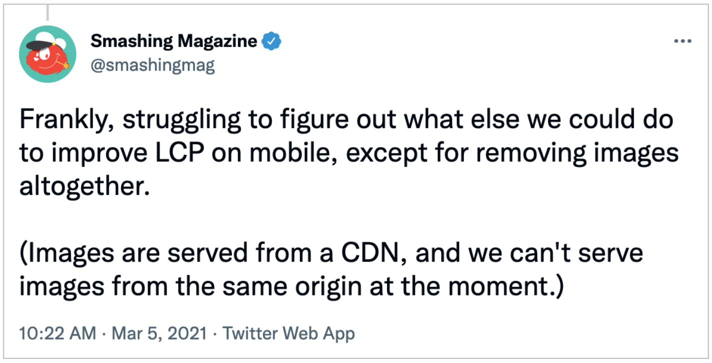
Improving Core Web Vitals: A Smashing Magazine Case Study

by Barry Pollard

Why are my core web vitals failing?" Many developers have been asking themselves that question lately.[1] Sometimes it's easy enough to find the answer to that question and the site just needs to invest in performance. Sometimes though, it's a little trickier and, despite thinking your site was great on the performance for some reason it still fails. That's what happened to our very own *smashingmagazine.com* and figuring out and fixing the issue took a bit of digging.

A Cry For Help

It all started with a series of tweets in March 2021 with a cry for help:


Smashing Magazine
@smashingmag

Frankly, struggling to figure out what else we could do to improve LCP on mobile, except for removing images altogether.

(Images are served from a CDN, and we can't serve images from the same origin at the moment.)

10:22 AM · Mar 5, 2021 · Twitter Web App

Smashing Magazine's tweet asking for help.

Well, this piqued my interest! I'm a big fan of Smashing Magazine and am very interested in web performance and the core web vitals. I've

1 The original version of this case study was published in December 2021: https://smashed.by/corewebvitalscasestudy

written a few articles here before on core web vitals, and am always interested to see what's in their annual Web Performance Checklist.[2] So, Smashing Magazine knows about web performance, and if they were struggling, then this could be an interesting test case to look at!

A few of us made some suggestions on that thread as to what the problem might be after using some of our favorite web performance analysis tools like WebPageTest or PageSpeed Insights.

Investigating the LCP Issue

The issue was that LCP was too slow on mobile. LCP, or largest contentful paint, is one of the three core web vitals that you must "pass" to get the full search ranking boost from Google as part of their Page Experience Update.[3] As its name suggests, LCP aims to measure when the largest content of the page is drawn (or "painted") to the screen. Often this is the hero image or the title text. It is intended to measure what the site visitor likely came here to see.

Previous metrics measured variations of the *first paint* to screen (often this was a header or background color): incidental content that isn't really what the user actually wants to get out of the page. The largest content is often a good indicator of what's most important. And the "contentful" part of the name shows this metric is intended to ignore (e.g. background colors); they might represent the largest content, but they are not "contentful" so they don't count towards LCP – instead, the algorithm tries to find something more relevant.

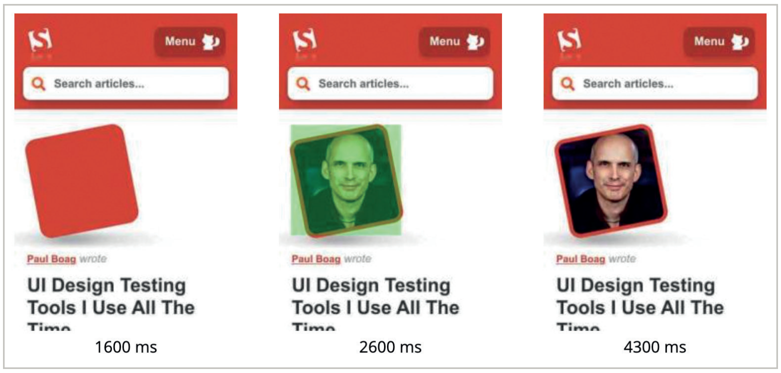
LCP only looks at the initial viewport. As soon as you scroll down or otherwise interact with the page the LCP element is fixed and we can calculate how long it took to draw that element from when the page first started loading – and that's your LCP!

2 https://smashed.by/performancechecklist
3 https://smashed.by/pageux

There are many ways of measuring your core web vitals,[4] but the definitive way – even if it's not the best way, as we'll see soon – is in Google Search Console (GSC).[5] From an SEO perspective, it doesn't really matter what other tools tell you: GSC is what Google Search sees. Of course, what matters is your users' experience rather than what some search engine crawler sees, but one of the great things about the core web vitals initiative is that it measures real user experience rather than what Google Bot sees! So, if GSC says you have bad experiences, then you have bad experiences according to your users.

Search Console told Smashing Magazine that its LCP on mobile for most pages needed improving – a standard enough output of that part of GSC and pretty easily addressed: just make your LCP element draw faster! This shouldn't take too long. Certainly not six months (or so we thought). So, first up is finding out what the LCP element is.

Running a failing article page through WebPageTest highlighted the LCP element:



The LCP image of a typical Smashing Magazine article.

4 https://smashed.by/corewebvitalsguide
5 https://smashed.by/gsc

Improving The LCP Image

OK, so the article author photo is the LCP element. The first instinct is to ask what we could do to make that faster. This involves delving into waterfalls, seeing when the image is requested, how long it takes to download, and then deciding how to optimize that. Here, the image was well optimized in terms of size and format (usually the first and easiest option for improving the performance of images!). The image was served from a separate assets domain (often bad for performance), but it wasn't going to be possible to change that in the short term, and Smashing Magazine had already added a `preconnect` resource hint to speed that up as best they could.

As I mentioned before, Smashing Magazine knows about web performance, had only recently worked on improving their performance,[6] and had done everything right here but was still failing. Interesting…

Other suggestions rolled in, including reducing load, delaying the service worker (to avoid contention), or investigating HTTP/2 priorities, but they didn't have the necessary impact on the LCP timing. So we had to reach into our web performance toolbag for all the tips and tricks to see what else we could do here.

If a resource is critical to the page load, you can inline it so it's included in the HTML itself. That way, the page includes everything necessary to do the initial paint without delays. For example, Smashing Magazine already inlined critical CSS to allow a quick first paint but did not inline the author's image. Inlining is a double-edged sword and must be used with caution. It beefs up the page and means subsequent page views do not benefit from the fact that data is already downloaded. I'm not a fan of over-inlining[7] because of this.

6 https://smashed.by/smashingmagperformance
7 https://smashed.by/inliningcss

So, it's not normally recommended to inline images. However, here the image was causing us real problems, was reasonably small, and was directly linked to the page. Yes, if you read a lot of articles by that one author it's a waste to redownload the same image multiple times instead of downloading the author's image once and reusing it, but in all likelihood, you're here to read different articles by different authors.

There have been a few advances in image formats recently, but AVIF is causing a stir as it's here already (at least in Chrome and Firefox), and it has impressive compression results over the old JPEG formats traditionally used for photographs. Vitaly didn't want to inline the JPEG version of the author images, but investigated whether inlining the AVIF version would work. Compressing the author image using AVIF, and then base64-ing the image into the HTML led to a 3 KB increase to the HTML page weight — which is tiny and so was acceptable.

Since AVIF was only supported in Chrome at the time (it came to Firefox after all this), and since Core Web Vitals is a Google initiative, it did feel slightly "icky" optimizing for a Google browser because of a Google edict. Chrome is often at the forefront of new feature support and that's to be commended, but it always feels a little off when those two sides of its business impact each other. Still, this was a new standard image format rather than some proprietary Chrome-only format (even if it was only supported in Chrome initially), and was a progressive enhancement for performance (Safari users still get the same content, just not quite as fast), so with the addition of the AVIF twist, Smashing took the suggestion and inlined the image and did indeed see impressive results in lab tools. Problem solved!

An Alternative LCP

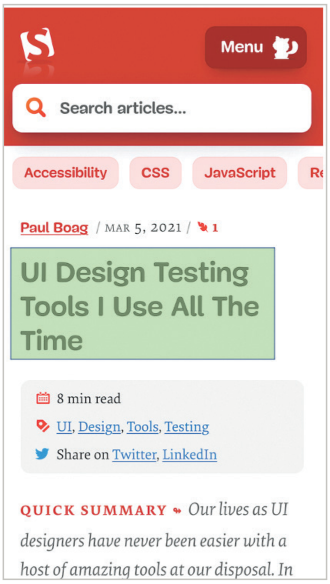
So, we let that bed in and waited the usual 28 days or so[8] for the Core Web Vitals numbers to all turn green… but they didn't. They flitted between green and amber so we'd certainly improved things, but hadn't solved the issue completely. After staying a long stretch in the amber "needs improvement" section, Vitaly reached out to see if there were any other ideas.

The image was drawing quickly. Not quite instantly (it still takes time to process an image after all) but as near as it could be. To be honest, I was running out of ideas but took another look with fresh eyes. And then an alternative idea struck me — were we optimizing the *right* LCP element? Authors are important of course, but is that really what the reader came here to see? Much as our egos would like to say yes, and that our beautiful shining mugs are much more important than the content we write, the readers probably don't think that (readers, huh — what can you do!).

The reader came for the article, not the author. So the LCP element should reflect that, which might also solve the LCP image drawing issue. To do that we just put the headline above the author image, and increased the font size on mobile a bit.
This may sound like a sneaky trick to fool the core web vital SEO gods at the expense of the users, but in this case, it helps both! Although many sites do try to go for the quick and easy hack or optimize for GoogleBot over real users, this was not a case of that and we were quite comfortable with the decision here. In fact, further tweaks removed the author image completely on mobile where there's limited space, and that article currently looks like this on mobile, with the LCP element highlighted:

To improve the metrics, you have to improve the experience.

8 https://smashed.by/cruxreport


Menu

Search articles...

Accessibility CSS JavaScript Re

Paul Boag / MAR 5, 2021 / 1

UI Design Testing Tools I Use All The Time

8 min read

UI, Design, Tools, Testing

Share on Twitter, LinkedIn

QUICK SUMMARY ↬ *Our lives as UI designers have never been easier with a host of amazing tools at our disposal. In*

Smashing Magazine article without author image and with the title highlighted as LCP element.

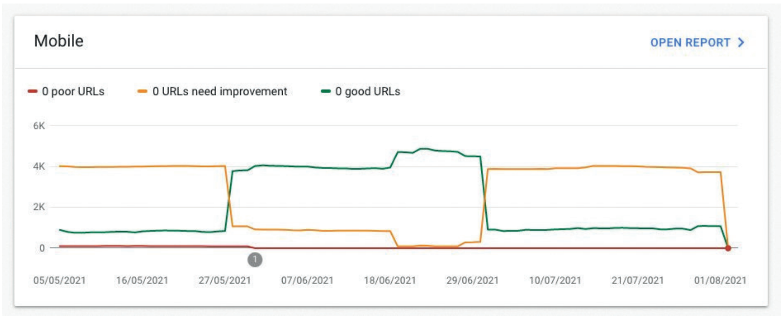
Here we show the title, the key information about the article, and the start of the summary — much more useful to the user than taking up all the precious mobile screen space with a big photo!

And that's one of the main things I like about the core web vitals: they are measuring user experience.

And NOW we were finally done. Text draws much quicker than images so that should sort out the LCP issue. Thank you all very much and good night!

I Hate That CWV Graph in Google Search Console…

Again we were disappointed. That didn't solve the issue and it wasn't long before the Google Search Console graph returned to amber:



Core web vitals graph from Google Search Console.

At this point, we should talk a little more about page groupings and core web vitals. You might have noticed from the above graph that pretty much the whole graph swings at once. But there was also a core group of about 1,000 pages that stayed green most of the time. Why is that?

Well, Google Search Console categorizes pages into page groupings and measures the Core Web Vitals metrics of those page groupings. This is an attempt to fill in missing data for those pages that don't get enough traffic to have meaningful user experience data. There's a number of ways that Google could have tackled this: they could have just not given any ranking boost to such pages, or maybe assumed the best and given a full boost to pages without any data. Or they could have fallen back to origin-level Core Web Vitals data. Instead, they tried to do something cleverer, which was an attempt to be helpful but is in many ways also more confusing: page groupings.

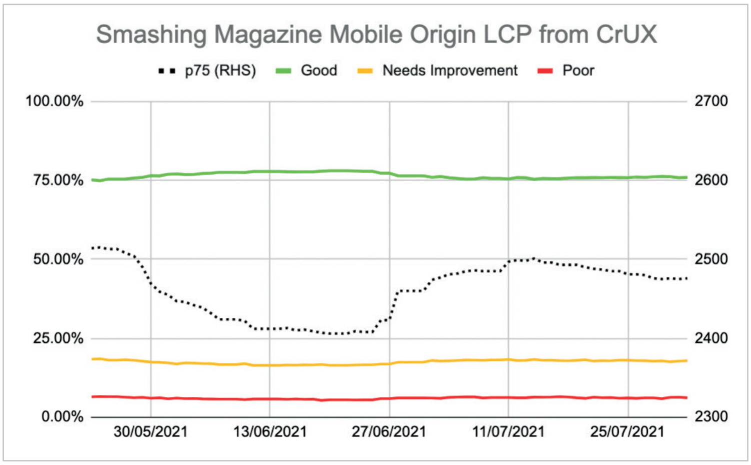
Basically, every page is assigned a page grouping. How Google does this isn't made clear, but URLs and technologies used on the page have been mentioned before. You also can't see what groupings Google has chosen for each of your pages, or if their algorithm got it right, which is another frustrating thing for website owners, though they do give sample URLs for each different core web vitals score below the graph in Google Search Console from which the grouping can sometimes be implied.

Page groupings can work well for sites like Smashing Magazine. For other sites, page groupings may be less clear, and many sites may just have one grouping. The Smashing site, however, has several different types of pages: articles, author pages, guides, and so on. If an article page is slow because the author image is the LCP image, and it is slow to load, then that will likely be the case for *all* article pages. And the fix will likely be the same for *all* article pages. So grouping them together there makes sense (assuming Google can accurately figure out the page groupings).

However, where it can get confusing is when a page does get enough visitors to get its own core web vitals score and it passes, but it's lumped in with a failing group. You can call the CrUX API for all the pages in your site, see most of them are passing, then be confused when those same pages are shown as failing in Search Console because they've been lumped in a group with failing URLs and most of the traffic for that group is for failing. I still wonder if Search Console should use page-level Core Web Vitals data when available, rather than always using the grouping data.

Anyway, that accounts for the large swings. Basically, all the articles (of which there are about 3,000) appear to be in the same page grouping (not unreasonably!) and that page grouping is either passing or failing. When it switches, the graph moves dramatically.

You can also get more detailed data on the core web vitals through the CrUX API. This is available at an origin-level (i.e. for the whole site), or for individual URLs (where enough data exists), but annoyingly not at the page grouping level. I'd been tracking the origin level LCP using the CrUX API to get a more precise measure of the LCP and it showed a depressing story too:



Tracking Smashing Magazine mobile origin LCP from CrUX.

We can see we've never really "solved" the issue and the amount of "Good" pages (the green line above) still hovered too close to the 75% pass rate. Additionally, the p75 LCP score (the dotted line which uses the right-hand axis) never really moved far enough away from the 2500 milliseconds threshold. It was no wonder the pages passing and failing were flipping back and forth. A bit of a bad day, with a few more slow page loads, was enough to flip the whole page grouping into the "needs improvement" category. We needed something more to give us some headroom to be able to absorb these "bad days".

At this point, it was tempting to optimize further. We know the article title was the LCP element so what could we do to further improve that? Well, it uses a font, and fonts have always been a bane of web performance.

But hold up a minute. Smashing Magazine *was* a fast site. Web performance tools like Lighthouse and WebPageTest showed that — even on slower network speeds. And it was doing everything right! It was built as a static site generator so didn't require any server-side generation to occur; it inlined everything for the initial paint so there were no resource loading constraints other than the HTML itself; it was hosted by Netlify on a CDN so should be near its users.

Sure, we could look at removing the font, but if Smashing Magazine couldn't deliver a fast experience given all that, then how could anyone else? Passing core web vitals shouldn't be impossible, nor require you to only be on a plain site with no fonts or images. Something else was up here and it was time to find out a bit more about what was going on instead of just blindly attempting another round of optimizations.

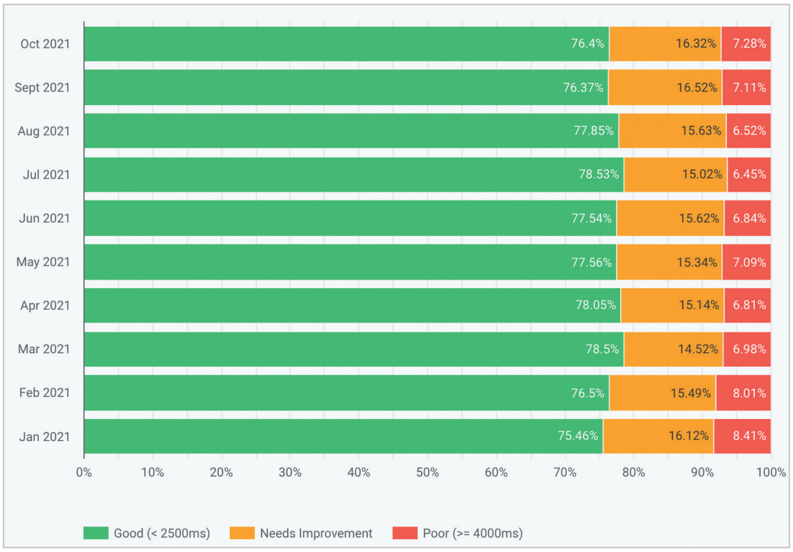
Digging a Little Deeper into the Metrics

Smashing Magazine didn't have a RUM solution, so instead we delved into the Chrome User Experience Report (CrUX) data that Google collects for the top 8 million or so websites and then makes available to query in various forms. It's this CrUX data that drives the Google Search Console data and, ultimately, the ranking impact. We'd already been using the CrUX API but decided to delve into other CrUX resources.

We used the sitemap and a Google Sheets script to look at all the CrUX data for the whole site where it was available (Fabian Krumbholz has since created a much more comprehensive tool[9] to make this easier!) and it showed mixed results for pages. Some pages passed, while others, particularly older pages, were failing.

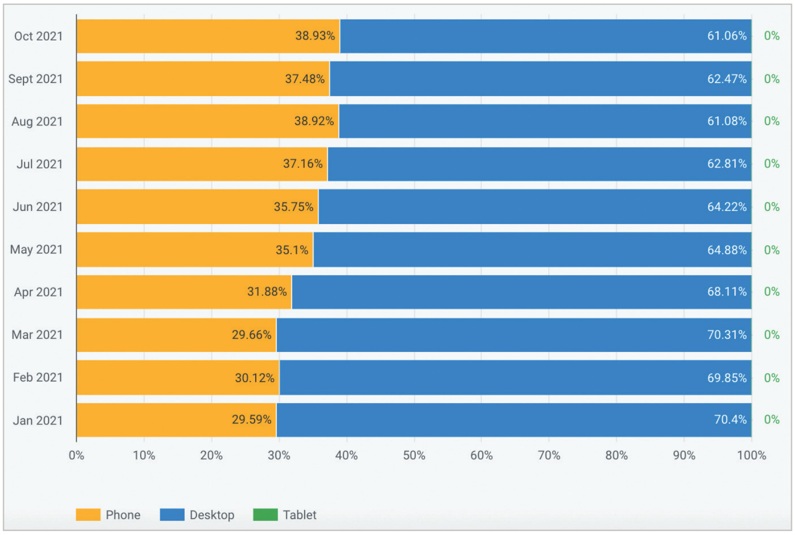
The CrUX dashboard didn't really tell us much that we didn't already know in this instance: the LCP was borderline, and unfortunately not trending down:

9 https://smashed.by/webvitalsoptimizer



CrUX Dashboard LCP trend for SmashingMagazine.com.

Digging into the other stats (TTFB, First Paint, Online, DOMContentLoaded) didn't give us any hints. There was, however, a noticeable increase in mobile usage:



CrUX Dashboard device trend for SmashingMagazine.com.

Was this part of a general trend in mobile adoption? Could that be what was affecting the mobile LCP despite the improvements we'd done? We had questions but no answers or solutions.

One thing I wanted to look at was the global distribution of the traffic. We'd noticed in Google Analytics a lot of traffic from India to old articles — could that be an issue?

The India Connection

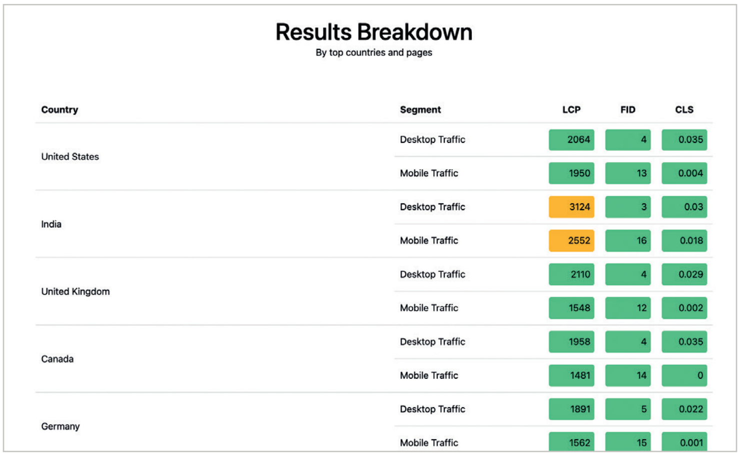
Country-level CrUX data isn't available in the CrUX dashboard but is available in the BigQuery CrUX dataset, and running a query in there at the *smashingmagazine.com* origin level[10] shows a wide disparity in LCP values (the SQL is included on the second tab of that link, in case you want to try the same thing on your own domain). Based on the top 10 countries in Google Analytics we have the following data:

COUNTRY	MOBILE P75 LCP VALUE	% OF TRAFFIC
United States	88.34%	23%
India	74.48%	16%
United Kingdom	92.07%	6%
Canada	93.75%	4%
Germany	93.01%	3%
Philippines	57.21%	3%
Australia	85.88%	3%
France	88.53%	2%
Pakistan	56.32%	2%
Russia	77.27%	2%

10 https://smashed.by/smashingmagcrux

India traffic is a big proportion for Smashing Magazine (16%) and it is not meeting the target for LCP at an origin level. That could be the problem and certainly was worth investigating further. There was also the Philippines and Pakistan data with very bad scores, but that was a relatively small amount of traffic.

At this point, I had an inkling what might be going on here – and a potential solution – so got Smashing Magazine to install the `web-vitals`[11] library to collect RUM data and post it back to Google Analytics for analysis. After a few days of collecting, we used the Web Vitals Report[12] to give us a look at the data in ways we hadn't been able to see before, in particular the country-level breakdown:


Results Breakdown

By top countries and pages

Country	Segment	LCP	FID	CLS
United States	Desktop Traffic	2064	4	0.035
	Mobile Traffic	1950	13	0.004
India	Desktop Traffic	3124	3	0.03
	Mobile Traffic	2552	16	0.018
United Kingdom	Desktop Traffic	2110	4	0.029
	Mobile Traffic	1548	12	0.002
Canada	Desktop Traffic	1958	4	0.035
	Mobile Traffic	1481	14	0
Germany	Desktop Traffic	1891	5	0.022
	Mobile Traffic	1562	15	0.001

Web Vitals Report for smashingmagazine.com broken down by country.

And there it was. All the top countries in the analytics did have very good LCP scores, except one: India. Smashing Magazine uses Netlify, which is a global CDN, and it does have a Mumbai presence,[13] so it should be as performant as other countries, but some countries are just slower than others (more on this later).

11 https://smashed.by/webvitalslibrary
12 https://smashed.by/webvitalsreport
13 https://smashed.by/cdnpops

However, the mobile traffic for India was only just outside the 2,500 limit, and it was only the second most visited country. Surely the good USA scores should have been enough to offset that? Well, the above two graphs show the countries order by traffic. But CrUX counts mobile and desktop traffic separately (and tablet btw, but no one ever seems to care about that!). What happens if we filter the traffic to just mobile traffic? And one step further – just mobile Chrome traffic (since only Chrome feeds CrUX and so only Chrome counts towards CWV)? Well, then we get a much more interesting picture:

COUNTRY	MOBILE P75 LCP VALUE	% OF MOBILE TRAFFIC
India	74.48%	31%
United States	88.34%	13%
Philippines	57.21%	8%
United Kingdom	92.07%	4%
Canada	93.75%	3%
Germany	93.01%	3%
Nigeria	37.45%	2%
Pakistan	56.32%	2%
Australia	85.88%	2%
Indonesia	75.34%	2%

India is actually the top mobile Chrome visitor, by quite some way – nearly triple the next highest visitor (USA)! The Philippines with its poor score has also shot up there to the number three spot, and Nigeria and Pakistan with their poor scores are also registering in the top 10. Now the bad overall LCP scores on mobile were starting to make sense.

While mobile has overtaken desktop as the most popular way to access the internet in the so-called Western world, there still is a fair mix of mobile and desktop here — often tied to our working hours when many of us are sat in front of a desktop.[14] The next billion

14 https://smashed.by/perfbyregion

users may not be the same, and mobile plays a much bigger part in those countries. The above stats show this is even true for sites like Smashing Magazine that you might consider would get more traffic from designers and developers sitting in front of desktops while designing and developing!

Additionally, because CrUX only measures Chrome users, that means countries with more iPhones (like the USA) will have a much smaller proportion of their mobile users represented in CrUX and so in core web vitals, thereby amplifying the effect of those countries.

Core Web Vitals Are Global

Core web vitals don't have a different threshold per country, and it doesn't matter if your site is visited by different countries – it simply registers all Chrome users the same. Google has confirmed this before, so Smashing Magazine will not get the ranking boost for the good USA scores, and not get it for the India users. Instead, all users go into the melting pot, and if the score for those page groupings does not meet the threshold, then the ranking signal for all users is affected.

Unfortunately, the world is not an even place. And web performance varies hugely by country[15] and shows a clear divide between richer and poorer countries. Technology costs money, and many countries are more focused on getting their populations online at all, rather than on continually upgrading infrastructure to the latest and greatest tech.

The lack of other browsers (like Firefox or iPhones) in CrUX has always been known, but we've always considered it more of a blind spot for measuring Firefox or iPhone performance. This example shows the impact is much bigger, and for sites with global traffic it skews the results significantly in favor of Chrome users, which often means poor countries, which often means worse connectivity.

15 https://smashed.by/perfbyregion

Should Core Web Vitals Be Split By Country?

On the one hand, it seems unfair to hold websites to the same standard if the infrastructure varies so much. Why should Smashing Magazine be penalized or held to a higher standard than a similar website that is only read by designers and developers from the Western world? Should Smashing Magazine block Indian users to keep the core web vitals happy? (I want to be quite clear here that this *never* came up in discussion, so please do take this as the author making the point and not a slight on Smashing!).

On the other hand, "giving up" on some countries by accepting their slowness risks permanently relegating them to the lower tier many of them are in. It's hardly the average Indian reader of Smashing Magazine's fault that their infrastructure is slower, and in many ways these are the people that deserve *more* highlighting and effort rather than less!

And it's not just a rich country versus poor country debate. Let's take the example of a French website which is aimed at readers in France, funded by advertising or sales from France, and has a fast website in that country. However, if the site is read by a lot of French Canadians, but suffers because the company does not use a global CDN, then should that company suffer in French Google Search because it's not as fast for those Canadian users? Should the company be held to ransom by the threat of core web vitals and have to invest in the global CDN to keep those Canadian readers, and so Google, happy?

Well, if a significant enough proportion of your viewers are suffering then that's exactly what the core web vital's initiative is supposed to surface. Still, it's an interesting moral dilemma which is a side effect of the core web vitals initiative being linked to SEO ranking boost: money always changes things!

One idea could be to keep the limits the same, but measure them per country. The French Google Search site could give a ranking boost to those users in France (because those users pass CWV for this site), while Google Search Canada might not (because they fail). That would level the playing field and measure sites to each country, even if the targets are the same.

Similarly, Smashing Magazine could rank well in the USA and other countries where they pass, but be ranked against other Indian sites (where the fact they are in the "needs improvement" segment might actually still be better than a lot of sites there, assuming they all suffer the same performance constraints).

Sadly, I think that would have a negative effect, with some countries again being ignored while sites only justify web performance investment for more lucrative countries. Plus, as this example already illustrates, the core web vitals are already complicated enough without bringing nearly 200 additional dimensions into play by having one for every country in the world!

So How To Fix It?

So we now finally knew why Smashing Magazine was struggling to pass core web vitals but what, if anything, could be done about it? The hosting provider (Netlify) already has the Mumbai CDN, which should therefore provide fast access for Indian users, so was this a Netlify problem? We had optimized the site as much as possible, so was this just something Smashing was going to have to live with? Well, no. Now we now return to our idea from earlier: optimizing the web fonts a bit more.

We could take the drastic option of not delivering fonts at all. Or perhaps not delivering fonts to certain locations (though that would be more complicated, given the SSG nature of Smashing Magazine's website). Alternatively, we could wait and load fonts in the front end, based on certain criteria, but that risked slowing down fonts for others while we assessed that criteria. If only there was some easy-to-use browser signal for when we should take this drastic action. Something like the SaveData header, which is intended exactly for this!

Save-Data and prefers-reduced-data

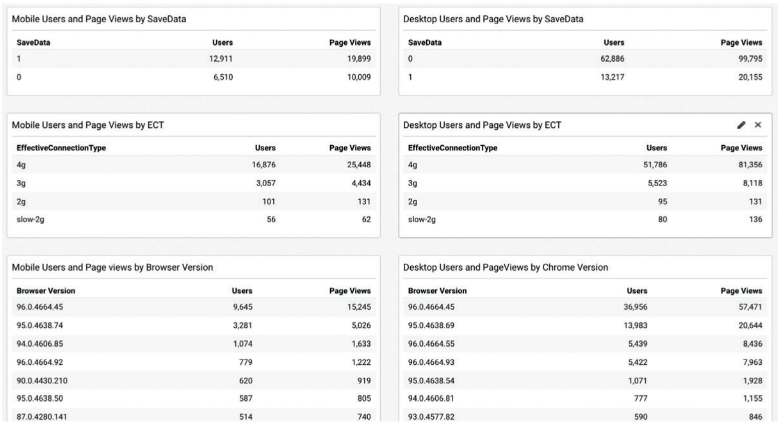
Save-Data is a setting that users can turn on in their browser when they really want to, well, save data. This can be useful for people on restricted data plans, for those traveling with expensive roaming charges, or for those in countries where the infrastructure isn't quite as fast as we'd like.

Users can turn on this setting in browsers that support it, and then websites can then use this information to optimize their sites even more than usual. Perhaps returning lower quality images (or turning images off completely), or not using fonts. And the best thing about this setting is that you are acting upon the user's request, and not arbitrarily making a decision for them (many Indian users might have fast access and not want a restricted version of the website!).

The Save Data information is available in two (soon to be three!) ways:

1. A Save-Data header is sent on each HTTP request. This allows dynamic back ends to change the HTML returned.
2. The `NetworkInformation.saveData` JavaScript API. This allows front-end scripts to check this and act accordingly.
3. The `prefers-reduced-data` media query, allowing CSS to set different options depending on this setting. This is available behind a flag in Chrome, but not yet on by default while it finishes standardization.

So the question is, do many Smashing Magazine readers (and particularly those in the countries struggling with core web vitals) use this option, and is this something we can therefore use to serve them a faster site? Well, when we added the `web-vitals` script mentioned above, we also decided to measure that, as well as the effective connection type (ECT). You can see the full script here.[16] After a bit of time allowing it to collect, we were able to display the results in a simple Google Analytics dashboard, along with the Chrome browser version:



Mobile Users and Page Views by SaveData

SaveData	Users	Page Views
1	12,911	19,899
0	6,510	10,009

Desktop Users and Page Views by SaveData

SaveData	Users	Page Views
0	62,886	99,795
1	13,217	20,155

Mobile Users and Page Views by ECT

EffectiveConnectionType	Users	Page Views
4g	16,876	25,448
3g	3,057	4,434
2g	101	131
slow-2g	56	62

Desktop Users and Page Views by ECT

EffectiveConnectionType	Users	Page Views
4g	51,786	81,356
3g	5,523	8,118
2g	95	131
slow-2g	80	136

Mobile Users and Page views by Browser Version

Browser Version	Users	Page Views
96.0.4664.45	9,645	15,245
95.0.4638.74	3,281	5,026
94.0.4606.85	1,074	1,633
96.0.4664.92	779	1,222
90.0.4430.210	620	919
95.0.4638.50	587	805
87.0.4280.141	514	740

Desktop Users and PageViews by Chrome Version

Browser Version	Users	Page Views
96.0.4664.45	36,956	57,471
95.0.4638.69	13,983	20,644
96.0.4664.55	5,439	8,436
96.0.4664.93	5,422	7,963
95.0.4638.54	1,071	1,928
94.0.4606.81	777	1,155
93.0.4577.82	590	846

Google Analytics Dashboard for India users of smashingmagazine.com.

So, the good news was that a large proportion of mobile Indian users (about two-thirds) did have this setting. The ECT was less useful with most showing as 4G. I've argued before that this API has gotten less and less useful as most users are classified under this 4G setting. Plus using this value effectively for initial loads is fraught with issues.[17]

More good news: most users seem to be on an up-to-date Chrome so would benefit from newer features like the `prefers-reduced-data` media query when it becomes fully available.

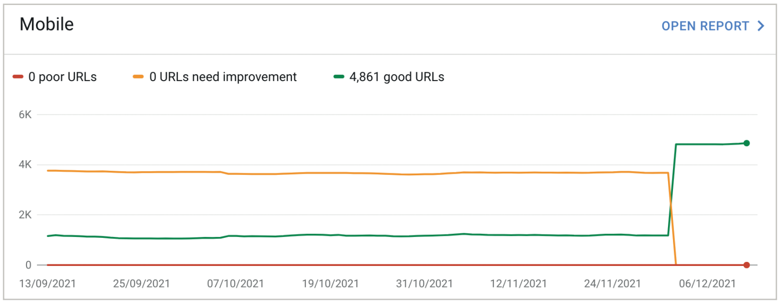
Ilya from the Smashing team applied the JavaScript API version to their font-loader script so additional fonts are not loaded for these

16 https://smashed.by/webvitalstracking
17 https://smashed.by/bandwidthqueries

users. The Smashing folks also applied the `prefers-reduced-data` media query to their CSS so fallback fonts are used rather than custom web fonts for the initial render, but this will not be taking effect for most users until that setting moves out of the experimental stage.

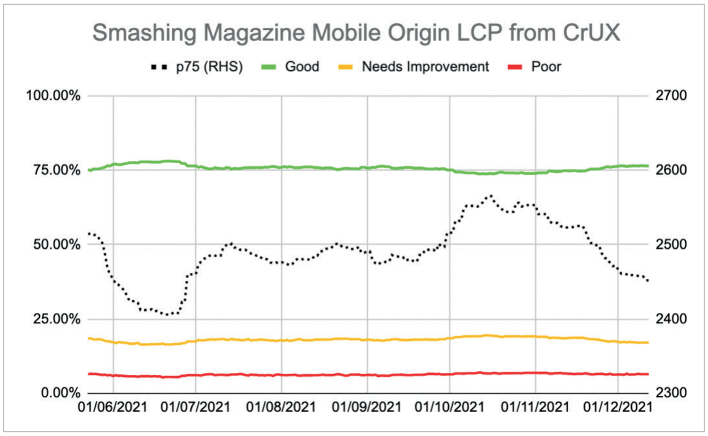
I Love That Graph in Google Search Console

And did it work? Well, we'll let Google Search Console tell that story as it showed us the good news a couple of weeks later:



Core Web Vitals graph going green in Google Search Console.

Additionally, since this was introduced in mid-November, the original level LCP score has steadily ticked downwards:



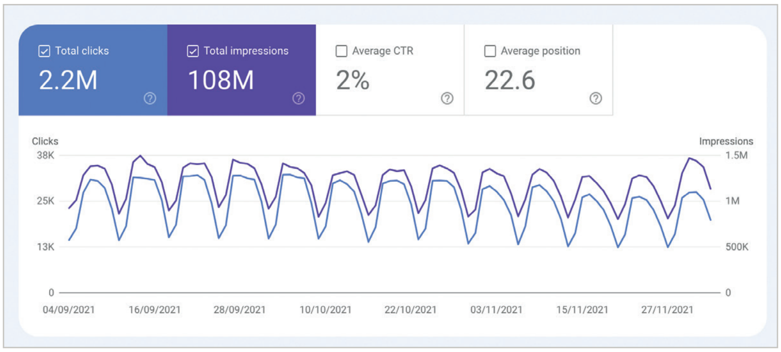
Updated tracking Smashing Magazine mobile origin LCP from CrUX.

There's still not nearly enough headroom to make me comfortable, but I'm hopeful that this will be enough for now, and will only improve when the `prefers-reduced-data` media query comes into play – hopefully soon.

Of course, a surge in traffic from mobile users with bad connectivity could easily be enough to flip the site back into the amber category, which is why you want that headroom. I'm sure the Smashing team will be keeping a close eye on their Google Search Console graph for a bit longer, but I feel we've made the best efforts our best effort to provide a basis to improve the experience of users, so I am hopeful it will be enough.

Impact of the User Experience Ranking Factor

The user experience ranking factor is supposed to be a small differentiator at the moment, and maybe we worried too much about a small issue that is, in many ways, outside of our control. If Smashing Magazine is borderline and the impact is small, then maybe the team should worry about other issues instead of obsessing over this one. But I can understand that and, as I said, Smashing Magazine is knowledgeable about performance and so understands why they wanted to solve – or at the very least understand! – this issue.



Search results graph from Google Search Console.

Was there any impact? Interestingly, we did see a large uptick in search impressions in the last week at the same time as it flipped to green.

It's since reverted back to normal, so this may have been an unrelated blip but interesting nonetheless!

Conclusions

So, an interesting case study with a few important points to take away:

- When RUM (including CrUX or Google Search Console) tells you there's a problem, there probably is! It's all too easy to try to compare your experiences and then blame the metric.
- Implementing your own RUM solution gives you access to much more valuable data than the high-level data CrUX is intended to provide, which can help you drill down into issues, plus also give you potentially more information about the devices your site visitors are using to visit your site.
- Core web vitals are global, and that causes some interesting challenges for global sites like Smashing Magazine. This can make it difficult to understand CrUX numbers unless you have a RUM solution and perhaps Google Search Console or CrUX could help surface this information more.
- Chrome usage also varies throughout the world, and on mobile is biased towards poorer countries where more expensive iPhones are less prevalent.
- Core web vitals are getting much better at measuring user experience. But that doesn't mean every user has to get the *same user experience* – especially if they are telling you (through things like the Save-Data option) that they would actually prefer a different experience.

I hope that this case study helps others in a similar situation who are struggling to understand their core web vitals. And I hope you can use the information here to make the experience better for your website visitors. Happy optimizing!

Note: *It should be noted that Vitaly, Ilya, and others in the Smashing team did all the work here, and a lot more performance improvements were not covered in the above article. I just answered a few queries for them on this specific problem over the last six months and then suggested this article might make an interesting case study for other readers to learn from.*

Smashing Magazine Key Takeaways

Identifying, fixing, and testing performance issues on a variety of devices and browsers is essential for improving Core Web Vitals scores.

When faced with recurring performance issues, Smashing Magazine used Lighthouse to audit their website and identify the areas where they could improve. They then used a variety of tools and techniques to fix the bottlenecks, such as optimizing images, reducing the number of third-party scripts, and using a caching plugin.

One of the challenges they faced was that core web vitals are measured globally, but Chrome usage varies throughout the world. This meant that they had to test their changes on a variety of devices and browsers to make sure that they were improving performance for all users.

Despite the challenges, Smashing Magazine was able to improve its core web vitals scores significantly. They also gained valuable insights into how to improve the user experience of their website. For example, they found that lazy-loading images and reducing the number of third-party scripts can have a significant impact on performance.

Tinder: Sophisticated Adaptive Loading Strategies

by Roderick Hsiao

Performance is now one of the core values of modern web apps.[1] Most users shift their web browser experiences from desktop to mobile. In addition to smaller screens, lower CPU power and network conditions have become important factors to make your content accessible to all users.

Video on adaptive loading: https://smashed.by/adaptiveloadingvideo

We want all of our users to enjoy the same experiences globally regardless of their network conditions (while respecting the user's preferences in terms of data consumption).

We use some aggressive prefetch/preload strategies[2] to make seamless experiences on interactions. However, these loading strategies rely heavily on predicting the user's intention to fetch the resources (JavaScript/CSS/images, etc.) needed to present the user experiences.

Some strategies we adopted:

1. **Route/Link base preload**: Similar to Addy Osmani's quicklink library,[3] we use a combination with React router, React lazy and intersection observer to preload the next page bundle when the link component enters the viewport and on idle.[4] (I also suggest you check Minko Gechev's *guess-js*,[5] which uses a data-driven approach to loading bundles.)

1 This case study was originally published in November 2019: https://smashed.by/adaptiveloading
2 https://smashed.by/reactperf
3 https://smashed.by/quicklink
4 https://smashed.by/idlize
5 https://smashed.by/guessjs

2. **Progressive image loading**: We create a library to preload different dimensions of the same image simultaneously via JavaScript before painting on the page. Users will see the smallest image first then the main picture without seeing a blank placeholder.

3. **Service worker precache**: For critical assets, we use Workbox to generate a precache manifest and load those assets up front in order to quickly serve bundles when needed.

4. **Browser resource priority hint**: By using HTML link `preload prefetch` resource priority hint, we are able to make the first page experience load instantly.

A deep dive of our web app could be found here: "A Tinder Progressive Web App Performance Case Study" by Addy Osmani[6]

Network-Aware Loading Strategies (Adaptive Loading)

Our performance journey doesn't end there. When testing on real mobile devices outside of the office, we notice that a lot of the time the experiences are less than desirable. The web app shows lots of loading states when entering a spotty network area even though we have already done a lot to improve the first page load. Data consumption also is one of the critical areas of focus when it comes to the international market where unlimited data might not be accessible or affordable to all users.

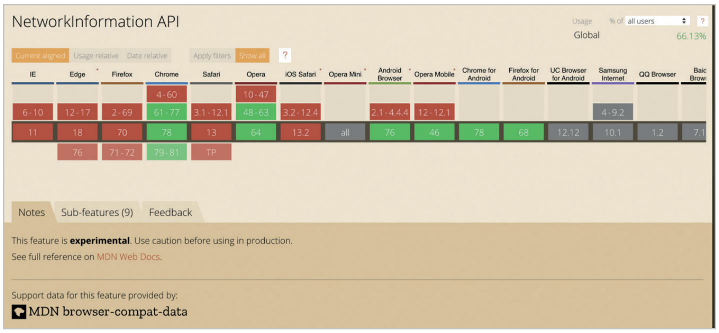
Thanks to the new browser APIs, we can now adaptively change our loading strategies based on different device/network conditions.

6 https://smashed.by/tinderpwa

NETWORK INFORMATION API

> *The Network Information API[7] provides information about the system's connection in terms of general connection type (e.g., 'wifi', 'cellular', etc.). This can be used to select high definition content or low definition content based on the user's connection.*

The API consists of the `NetworkInformation` interface and a single property to the `Navigator` interface: `Navigator.connection`.



NetworkInformation API

Usage % of all users ?
Global 66.13%

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baic Brow
			🟥 4-60		🟥 10-47										
🟥 6-10	🟥 12-17	🟥 2-69	🟩 61-77	🟥 3.1-12.1	🟩 48-63	🟥 3.2-12.4		🟥 2.1-4.4.4	🟥 12-12.1				▨ 4-9.2		
🟥 11	🟥 18	🟥 70	🟩 78	🟥 13	🟩 64	🟥 13.2	▨ all	🟩 76	🟩 46	🟩 78	🟩 68	▨ 12.12	▨ 10.1	▨ 1.2	▨ 7.1
	🟥 76	🟥 71-72	🟩 79-81	🟥 TP											

Notes | Sub-features (9) | Feedback

This feature is **experimental**. Use caution before using in production.
See full reference on MDN Web Docs.

Support data for this feature provided by:
MDN browser-compat-data

The current state of browser support for the Network Information API.

Although not fully supported by browser vendors, the API is heavily used in our web app. (In particular, browser support aligns with our international user base, where Android dominates the market.) We regard adaptive loading as progressive enhancement and serve default behavior for unsupported browsers.

We collect two main pieces of information from the API at Mozilla.

1. **NetworkInformation.effectiveType**
   "Returns the effective type of the connection meaning one of 'slow-2g', '2g', '3g', or '4g'. This value is determined

7 https://smashed.by/networkinformationapi

using a combination of recently observed round-trip time and downlink values."

2. **NetworkInformation.saveData**
   "Returns true if the user has set a reduced data usage option on the user agent."

Here are some examples we adopted for network-aware loading (based on the above prefetch scenario).

ROUTE/LINK BASE PRELOAD

As browsers have limited amounts of parallel download, it is critical to yield the download quota to high-priority resources. We noticed that when network conditions are poor, aggressive preloading will cause the main experiences to be flaky. Some important resources get deferred and the device's CPU occupied. We decided to only preload the next route when the user is in good network conditions.

We decided to only preload the next route when the user is in good network conditions.

```
// isOKConnection: network condition > 2g
const preload = (path: string, cb: () => void): void => {
   if (!isOKConnection() || isDataSavingMode()) {
     // don't preload if network is very bad
     return;
   }
    requestIdleCallbackManager.addTask(() => {
     routes.preload(path).then(() => cb());
   });
 };
```

PROGRESSIVE IMAGE LOADING

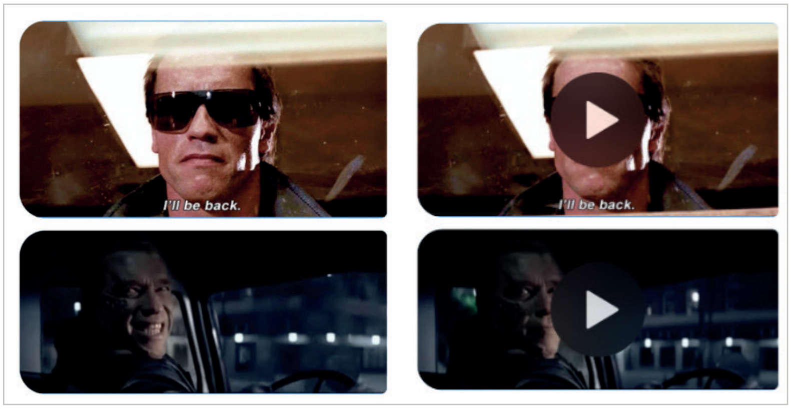
As mentioned above, we fetch multiple dimensions of the same image to progressively display in the browser. However, in fast network conditions, images complete loading almost at the same time, and the logic causes CPU overhead: fetching and swapping images after each image completes loading. We also honor the user's data consumption preference to prevent aggressively prefetching resources if they have the Data Saver turned on.

VIDEOS

We use video to display GIF images (much smaller size) and disable video autoplay for slow networks. (We selectively pass the `poster` attribute to the `video` element, which means browsers will only download the video resource after user interaction.)

```
const NetworkAwareVideo = () => {
   const shouldShowPoster = !isOKConnection() /* effectiveType
> 2g */ || isDataSavingMode();
   const autoplayProps = shouldShowPoster ? {
     poster
   }: {
     autoPlay:  true
   };
   return (
      <video
         className={className}
         loop
         muted
         onCanPlay={handleCanPlay}
         playsInline
         {...autoplayProps}
       >
         <source src={source} type="video/mp4" />
       </video>
    );
 }
```

Code snippet for adaptive video autoplay.



Autoplay (left) and poster without autoplay (right).

CAROUSEL

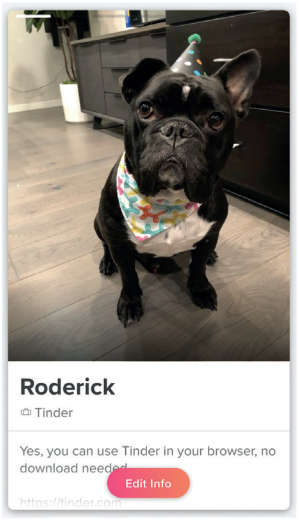
To prevent users seeing blank placeholders when navigating to the second page of the carousel, we used to load the next page photo together. This causes a janky swipe experience on our international market.

We changed the behavior to load only the first image under a slow network, and start prefetching the next page image if the user intends to view more photos for a specific profile (when they view the next page, or another interaction indicates the intention).

> From our experiences, we used a heuristic approach to decide one base strategy that will cover most browser metrics and devices. Then we added some enhancements for modern browsers to handle use cases in a more sophisticated way.

By implementing adaptive loading, we noticed an improvement in user engagement (more user interactions such as click or swipe) for emerging markets where Android is the major device OS and user experiences slow down networks frequently.

Progressively Increase Optimizations



Adaptive loading provides a new way to dynamically decide the serving logic to cover broader real-world scenarios, and it also opens up a space of creativity in terms of optimization strategies.

From our experiences, we used a heuristic approach to decide one base strategy that will cover most browser metrics and devices. Then we added some enhancements for modern browsers to handle use cases in a more sophisticated way. This approach also enabled us to maintain code complexity while adding or removing enhancements.

Tinder Key Takeaways

Adapt prefetch/preload based on network conditions and user preferences to save data on slower networks.

Most Tinder users access the app on mobile where the user experience is defined by the type of network used. Aggressive prefetch and preload are used to create seamless user interactions. However, aggressive preloads can slow down the main experience on poor network conditions. In cases where there is a limit on free data, users may have also set a data consumption preference on the browser that should be honored by apps.

To address this, the Tinder app preloads the next route only on good networks. They achieve this by using adaptive loading techniques that use the NetworkInformation api. The same principles are applied when prefetching images for progressive loading and image carousel.

Interview

Roderick Hsiao

Staff Engineer at Tinder

Author of **"Tinder: Sophisticated Adaptive Loading Strategies"**

What excited you or your team the most about the work in the case study?

NetworkInformation API opens a new chapter for dynamic optimization from the traditional heuristic approach.

We had multiple layer optimization strategies at Tinder but not all of them can reflect real-world experiences under different network or device conditions. Some optimization even creates less than preferred user experiences when adopted. Aggressive prefetching, for example, creates janky a UI experience under slow network conditions or on devices with lower computing power.

Client-side network calculations have been implemented before, but not all the results are reliable and sometimes even create a JavaScript overhead. With natively supported network condition APIs, we should be able to provide more sophisticated performance optimization to more users.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

We started to collect and analyze our user device and network information and page level performance instrumentation (page transition time) before adopting optimization.

To our surprise, although we had put a lot of effort into placing different optimizations for initial load and rendering performance, the runtime performance was still not ideal, and some key metrics showed degraded experiences when adopting our previous strategies.

After the new granular approach (divide user experiences in different network/device levels and dynamically serve different logic), we are seeing the metrics start to shift in our desired direction, and the gap for different user groups starts to align. Some key business metrics such as user engagement also improved.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

Our web app architecture – which centralized app-level logic such as app context, routing, content negotiation information, and more – enabled us to widely adopt optimization strategies for the top level of the web app easily.

After the new granular approach, we are seeing the metrics start to shift in our desired direction, and the gap for different user groups starts to align. Some key business metrics such as user engagement also improved.

Some optimization, however, is implemented at component/feature level, which makes the code logic cumbersome and challenging to extend. From a scalable point of view, it will be better to have an overview of different places implementing adaptive loading strategies and design feature implementation accordingly.

We will also spend more time designing implementation of instrumentation and measurement so experiment results can be presented in a scalable and reliable way.

What came next after the case study was published?

As we spent more time developing and investigating low-end devices, we noticed that our web app didn't really work as smoothly as on our own devices. We then started to design and optimize some runtime logic, such as virtualized lists with adaptive loading. There is much more we can do to make the app scalable. Our page navigation time on low-end devices improved around 20% after new optimization was implemented.

We also spent more effort on tooling to help us better understand real user experiences, such as utilizing CrUX, which shows a really different point of view for lab data versus field data.

Do you have any advice for teams that would like to follow in your footsteps?

Designing how to experiment and collect the result is equally important as designing adaptive loading strategies. Collecting user runtime data could be challenging as there are lots of variants that could make the field data unreliable. The user experience metrics could also shift in different directions; for instance, excessive pre-fetching could help load-time performance, but users might have a janky experience (or input delay), which might be hard to notice without comprehensive instrumentation. Spending more time analyzing the potential impact of each optimization strategy will be critical to evaluate success.

Find the bottleneck and start something simple. Web applications can be pretty complicated and lots of reasons could cause the app to perform unexpectedly, especially across different browsers and devices. Collect user data and dive deep into the code implementation first to find the bottleneck. Start with something simple but measurable, then develop in a more granular way over time.

Set up a good architecture to make the enhancement manageable. As the implementations could spread throughout your codebase, adopting optimization while maintaining good code complexity is also important to develop at scale. A very complex codebase will eventually prevent the team from enhancing any further.

Review and adjust your strategies. User devices and networks can improve over time. Regularly reviewing your optimization strategy will be important to maintain good user experiences.

Use real devices to test. Experience what the end user really feels on a 5+ year-old device – which is actually used by a good amount of users. A mindset of making the web app accessible to all users will be the key determining your app design direction.

React at 60 fps: Improving Scrolling Comments in Figma

by Kiko Lam

Figma enables closer collaboration between designers and non-designers by tightening the feedback loop.[1] By commenting directly on a file or prototype, teammates have important context, without needing to send files back and forth.

Since we first introduced Figma, we've been making consistent improvements to reach new levels of scale.[2] As more users left an increasing number of comments on their files, we started to observe performance problems. Knowing that Figma supports teams and organizations of all sizes, we had to do better. So we kicked off a project to improve the speed at which comments respond when users zoom and pan on the canvas.

React Faster, per Second

Our primary goal was to render the editor at 60 fps. No matter how our users collaborated, or how many comments and threads they created, we wanted the editor to perform at a speed that could flex to support them.

BUT FIRST, INFRASTRUCTURE

Before we dive into performance, it's important to understand a bit about Figma's technology. Figma is built on an unconventional stack – like our CTO Evan shared,[3] we essentially made "a browser inside a

1 The original version of this case study was published in August 2020: https://smashed.by/figmascrolling
2 https://smashed.by/introducingfigma
3 https://smashed.by/buildingfigma

browser." Our design editor is powered by WebGL and WebAssembly, with some of the user interface implemented in TypeScript and React.[4] Unlike most static interfaces built in React, comments are dynamic, and they can pan and zoom as part of the canvas. As you scroll around the canvas, we anchor your comment to something we call a comment pin, which ensures that your feedback stays exactly where you want it.

To do so, we need to get constant viewport updates from our editor. The viewport updates are stored in Redux and retrieved by the comment components. Each comment pin component uses this information to calculate where the comment pins should be rendered on the canvas in relation to the viewport.

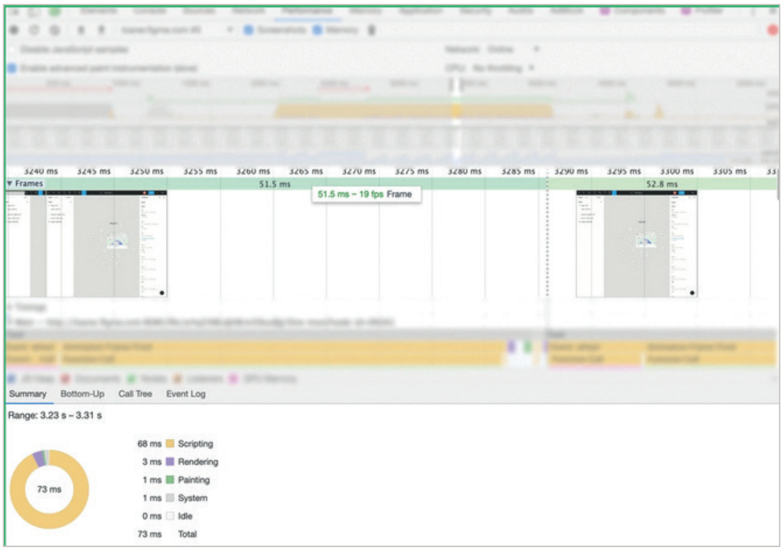
Getting to the Bottom of Slow Performance

In order to improve performance on this particular view, we needed to identify what was slowing it down. We used two main tools: Chrome performance tools and React Profiler.

COMPONENTS CONSTANTLY RE-RENDER

The profile generated from the Chrome performance tools shows that most of the time was spent on JavaScript (JS). About 68 ms per frame is spent on JS on a page with 30 comments, and only a small portion of the computing time per frame is spent on rendering and painting. Scripting refers to JS events and event handlers; rendering and painting have to do with the translation of HTML elements to displayable onscreen elements. It's promising that most improvement could be done on the JS and React optimization, but we still needed to understand more of what was happening under the hood of rendering the comment components in React.

4 https://smashed.by/webassembly

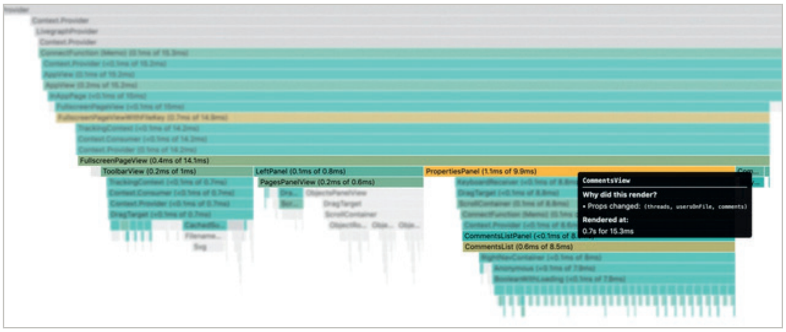


Range: 3.23 s – 3.31 s

68 ms	🟨	Scripting
3 ms	🟪	Rendering
1 ms	🟩	Painting
1 ms	▨	System
0 ms	□	Idle
73 ms		Total

Chrome's performance tool showed we spent the majority of time on scripting and rendered the comments view at 19 fps with 30 comment threads.

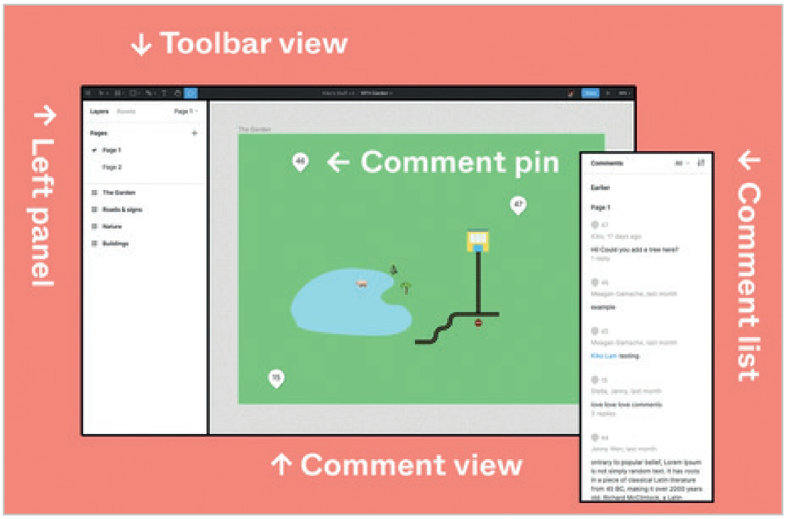
We used React Profiler to pinpoint which components were actually re-rendering. React profile shows that only about 1.8 ms is spent rendering the comments view. This re-rendering is necessary because its content is changing. However, from the React Profiler we observed that a lot of time was consumed rendering many fixed position components like the left panel, toolbar view, and properties


FullscreenPageView (0.4ms of 14.1ms)
ToolbarView (0.2ms of 1ms)
LeftPanel (0.1ms of 0.8ms)
PagesPanelView (0.2ms of 0.6ms)
PropertiesPanel (1.1ms of 9.9ms)
CommentsListPanel
CommentsList (0.6ms of 8.5ms)

CommentsView
Why did this render?
• Props changed: (threads, usersOnFile, comments)
Rendered at:
0.7s for 15.3ms

React Profiler shows the left panel, toolbar view, properties panel, comments list, and comments view re-rendered with every viewport change.

panel. But intuitively, only the comment should care about the viewport change, not these fixed components. The biggest inefficiency that creeps in as React applications grow is needlessly re-rendering components, which is exactly what we observed. This was a red flag, and we needed to address it.



How different components are structured in the Figma editor.

We started investigating why the other components were re-rendering when viewport information in the Redux store changed. We found that Redux runs every single middleware and loops through and runs `mapStateToProps` for every connected component, each time an action is dispatched. It then passes all of the data down through multiple layers to the comments view. But in our case, the only thing that should need this is the comments view. We had instances where we were passing in anonymous functions to force the components to render over and over again.

The biggest inefficiency that creeps in as React applications grow is needlessly re-rendering components, which is exactly what we observed. This was a red flag, and we needed to address it.

Our Approach

To fix the unnecessary re-rendering, we decided to remove viewport information from our Redux store and instead implemented our own event emitter[5] in our React codebase to broadcast this piece of information. We switched over from old components to functional components and, using React Hooks – which enabled us to memorize expensive computation – we now only do them when information changes. By avoiding dispatching an action to update viewport information in Redux, we successfully stopped running `mapStateToProps` for every connected component and avoided passing all of the data down through multiple layers to the comments view. As a result, we essentially prevented other components that don't need `ViewportInfo` from re-rendering.

BETTER, BUT NOT QUITE THERE

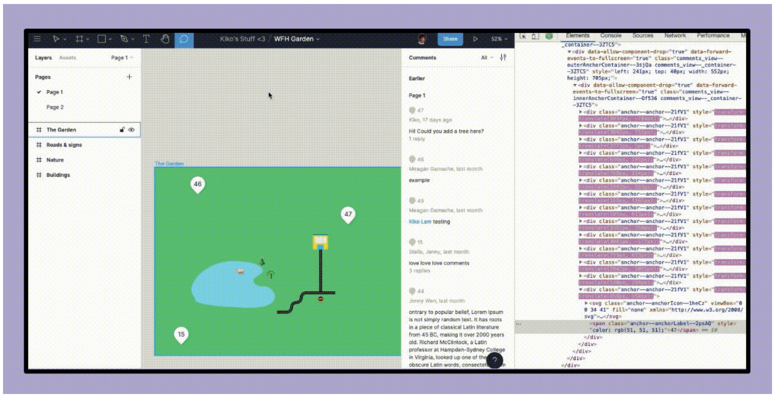
At this point, we ran the Chrome performance tool and React Profiler again. We saw that the constant re-rendering had stopped and the frame rate of the comment view had significantly improved from 15 fps to 50 fps with 50 comment pins. However, we still weren't quite at our goal of 60 fps. We also observed that performance linearly degrades with an increasing number of comment pins. So, we still had work ahead of us.

O(n) OPERATION ON EVERY VIEWPORT CHANGE

TJ Pavlu, an engineer on my team, worked with me on further improvements. By observing how the comment pins move on the document level, we noticed that every comment pin performs a transform action when the viewport moves. Each of the comment pin components was recomputing its pin position and performing a `transform-style` action with each viewport change (which you'll see below). In turn, comments view triggers an O(n) operation, where

5 https://smashed.by/eventemitters

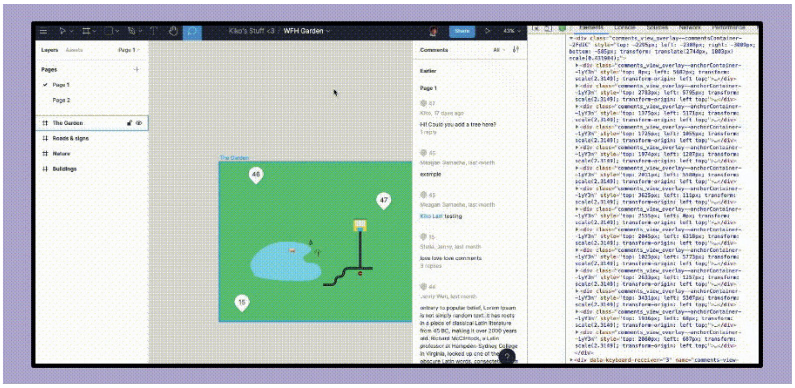
n is the number of comment threads as we pan and zoom. This might seem trivial for files with just a few comments, but the more comments there are, the slower the operation.



```
_container--3ZTC5">
  ▼<div data-allow-component-drop="true" data-forward-
  events-to-fullscreen="true" class="comments_view--
  outerAnchorContainer--3sjQa comments_view--_container-
  -3ZTC5" style="left: 241px; top: 40px; width: 552px;
  height: 705px;">
    ▼<div data-allow-component-drop="true" data-forward-
    events-to-fullscreen="true" class="comments_view--
    innerAnchorContainer--0f536 comments_view--_container-
    -3ZTC5">
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(3034px, -784px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(3092px, 731px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(2771px, 5px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(649px, 105px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(765px, 334px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(2082px, 353px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(168px, 1165px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(305px, 613px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(3362px, 350px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(3081px, -177px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(703px, 653px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(2841px, 1065px)">…</div>
      ▶<div class="anchor--anchor--21fV1" style="transform:
      translate(149px, 284px)">…</div>
      ▼<div class="anchor--anchor--21fV1" style="transform:
      translate(459px, 358px)">
        ▶<svg class="anchor--anchorIcon--1heCz" viewBox="0
        0 34 41" fill="none" xmlns="http://www.w3.org/2000/
        svg">…</svg>
          <span class="anchor--anchorLabel--2psAQ" style=
          "color: rgb(51, 51, 51);">47</span> == $0
        </div>
      </div>
    </div>
  </div>
</div>
```

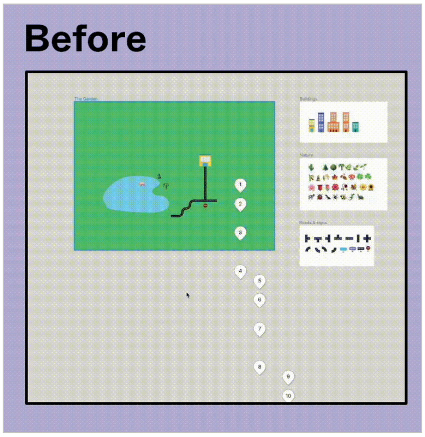
With every viewport change, each of the comment pin components recomputed its pin position and performed a `transform-style` *action.*

We came up with the solution to create an overlay container on the canvas and then to position the comment pins statically on this container. From there, we repositioned the overlay container (one computation) using CSS `translate` instead of doing so with each comment pin (*n* computations) as the viewport moves (illustrated in the second screen recording). Now, every viewport change triggers an `O`(1) operation instead of `O`(*n*) operation.



```
▼<div class="comments_view_overlay--commentsContainer-
-2FdIC" style="top: -2205px; left: -2380px; right: -3000px;
bottom: -585px; transform: translate(2744px, 1003px)
scale(0.431984);">
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 0px; left: 5082px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2783px; left: 5795px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 1375px; left: 5171px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 1725px; left: 1055px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 1974px; left: 1287px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2011px; left: 5580px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 3625px; left: 111px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2555px; left: 0px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2045px; left: 6318px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 1023px; left: 5773px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2633px; left: 1257px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 3431px; left: 5307px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 1936px; left: 68px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
  ▶<div class="comments_view_overlay--anchorContainer-
  -1yY3n" style="top: 2060px; left: 687px; transform:
  scale(2.3149); transform-origin: left top;">…</div>
</div>
▶<div data-keyboard-receiver="3" name="comments-view-
```

Only the overlay parent component recomposes its position on viewport changes.

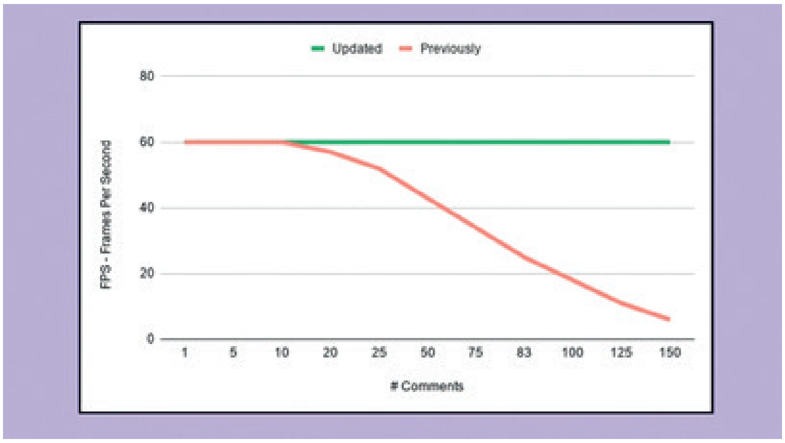


Panning is much smoother now..

We created this overlay container by creating a box around the most top-left pin and the most bottom-right pin. This means every time a new comment is added, we have to recompute this top-left/bottom-right boundary box. This trade-off is worth it because: a) comments are added less often than panning around the canvas; and b) this boundary box calculation happens when the canvas isn't moving.

Better Performance, Not Perfection

Based on how we scoped the project – achieving 60 fps for files with up to 150 comments – it was a success. You can see from the screen recording below that the interaction is much smoother and delivers a better user experience.



Now, we maintain 60fps rendering, no matter how many comments there are on a file.

But with performance, the work is never truly done. Moving forward, it'll be an ongoing process of setting new goals and identifying potential bottlenecks.

Beyond performance, we improved our React codebase and moved from old components to a new functional components system, while also taking advantage of React hooks. We'll continue to revisit our systems to ensure that Figma is built for scale.

Figma Key Takeaways

Optimizing the frame rate on a dynamic canvas for a smooth editing experience.

Figma allows for extensive collaboration between app designers and stakeholders through its intricate comments system. The Figma team realized that when the number of comments on a canvas increased, the frame rate of the editor measured in frames per second would go down. They analyzed the situation using Chrome performance tools and the React profiler, and realized that viewport updates received from the editor would cause not just the comments view to reload but also the other fixed position components.

Once they understood this, they were able to change their architecture to only re-render the comments view relative to a fixed canvas whenever viewport updates were received from the editor. After this change, they were able to achieve a framerate of 60 fps for files with up to 150 comments.

A Netflix Web Performance Case Study

by Addy Osmani

There are no silver bullets to web performance.[1] Simple static pages benefit from being server-rendered with minimal JavaScript. Libraries can provide great value for complex pages when used with care.

Netflix is one of the most popular video streaming services. Since launching globally in 2016, the company has found that many new users are not only signing up on mobile devices but are also using less-than-ideal connections to do so.

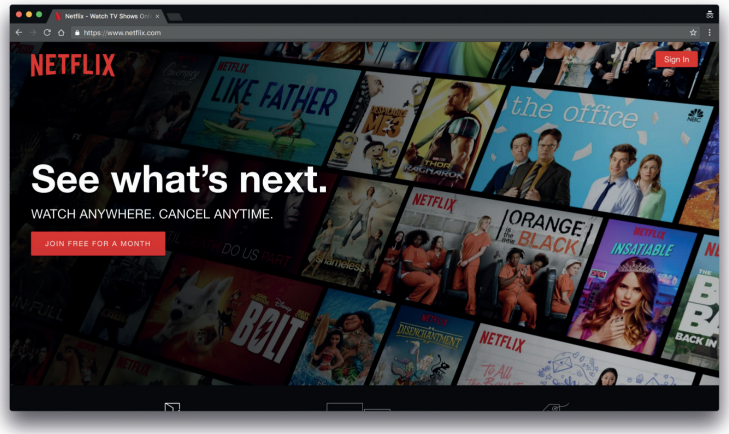
By refining the JavaScript used for *Netflix.com*'s sign-up process and using prefetching techniques, the developer team was able to provide a better user experience for both mobile and desktop users and offer several improvements.

- Loading and time to interactive (TTI) decreased by 50% (for the logged-out desktop homepage at *Netflix.com*)
- JavaScript bundle size reduced by 200 KB by switching from React and other client-side libraries to vanilla JavaScript. React was still used server-side.
- Prefetching HTML, CSS, and JavaScript (React) reduced TTI by 30% for future navigations.

1 The original version of this case study was published in November 2018: https://smashed.by/netflixperf

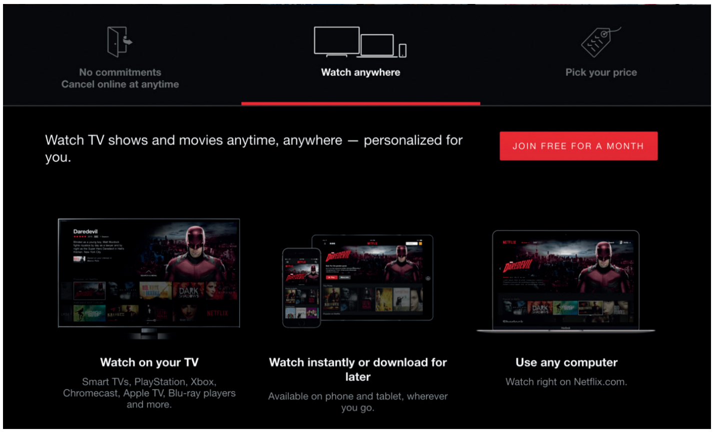
Reducing Time to Interactive by Shipping Less JavaScript

The area optimized for performance by the Netflix developers was the logged-out homepage, where users come to sign up or sign in to the site.



The Netflix.com homepage for new and logged-out members.

This page initially contained 300 KB of JavaScript, some of which was React and other client-side code (such as utility libraries like Lodash), and some of which was context data required to hydrate React's state.



Watch TV shows and movies anytime, anywhere — personalized for you.

Watch on your TV
Smart TVs, PlayStation, Xbox, Chromecast, Apple TV, Blu-ray players and more.

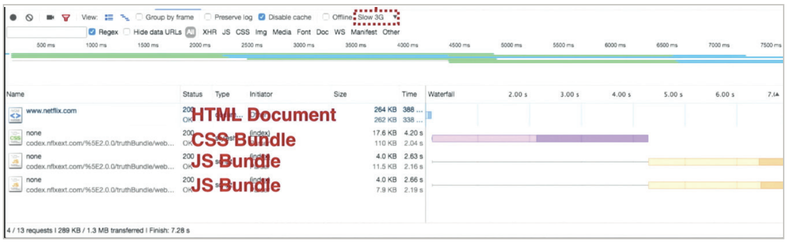
Watch instantly or download for later
Available on phone and tablet, wherever you go.

Use any computer
Watch right on Netflix.com.

Homepage tabs are an example of a component initially written using React.

All of Netflix's web pages are served by server-side rendered React, serving the generated HTML and then serving the client-side application, so it was important to keep the structure of the newly optimized homepage similar to maintain a consistent developer experience.

Using Chrome's DevTools and Lighthouse to simulate the logged-out homepage page being loaded on a 3G connection showed that the logged-out homepage took 7 seconds to load, far too long for just a simple landing page, so the potential for improvement was investigated. With some performance auditing, Netflix discovered their client-side JS had a high cost.



Name	Status	Type	Initiator	Size	Time
www.netflix.com	200 OK	[illegible]	[illegible]	264 KB / 262 KB	388 ... / 338 ...
none codex.nflxext.com/%5E2.0.0/truthBundle/web...	200 OK	[illegible]	(index)	17.6 KB / 110 KB	4.20 s / 2.04 s
none codex.nflxext.com/%5E2.0.0/truthBundle/web...	200 OK	[illegible]	(index)	4.0 KB / 11.5 KB	2.63 s / 2.16 s
none codex.nflxext.com/%5E2.0.0/truthBundle/web...	200 OK	[illegible]	(index)	4.0 KB / 7.9 KB	2.66 s / 2.19 s

4 / 13 requests | 289 KB / 1.3 MB transferred | Finish: 7.28 s

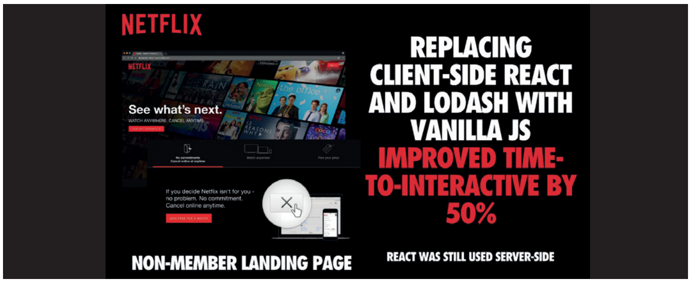
Network throttling for the unoptimized Netflix.com in Chrome DevTools.

By turning off JavaScript in the browser and observing which elements of the site still functioned, the developer team could determine if React was truly necessary for the logged-out homepage to function.

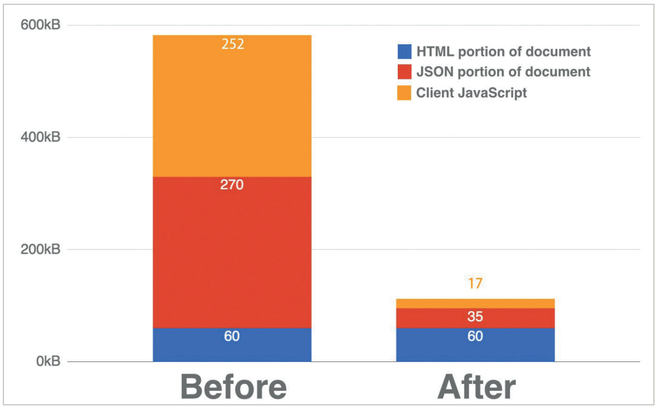
Since most of the elements on the page were basic HTML, remaining elements such as JavaScript click handling and class adding could be replaced with plain JavaScript, and the page's language switcher, originally built using React, was rebuilt in vanilla JavaScript using less than 300 lines of code.

Components ported to vanilla JavaScript were:

- basic interactions (tabs halfway down the homepage)
- language switcher
- cookie banner (for non-US visitors)
- client-side logging for analytics
- performance measurement and logging
- ad attribution pixel bootstrap code (which are sandboxed in an iframe for security)


NETFLIX
See what's next.
NON-MEMBER LANDING PAGE
REPLACING CLIENT-SIDE REACT AND LODASH WITH VANILLA JS
IMPROVED TIME-TO-INTERACTIVE BY 50%
REACT WAS STILL USED SERVER-SIDE

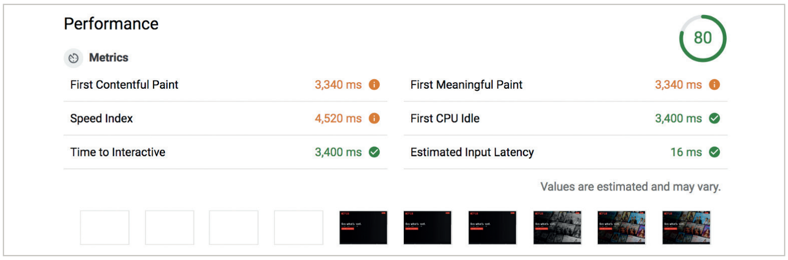
Even though React's initial footprint was just 45 KB, removing React, several libraries, and the corresponding app code from the client-side



Payload comparison before and after removing client-side React, Lodash, and other libraries.

reduced the total amount of JavaScript by over 200 KB, causing an over 50% reduction in Netflix's TTI for the logged-out homepage.

In a lab[2] environment using Lighthouse, we can validate that users can now interact with the Netflix homepage quickly. Desktop TTI is less than 3.5 seconds.



Performance

80

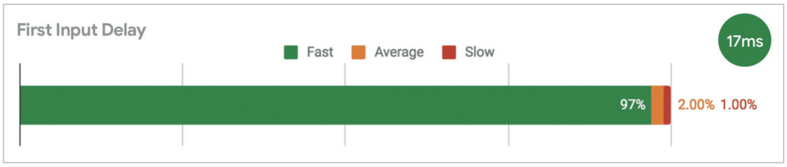
Metrics

First Contentful Paint	3,340 ms	First Meaningful Paint	3,340 ms
Speed Index	4,520 ms	First CPU Idle	3,400 ms
Time to Interactive	3,400 ms	Estimated Input Latency	16 ms

Values are estimated and may vary.

Lighthouse report after the time to interactive optimizations were made.

What about metrics from the field? Using the Chrome User Experience Report we can see first input delay (FID) – the time between a user's first interaction with your site and when the browser is actually able to respond – is fast for 97% of Netflix users on desktop.[3] This is great.



First input delay measures the delay users experience when interacting with the page.

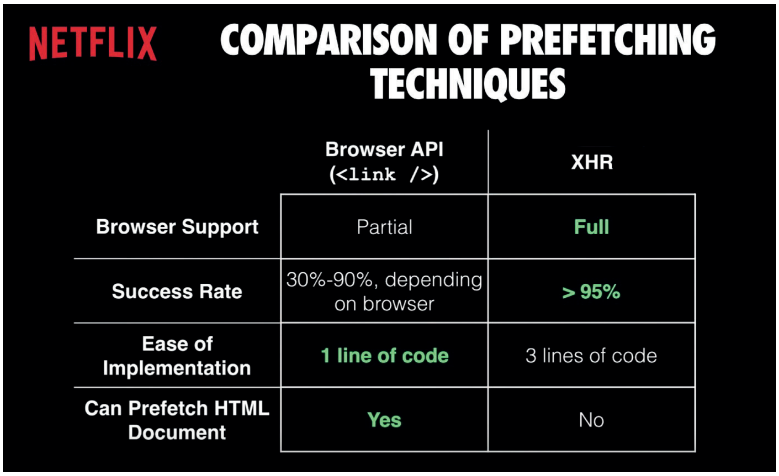
Prefetching React for Subsequent Pages

To further improve performance when navigating their logged-out homepage, Netflix utilized the time spent by users on the landing page to prefetch resources for the next page users were likely to land on.

2 https://smashed.by/firstinputdelay
3 https://smashed.by/netflixquery

This was achieved by using two techniques: the built-in `<link rel=prefetch>`[4] browser API, and XHR prefetching.

The built-in browser API consists of a simple `link` tag within the `head` tag of the page. It suggests to the browser that the resource (e.g. HTML, JS, CSS, images) can be prefetched, though it doesn't guarantee that the browser actually will prefetch the resource, and it hasn't yet been fully adopted by all browsers.[5]



NETFLIX

COMPARISON OF PREFETCHING TECHNIQUES

	Browser API (`<link />`)	XHR
Browser Support	Partial	Full
Success Rate	30%-90%, depending on browser	> 95%
Ease of Implementation	1 line of code	3 lines of code
Can Prefetch HTML Document	Yes	No

Comparison of prefetching techniques.

XHR prefetching, on the other hand, has been a browser standard for many years and produced a 95% success rate when the Netflix team prompted the browser to cache a resource. While XHR prefetching cannot be used to prefetch HTML documents, it was used by Netflix to prefetch the JavaScript and CSS bundle for subsequent pages.

Note: Netflix's HTTP response header configuration is preventing HTML caching with XHR (they do `no-cache` on the second page's HTML). `Link` prefetch is otherwise working as expected because it will work on HTML even if `no-cache` is present up to a certain point.

4 https://smashed.by/linkprefetching
5 https://smashed.by/linkprefetchsupport

```
// create a new XHR request
const xhrRequest = new XMLHttpRequest();
// open the request for the resource to "prefetch"
xhrRequest.open('GET', '../bundle.js', true);
// fire!
xhrRequest.send();
```

By using both the built-in browser API and XHR to prefetch HTML, CSS, and JS, the TTI was reduced by 30%. This implementation also required no JavaScript to be rewritten and didn't negatively impact the performance of the logged-out homepage, and hence offered a valuable tool for improving page performance at a very low risk.

NETFLIX
See what's next.
WATCH ANYWHERE. CANCEL ANYTIME.
PREFETCHING HTML, CSS AND JAVASCRIPT FOR FUTURE PAGES IMPROVED TIME-TO-INTERACTIVE BY 30%

After prefetching was implemented, the Netflix developers observed improvements by analyzing reductions in the TTI metric on the page, as well as using Chrome's developer tools to directly measure cache hits of resources.

Netflix Logged-Out Homepage: Optimization Summary

By prefetching resources and optimizing the client-side code on Netflix's logged-out homepage, Netflix was able to greatly improve its TTI metrics during the sign-up process. By prefetching future pages using the built-in browser API and XHR prefetching, Netflix was able to reduce TTI by 30%. This was for the second-page loading, which contained the bootstrapping code for single-page app sign-up flow.

The code optimizations carried out by the Netflix team showed that while React is a useful library, it may not provide an adequate solution to every problem. By removing React from the client-side code on the first landing page for sign-up, the TTI was improved by over 50%. Reducing TTI on the client-side also caused users to click the sign-up button at a greater rate, showing that code optimization can lead to a greater user experience overall.

The code optimizations carried out by the Netflix team showed that while React is a useful library, it may not provide an adequate solution to every problem. By removing React from the client-side code on the first landing page for sign-up, the TTI was improved by over 50%.

While Netflix didn't use React for the homepage, they prefetched it for subsequent pages. This allowed them to leverage client-side React throughout the rest of the single-page application sign-up process.

For more details on these optimizations, see "Building Performance Signup Flows in React" by Tony Edwards.[6]

6 https://smashed.by/tonyedwards

Conclusion

Netflix discovered opportunities to improve its TTI by keeping a close eye on the cost of JavaScript. To discover if your site has opportunities to do better here, consult your performance tools.[7]

The trade-off Netflix decided to make is to server-render the landing page using React, but also prefetch React and the code for the rest of the sign-up flow while on the landing page. This optimizes first-load performance, but also optimizes the time to load for the rest of the sign-up flow, which has a much larger JS bundle size to download since it's a single-page app.

Consider if leveraging vanilla JavaScript is an option for flows in your site. If you absolutely need to use libraries, try to only ship down code your users will need. Techniques like prefetching can help improve page load times for future page navigations.

Additional Notes

Netflix considered using Preact;[8] however, for a simple page flow with low interactivity, using vanilla JavaScript was a simpler choice for their stack.

Netflix experimented with service workers for static resource caching. At the time, Safari didn't support the API (it now does) but Netflix is exploring them again now. The Netflix sign-up flow needs greater legacy browser support than the member experience. Many users will sign-up on an older browser, but watch Netflix on their native mobile app or a TV device.

7 https://smashed.by/speedtools
8 https://preactjs.com/

The Netflix landing page is quite dynamic. It's the most heavily A/B tested page in the sign-up flow, with machine-learning models used to customize messaging and imagery depending on location, device type, and many other factors. With almost 200 countries supported, there are different localization, legal, and value messaging challenges for each derivative. For more on A/B testing, see "Testing into a Better User Experience" by Ryan Burgess.[9]

Netflix Key Takeaways

Replacing client-side React and Lodash with vanilla JavaScript improved time to interactive by 50%.

The logged-out or new user homepage for the Netflix app had initially contained 300 KB of JavaScript. This included React and Lodash code, including that required for hydration, and took 7 seconds to load on a 3G connection. The time was too high for a sign-up or sign-in page.

Most of the page was static, containing basic HTML. The remaining dynamic elements could be easily rebuilt using 300 lines of plain JavaScript code. This reduced the total amount of JavaScript by over 200 KB, thereby improving the time to interactive for the page. The use of XHR prefetching to fetch the next page users were likely to land on further improved the performance.

9 https://smashed.by/testingux

Shopping for Speed on eBay.com

by Addy Osmani & Senthil Padmanabhan

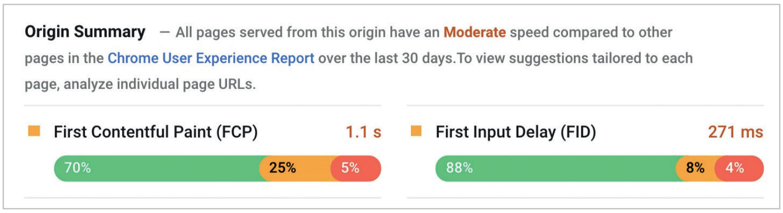
Speed" was an aptly named company-wide initiative for eBay in 2019, with many teams determined to make the site and apps as fast as possible for users.[1] In fact, for every 100 milliseconds improvement in search page loading time, eBay saw a 0.5% increase in "Add to Cart" count.

Through the adoption of performance budgets[2] (derived after doing a competitive study with the Chrome User Experience Report) and a focus on key user-centric performance metrics, eBay was able to make significant improvements to site speed.

	Web	**iOS**	**Android**
Homepage	10% ▲	12% ▲	28% ▲
Search	13% ▲	6% ▲	3% ▲
Item	3% ▲	7% ▲	5% ▲

eBay's speed improvements.

Ebay's Chrome User Experience Report data highlights these improvements too.



Origin Summary — All pages served from this origin have an Moderate speed compared to other pages in the Chrome User Experience Report over the last 30 days.To view suggestions tailored to each page, analyze individual page URLs.

Chrome User Experience Report data for first contentful paint and first input delay for the eBay.com origin.

1 The original versions of these case studies were published in 2020: https://smashed.by/shoppingforspeed & https://smashed.by/thousandcuts
2 https://smashed.by/perfbudgets

There's still more work ahead but here are eBay's findings so far.

Web Performance "Cuts"

The improvements eBay made were possible due to the reduction or "cuts" in the size and time of various factors that influence a user's journey.

REDUCE PAYLOAD ACROSS ALL TEXT RESOURCES

One way to make sites fast is to simply load less code. eBay reduced its text payloads by trimming all the unused and unnecessary bytes[3] of JavaScript, CSS, HTML, and JSON responses served to users. Previously, with every new feature, eBay kept increasing the payload of their responses, without cleaning up what was unused. This added up over time and became a performance bottleneck. Teams usually procrastinated on this clean-up activity, but you'd be surprised by how much eBay saved.

The "cut" here was the wasted bytes in the response payload.

CRITICAL PATH OPTIMIZATION FOR ABOVE-THE-FOLD CONTENT

Not every pixel on the screen is equally important. The content above the fold is more critical than something below the fold (and may require consideration per responsive viewport).[4] iOS/Android/desktop and web apps are aware of this, but what about services? eBay's service architecture has a layer called experience services, which the front ends (platform-specific apps and web servers) talk to.[5] This layer is specifically designed to be view- or device-based, rather than entity-based, like item, user, or order. eBay then introduced the concept of the critical path for experience services. When

3 https://smashed.by/unusedcode
4 https://smashed.by/abovethefold
5 https://smashed.by/experienceservices

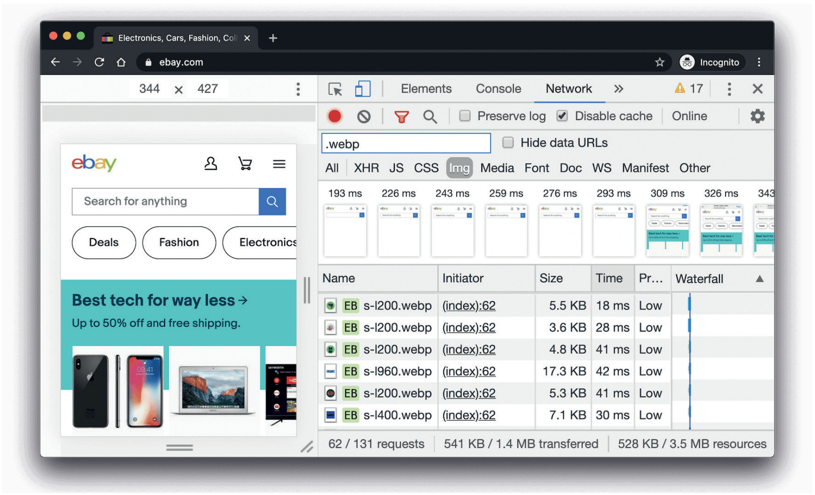
a request comes to these services, they work on getting the data for above-the-fold content immediately, by calling other upstream services in parallel. Once data is ready, it is instantly flushed. The below-the-fold data is sent in a later chunk or lazy-loaded. The outcome: users get to see above-the-fold content quicker.

The "cut" here was the time spent by services to display relevant content.

IMAGE OPTIMIZATIONS

Images are one of the largest contributors to page bloat. Even small optimizations go a long way. eBay did two optimizations for images.

First, eBay standardized on the WebP image format for search results across all platforms, including iOS, Android, and supported browsers.[6] The search results page is the most image-heavy page at eBay, and they were already using WebP, but not in a consistent pattern.



Filter: .webp

Name	Initiator	Size	Time	Pr...	Waterfall
EB s-l200.webp	(index):62	5.5 KB	18 ms	Low	
EB s-l200.webp	(index):62	3.6 KB	28 ms	Low	
EB s-l200.webp	(index):62	4.8 KB	41 ms	Low	
EB s-l960.webp	(index):62	17.3 KB	42 ms	Low	
EB s-l200.webp	(index):62	5.3 KB	41 ms	Low	
EB s-l400.webp	(index):62	7.1 KB	30 ms	Low	

62 / 131 requests | 541 KB / 1.4 MB transferred | 528 KB / 3.5 MB resources

WebP images being served to supported browsers on eBay.com.

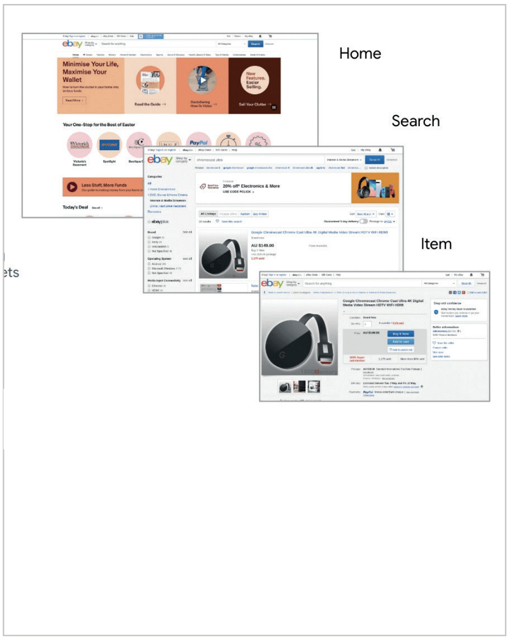
6 https://smashed.by/webpimages

Second, although eBay's listing images are heavily optimized (in both size and format), the same rigor did not apply for curated images (for example, the top module on the homepage). eBay has a lot of hand-curated images, which are uploaded through various tools. Previously the optimizations were up to the uploader, but now eBay enforces the rules within the tools, so all images uploaded will be optimized appropriately.

The "cut" here was the wasted image bytes sent to users.

PREDICTIVE PREFETCH OF STATIC ASSETS

A user session on eBay is not just one page: it is a flow. The flow can be a navigation, for example, from the homepage to a search page to an item page. So why don't pages in the flow help one another? That is the idea of predictive prefetch,[7] where one page prefetches the static assets required for the next likely page.



Predictive Asset Prefetching: Home prefetches assets for Search and Search prefetches them for the item page.

With predictive prefetch, when a user navigates to the predicted page, the assets are already in the browser cache. This is done for CSS and JavaScript assets,

7 https://smashed.by/predictiveprefetch

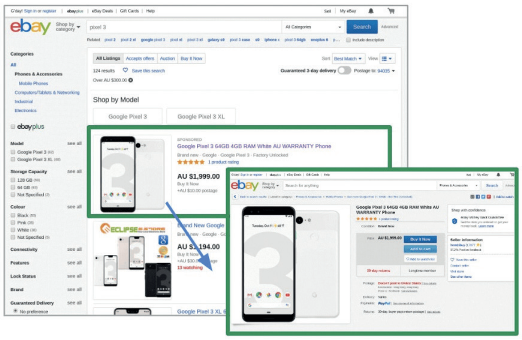
where the URLs can be retrieved ahead of time. One thing to note here is that it helps only on first-time navigations. On subsequent navigations, the static assets will already be in the cache.

The "cut" here was the network time for CSS and JavaScript static assets on the first navigation.

PREFETCHING TOP SEARCH RESULTS

When a user searches eBay, eBay's analytics data suggests it is highly likely that the user will navigate to an item in the top ten of the search results. So eBay now prefetches the items from search and keeps them ready for when the user navigates. The prefetching happens at two levels.

The first level happens server-side, where the item service caches the top ten items in search results. When the user goes to one of those items, eBay now saves server processing time. Server-side caching is leveraged by platform-specific apps and is rolled out globally.



Prefetch the top 10 items in Search results pages for fast subsequent loads using `requestIdleCallback()`

759ms

Faster Median Above Fold Time
(similar to FMP): 1.1s -> 0.39s

"eBay saw a positive impact on conversions from prefetching"

The other level happens in the browser cache, which is available in Australia. Item prefetch was an advanced optimization due to the dynamic nature of items. There are also many nuances to it: page impressions, capacity, auction items, and so on. You can learn more about it in LinkedIn's Performance Engineering Meetup presentation, or stay tuned for a detailed blog post on the topic from eBay's engineers.[8]

The "cut" here could either be server processing time or network time, depending on where the item is cached.

EAGER DOWNLOADING OF SEARCH IMAGES

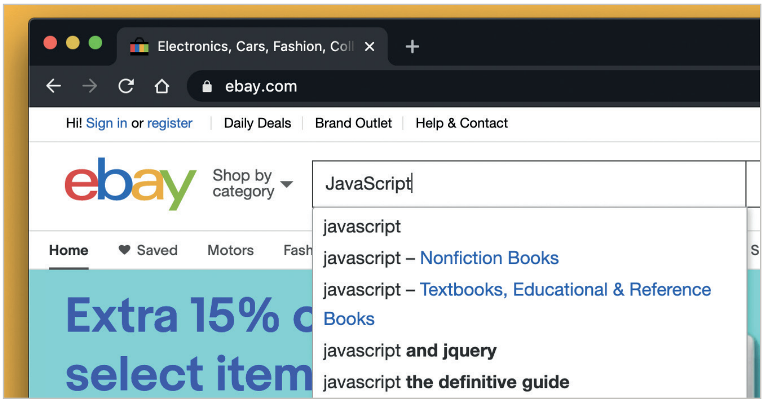
In the search results page, when a query is issued at a high level, two things happen. One is the recall/ranking step, where the most relevant items matching the query are returned. The second step is augmenting the recalled items with additional user context-related information, such as shipping costs. eBay now immediately sends the first ten item images to the browser in a chunk along with the header, so the downloads can start before the rest of the markup arrives. As a result, the images will now appear quicker. This change is rolled out globally for the web platform.

The "cut" here was the download start time for search result images.

EDGE CACHING FOR AUTOSUGGESTION DATA

When users type in letters in the search box, suggestions pop up. These suggestions do not change for letter combinations for at least a day. They are ideal candidates to be cached and served from a CDN (content delivery network - for a maximum of 24 hours) instead of requests going all the way to a data center. International markets especially benefit from CDN caching.

8 https://smashed.by/linkedinperf



CDN Caching of search suggestions.

There was a catch, though. eBay had some elements of personalization in the suggestions pop-up, which couldn't be cached efficiently. Fortunately, it was not an issue in the platform-specific apps, as the user interfaces for personalization and suggestions could be separated. For the web, in international markets, latency was more important than the small benefit of personalization. With that out of the way, eBay now has autosuggestions served from a CDN cache globally for platform-specific apps and non-US markets for eBay.com.

The "cut" here was the network latency and server processing time for autosuggestions.

EDGE CACHING FOR UNRECOGNIZED HOMEPAGE USERS

For the web platform, the homepage content for unrecognized users is the same for a particular region. These are users who are either using eBay for the first time or starting a fresh session, hence no personalization. Though the components on the homepage keep changing frequently there is still room for caching.

eBay decided to cache the unrecognized user content (HTML) on their edge network (points of presence[9]) for a short period. First-

9 https://smashed.by/pointofpresence

time users can now get homepage content served from a server near them, instead of from a faraway data center. eBay is still experimenting with this in international markets, where it will have a bigger impact.

The "cut" here is again both network latency and server processing time for unrecognized users.

OPTIMIZATIONS FOR OTHER PLATFORMS

iOS/Android App Parsing Improvements

iOS/Android apps talk to back-end services whose response format is typically JSON. These JSON payloads can be large. Instead of parsing the whole JSON to render something on the screen, eBay introduced an efficient parsing algorithm that optimizes for content that needs to be displayed immediately.

Performance is a feature and a competitive advantage. Optimized experiences lead to higher user engagement, conversions, and ROI.

Users can now see the content quicker. In addition, for the Android app, eBay starts initializing the search view controllers as soon as the user starts typing in the search box (iOS already had this optimization). Previously this happened only after users pressed the search button. Now users can get to their search results faster.

The "cut" here was the time spent by devices to display relevant content.

Android App Start-Up Time Improvements

This applies to cold start time optimizations for Android apps.[10] When an app is cold-started, a lot of initialization happens both at the OS level and application level. Reducing the initialization time at

10 https://smashed.by/coldstart

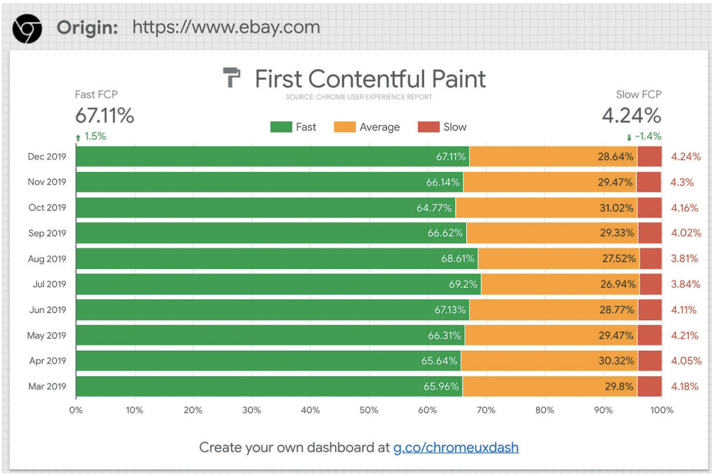
the application level helps users see the home screen quicker. eBay did some profiling and noticed that not all initializations are required to display content and that some can be done lazily.

More importantly, eBay observed that there was a blocking third-party analytics call that delayed the rendering on the screen. Removing the blocking call and making it async further helped cold start times.

The “cut” here was the unnecessary start-up time for Android apps.

Conclusions

All the performance “cuts” eBay made contributed collectively towards moving the needle, and it happened over a period of time. The releases were phased in throughout the year, with each release shaving off tens of milliseconds, ultimately reaching the point where eBay is now:



The impact of eBay’s speed efforts on their field metrics over time, as illustrated by the Chrome UX Report Dashboard.

Performance is a feature and a competitive advantage. Optimized experiences lead to higher user engagement, conversions, and ROI. In eBay's case, these optimizations varied from things that were low-effort to a few that were advanced.

Ebay Key Takeaways

Introducing cuts across the board to meet performance budgets and improve speed.

eBay launched its "Speed" initiative in 2019 to focus on improving the performance of critical eBay pages across all platforms. They analyzed the Chrome user experience reports with a focus on user-centric metrics to define their desired performance budgets. The next step was to introduce cuts and optimizations for each entity to meet the performance budgets of the whole. Some of the key cuts introduced were as follows:

- Trim unused and unnecessary bytes for JavaScript, CSS, HTML, and JSON responses served to users.
- Optimization for above-the-fold content to be fetched and delivered instantly to cut the time spent on rendering critical content.
- Standardize the use of WebP images for all platforms and optimize curated images during upload.
- Cut down on time to load the next page by using predictive prefetch to prefetch static content for the next page likely to be requested.
- Cut search time by prefetching the top search results.

How CLS Optimizations Increased Yahoo! JAPAN News's Page Views

by Shunya Shishido, Tomoki Kiraku, & Milica Mihajlija

Yahoo! JAPAN is one of the largest media companies in Japan, providing over 79 billion page views per month.[1] Their news platform, Yahoo! JAPAN News has more than 22 billion page views per month and an engineering team dedicated to improving the user experience.

By continuously monitoring core web vitals (CWV), they correlated the site's improved cumulative layout shift (CLS) score with a 15% increase in page views per session and 13% increase in session duration. Cumulative layout shift measures how visually stable a website is – it helps quantify how often users experience unexpected layout shifts.

Page content moving around unexpectedly often causes accidental clicks, disorientation on the page, and, ultimately, user frustration. Frustrated users tend not to stick around for long. To keep users happy, the page layout should stay stable through the entire life cycle of the user journey. For Yahoo! JAPAN News this improvement had a significant positive impact on business critical engagement metrics.

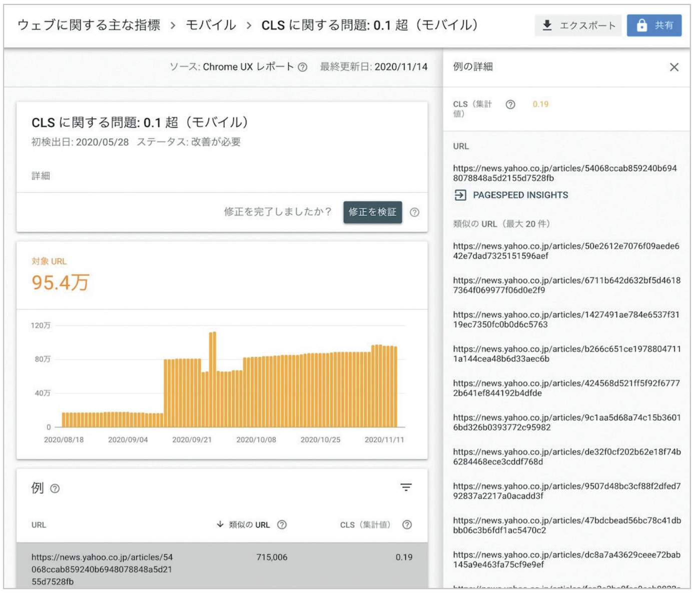
To keep users happy, the page layout should stay stable through the entire life cycle of the user journey.

For technical details on how they improved the CLS, read the Yahoo! JAPAN News engineering team's post.[2]

1 The original version of this article was published in March 2021: https://smashed.by/japannews
2 https://smashed.by/yahootech

Identifying the Issue

Monitoring core web vitals, including CLS, is crucial in catching issues and identifying where they're coming from. CWV tools[3] were used at Yahoo! JAPAN News for monitoring. Search Console provided a great overview of groups of pages with performance issues, and Lighthouse helped identify per-page opportunities to improve page experience. Using these tools, they discovered that the article detail page had poor CLS.



CLS に関する問題: 0.1 超（モバイル）
初検出日: 2020/05/28 ステータス: 改善が必要

詳細

対象 URL
95.4万

例

URL	類似の URL	CLS（集計値）
https://news.yahoo.co.jp/articles/54068ccab859240b6948078848a5d2155d7528fb	715,006	0.19

例の詳細

CLS（集計値） 0.19

URL

https://news.yahoo.co.jp/articles/54068ccab859240b6948078848a5d2155d7528fb

類似の URL（最大 20 件）

https://news.yahoo.co.jp/articles/50e2612e7076f09aede642e7dad7325151596aef

https://news.yahoo.co.jp/articles/6711b642d632bf5d46187364f069977f06d0e2f9

https://news.yahoo.co.jp/articles/1427491ae784e6537f3119ec7350fc0b0d6c5763

https://news.yahoo.co.jp/articles/b266c651ce19788047111a144cea48b6d33aec6b

https://news.yahoo.co.jp/articles/424568d521ff5f92f67772b641ef844192b4dfde

https://news.yahoo.co.jp/articles/9c1aa5d68a74c15b36016bd326b0393772c95982

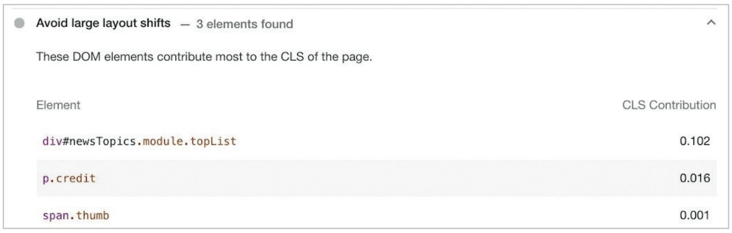
https://news.yahoo.co.jp/articles/de32f0cf202b62e18f74b6284468ece3cddf768d

https://news.yahoo.co.jp/articles/9507d48bc3cf88f2dfed792837a2217a0acadd3f

https://news.yahoo.co.jp/articles/47bdcbead56bc78c41dbbb06c3b6fdf1ac5470c2

https://news.yahoo.co.jp/articles/dc8a7a43629ceee72bab145a9e463fa75cf9e9ef

Google Search Console Core Web Vitals report.



Avoid large layout shifts — 3 elements found

These DOM elements contribute most to the CLS of the page.

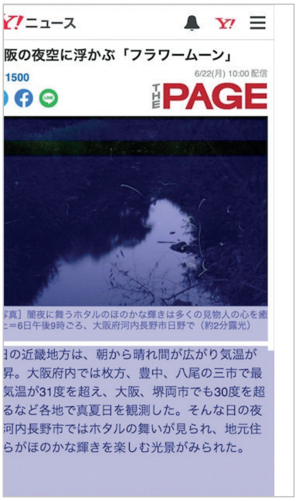
Element	CLS Contribution
`div#newsTopics.module.topList`	0.102
`p.credit`	0.016
`span.thumb`	0.001

Lighthouse's "Avoid large layout shifts" audit shows which elements are contributing to CLS score and how much.

3 https://smashed.by/vitalstools

It's important to keep in mind the cumulative part of the cumulative layout shift – the score is captured through the entire page life cycle. In the real world, the score can include shifts that happen as a result of user interactions, such as scrolling a page or tapping a button. To collect CLS scores from the field data, the team integrated web-vitals JavaScript library reporting.[4]

As a part of a performance monitoring strategy, they're also working on building an internal tool with Lighthouse CI to continuously audit performance across businesses in the company.[5]



阪の夜空に浮かぶ「フラワームーン」

6/22(月) 10:00 配信

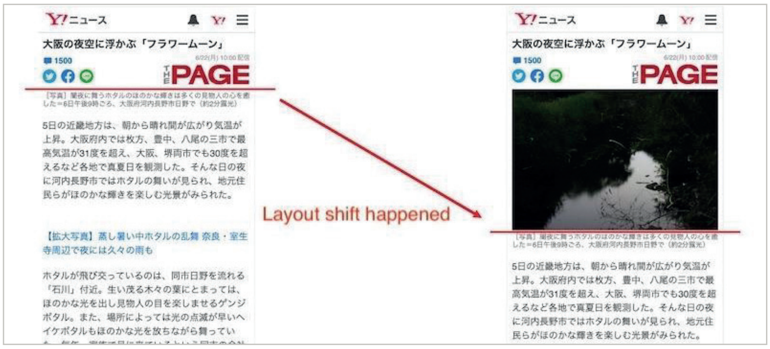
写真］闇夜に舞うホタルのほのかな輝きは多くの見物人の心を癒
た＝6日午後9時ごろ、大阪府河内長野市日野で（約2分露光）

日の近畿地方は、朝から晴れ間が広がり気温が
昇。大阪府内では枚方、豊中、八尾の三市で最
気温が31度を超え、大阪、堺両市でも30度を超
るなど各地で真夏日を観測した。そんな日の夜
河内長野市ではホタルの舞いが見られ、地元住
らがほのかな輝きを楽しむ光景がみられた。

Visualized layout shifts.

The team used Chrome DevTools to identify which elements were making layout shifts on the page. Layout Shift Regions in DevTools visualizes elements that contribute to CLS by highlighting them with a blue rectangle whenever a layout shift happens.[6]

They figured out that a layout shift occurred after the hero image at the top of the article was loaded for the first view.



大阪の夜空に浮かぶ「フラワームーン」

6/22(月) 10:00 配信

［写真］闇夜に舞うホタルのほのかな輝きは多くの見物人の心を癒
した＝6日午後9時ごろ、大阪府河内長野市日野で（約2分露光）

5日の近畿地方は、朝から晴れ間が広がり気温が
上昇。大阪府内では枚方、豊中、八尾の三市で最
高気温が31度を超え、大阪、堺両市でも30度を超
えるなど各地で真夏日を観測した。そんな日の夜
に河内長野市ではホタルの舞いが見られ、地元住
民らがほのかな輝きを楽しむ光景がみられた。

【拡大写真】蒸し暑い中ホタルの乱舞 奈良・室生
寺周辺で夜には久々の雨も

ホタルが飛び交っているのは、同市日野を流れる
「石川」付近。生い茂る木々の葉にとまっては、
ほのかな光を出し見物人の目を楽しませるゲンジ
ボタル。また、場所によっては光の点滅が早いヘ
イケボタルもほのかな光を放ちながら舞ってい

大阪の夜空に浮かぶ「フラワームーン」

6/22(月) 10:00 配信

［写真］闇夜に舞うホタルのほのかな輝きは多くの見物人の心を癒
した＝6日午後9時ごろ、大阪府河内長野市日野で（約2分露光）

5日の近畿地方は、朝から晴れ間が広がり気温が
上昇。大阪府内では枚方、豊中、八尾の三市で最
高気温が31度を超え、大阪、堺両市でも30度を超
えるなど各地で真夏日を観測した。そんな日の夜
に河内長野市ではホタルの舞いが見られ、地元住
民らがほのかな輝きを楽しむ光景がみられた。

Layout shift on the article detail page.

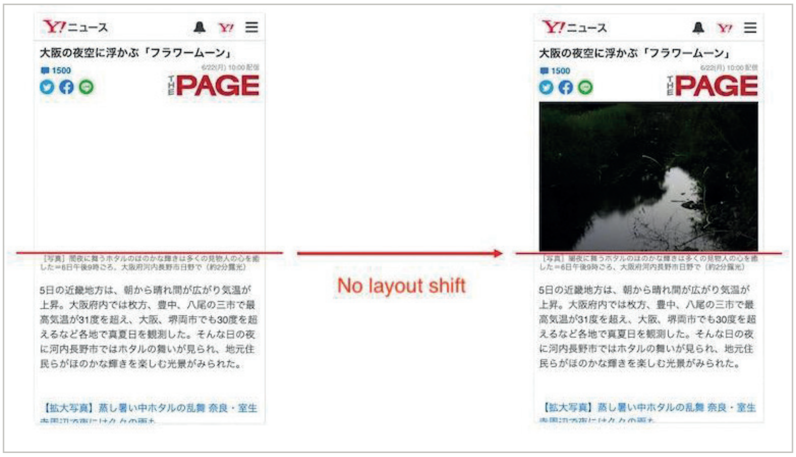
4 https://smashed.by/chromewebvitals
5 https://smashed.by/lighthouseci
6 https://smashed.by/shiftregions

In these examples, when the image finishes loading, the text gets pushed down (the position change is indicated with the red line).

Improving CLS for Images

For fixed-size images, layout shifts can be prevented by specifying the `width` and `height` attributes in the `img` element and using the CSS `aspect-ratio` property available in modern browsers. However, Yahoo! JAPAN News needed to support not only modern browsers, but also browsers installed in relatively old operating systems such as iOS 9.

They used aspect ratio boxes – a method that uses markup to reserve the space on the page before the image is loaded.[7] This method requires knowing the aspect ratio of the image in advance, which they were able to get from the back-end API.



大阪の夜空に浮かぶ「フラワームーン」

【写真】闇夜に舞うホタルのほのかな輝きは多くの見物人の心を癒した＝6日午後9時ごろ、大阪府河内長野市日野で（約2分露光）

5日の近畿地方は、朝から晴れ間が広がり気温が上昇。大阪府内では枚方、豊中、八尾の三市で最高気温が31度を超え、大阪、堺両市でも30度を超えるなど各地で真夏日を観測した。そんな日の夜に河内長野市ではホタルの舞いが見られ、地元住民らがほのかな輝きを楽しむ光景がみられた。

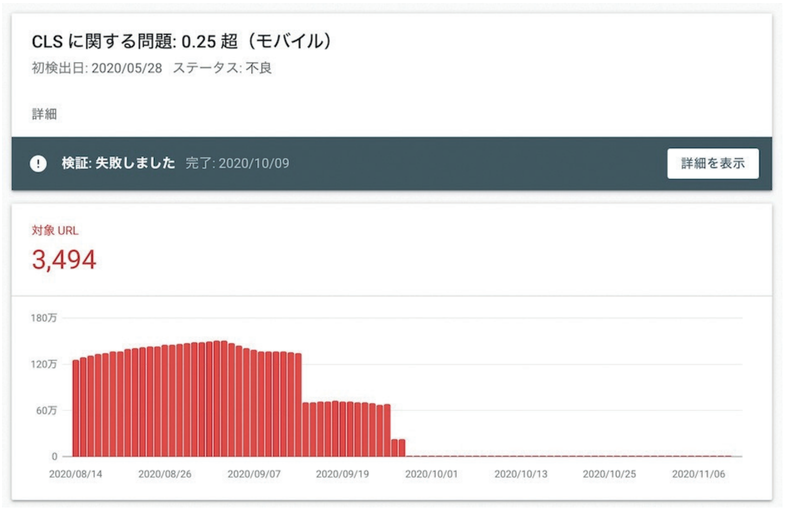
【拡大写真】蒸し暑い中ホタルの乱舞 奈良・室生

Left: Reserved blank space for the image at the top of the page;
Right: The hero image loaded in the reserved space without layout shifts.

7 https://smashed.by/aspectratioboxes

Results

The number of URLs with poor performance in Search Console decreased by 98%, and CLS in lab data decreased from about 0.2 to 0. More importantly, there were several correlated improvements in business metrics.[8]


CLS に関する問題: 0.25 超（モバイル）
初検出日: 2020/05/28　ステータス: 不良
詳細
検証: 失敗しました　完了: 2020/10/09　詳細を表示
対象 URL
3,494

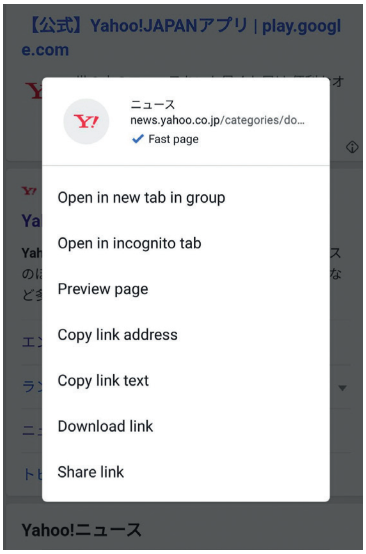
Search Console after improvements.

Search Console does not reflect improvements in real-time.

When Yahoo! JAPAN News compared user engagement metrics before and after CLS optimization, they saw multiple improvements:

- 15.1% more page views per session
- 13.3% longer session duration
- 1.72 percentage points lower bounce rate

8 https://smashed.by/layoutshift



"Fast page" label in Chrome on Android.

By improving CLS and other core web vitals metrics, Yahoo! JAPAN News also got the "Fast page" label in the context menu of Chrome Android.[9]

Layout shifts are frustrating and discourage users from reading more pages, but that can be improved by using the appropriate tools, identifying issues, and applying best practices. Improving CLS is a chance to improve your business.

Yahoo! Japan Key Takeaways

Improving CLS for images led to a 15.1% increase in page views per session and 13.3% longer session durations.

Yahoo! Japan News gets 22 billion page views per month. Due to the availability of high-end phones and a decent network, speed may not be a core performance issue in Japan. However, page content that moves around unexpectedly can lead to accidental clicks, disorientation, and ultimately user frustration. Yahoo! Japan News's engineering team realized that this was the issue with their article details page and as a result, it had poor CLS.

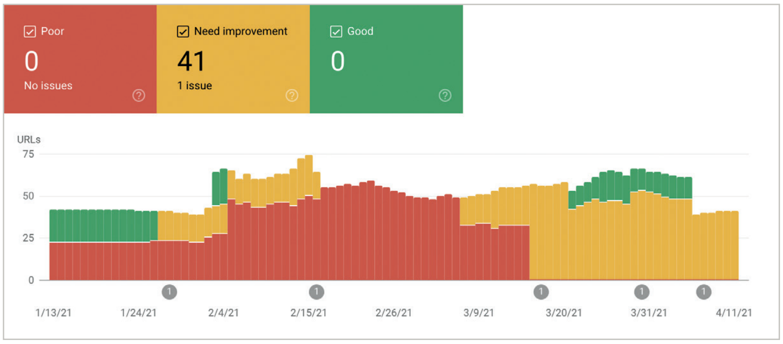
Through tests, they realized that there was a layout shift after the hero image at the top of the article was loaded for the first view. They used aspect ratio boxes to reserve space for the image before it is loaded. This change alone was enough to get rid of the CLS completely, bringing it down to zero from 0.2. This directly resulted in a significant improvement in user engagement metrics for the site.

9 https://smashed.by/fastpage

Instant Domain Search: How We Improved Our Core Web Vitals

by Beau Hartshorne

Last year,[1] Google started emphasizing the importance of core web vitals and how they reflect a person's real experience when visiting sites around the web.[2] Performance is a core feature of our company, Instant Domain Search[3] — it's in the name. Imagine our surprise when we found that our vitals scores were not great for a lot of people. Our fast computers and fiber internet masked the experience real people have on our site. It wasn't long before a sea of red "poor" and yellow "needs improvement" notices in our Google Search Console needed our attention. Entropy had won, and we had to figure out how to clean up the jank — and make our site faster.



This is a screenshot from our mobile core web vitals report in Google Search Console. We still have a lot of work to do!

I founded Instant Domain Search in 2005 and kept it as a side-hustle while I worked on a Y Combinator company (Snipshot, W06), before working as a software engineer at Facebook. We've recently grown to a small group based mostly in Victoria, Canada, and we are

1 The original version of this case study was published in May 2021: https://smashed.by/instantdomainsearch
2 https://smashed.by/evaluatingpageux
3 https://instantdomainsearch.com/

working through a long backlog of new features and performance improvements. Our poor web vitals scores, and the looming Google update, brought our focus to finding and fixing these issues.[4]

When the first version of the site was launched, I'd built it with PHP, MySQL, and XMLHttpRequest. Internet Explorer 6 was fully supported, Firefox was gaining share, and Chrome was still years from launch. Over time, we've evolved through a variety of static-site generators, JavaScript frameworks, and server technologies. Our current front-end stack is React served with Next.js and a back-end service built in Rust to answer our domain name searches. We try to follow best practice by serving as much as we can over a CDN, avoiding as many third-party scripts as possible, and using simple SVG graphics instead of bitmap PNGs. It wasn't enough.

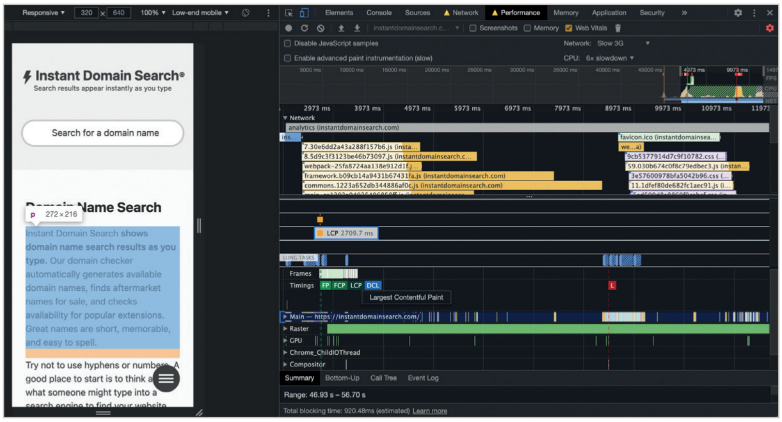
Next.js lets us build our pages and components in React and TypeScript. When paired with VS Code, the development experience is amazing. Next.js generally works by transforming React components into static HTML and CSS. This way, the initial content can be served from a CDN, and then Next can "hydrate" the page to make elements dynamic. Once the page is hydrated, our site turns into a single page app where people can search for and generate domain names. We do not rely on Next.js to do much server-side work; the majority of our content is statically exported as HTML, CSS, and JavaScript to be served from a CDN.[5]

When someone starts searching for a domain name, we replace the page content with search results. To make the searches as fast as possible, the front end directly queries our Rust back end, which is heavily optimized for domain lookups and suggestions. Many queries we can answer instantly, but for some TLDs we need to do slower DNS queries, which can take a second or two to resolve. When some of these slower queries resolve, we will update the UI with

4 https://smashed.by/pageexperiencedetails
5 https://smashed.by/statichtmlexport

whatever new information comes in. The results pages are different for everyone, and it can be hard for us to predict exactly how each person experiences the site.

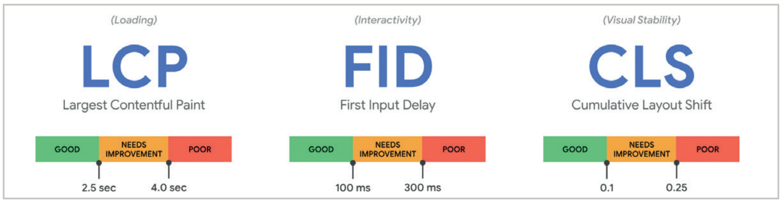
The Chrome DevTools are excellent, and a good place to start when chasing performance issues. The Performance view shows exactly when HTTP requests go out, where the browser spends time evaluating JavaScript, and more:



Screenshot of the Performance pane in Chrome DevTools. We have enabled web vitals which lets us see which element caused the LCP.

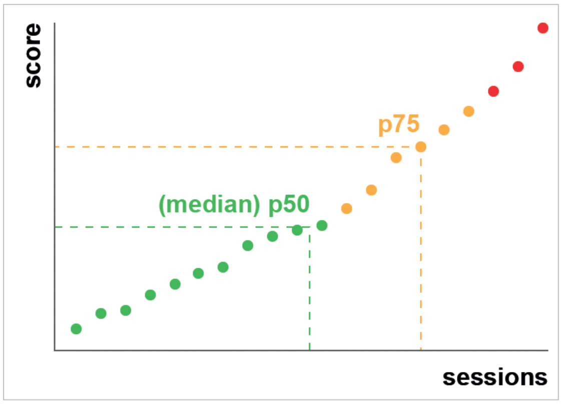
There are three core web vitals metrics that Google will use to help rank sites in their upcoming search algorithm update. Google puts experiences into “Good”, “Needs Improvement”, and “Poor” based on the LCP, FID, and CLS scores real people have on the site:

- **LCP**, or largest contentful paint, defines the time it takes for the largest content element to become visible.
- **FID**, or first input delay, relates to a site’s responsiveness to interaction — the time between a tap, click, or keypress in the interface and the response from the page.
- **CLS**, or cumulative layout shift, tracks how elements move or shift on the page absent of actions like a keyboard or click event.



A summary of LCP, FID and CLS. (Image credit: Web Vitals by Philip Walton[6])

Chrome is set up to track these metrics across all logged-in Chrome users, and sends anonymous statistics summarizing a customer's experience on a site back to Google for evaluation. These scores are accessible via the Chrome User Experience Report, and are shown when you inspect a URL with the PageSpeed Insights tool. The scores represent the 75th percentile experience for people visiting that URL over the previous 28 days. This is the number they will use to help rank sites in the update.



The median, also known as the 50th percentile or p50, is shown in green. The 75th percentile, or p75, is shown here in yellow. In this illustration, we show 20 sessions. The 15th worst session is the 75th percentile, and what Google will use to score this site's experience.

A 75th percentile (p75) metric strikes a reasonable balance[7] for performance goals. Taking an average,[8] for example, would hide a lot of bad experiences people have. The median,[9] or 50th percentile (p50), would mean that half of the people using our product were having

6 https://smashed.by/webvitals
7 https://smashed.by/choiceofpercentile
8 https://smashed.by/average
9 https://smashed.by/median

a worse experience. The 95th percentile (p95), on the other hand, is hard to build for as it captures too many extreme outliers on old devices with spotty connections. We feel that scoring based on the 75th percentile is a fair standard to meet.

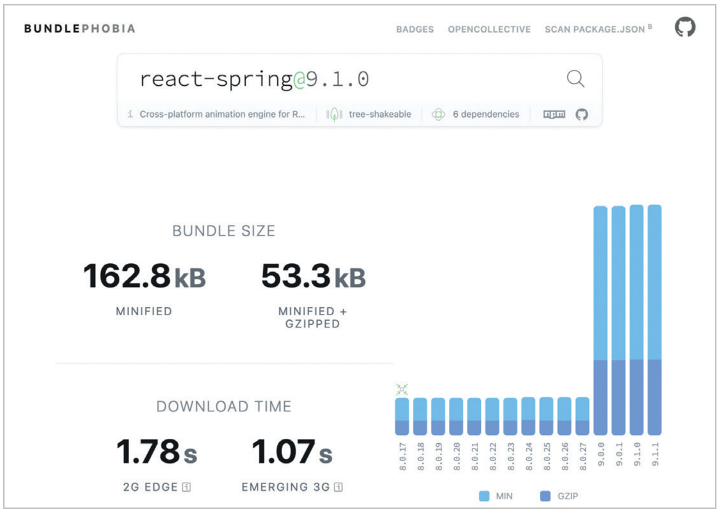
To get our scores under control, we first turned to Lighthouse for some excellent tooling built into Chrome and hosted at *web.dev/measure/*, and at PageSpeed Insights. These tools helped us find some broad technical issues with our site. We saw that the way Next.js was bundling our CSS and slowed our initial rendering time which affected our FID. The first easy win came from an experimental Next.js feature, optimizeCss,[10] which helped improve our general performance score significantly.

Lighthouse also caught a cache misconfiguration that prevented some of our static assets from being served from our CDN. We are hosted on Google Cloud Platform, and the Google Cloud CDN requires that the Cache-Control header contains "public".[11] Next.js does not allow you to configure all of the headers[12] it emits, so we had to override them by placing the Next.js server behind Caddy,[13] a lightweight HTTP proxy server implemented in Go. We also took the opportunity to make sure we were serving what we could with the relatively new `stale-while-revalidate` support in modern browsers which allows the CDN to fetch content from the origin (our Next.js server) asynchronously in the background.

It's easy — maybe too easy — to add almost anything you need to your product from npm. It doesn't take long for bundle sizes to grow. Big bundles take longer to download on slow networks, and the 75th percentile mobile phone will spend a lot of time blocking the main UI thread while it tries to make sense of all the code it just downloaded. We liked BundlePhobia which is a free tool that shows how many dependencies and bytes an npm package will add to your

10 https://smashed.by/optimizecss
11 https://smashed.by/cacheability
12 https://smashed.by/cachecontrolpublic
13 https://caddyserver.com/

bundle. This led us to eliminate or replace a number of react-spring powered animations with simpler CSS transitions.

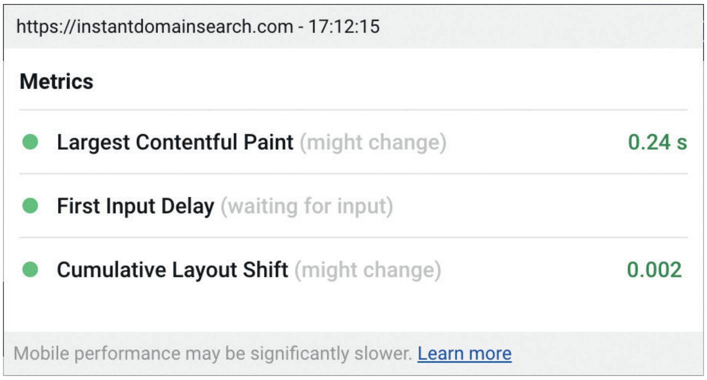


We used BundlePhobia to help track down big dependencies that we could live without.

Through the use of BundlePhobia and Lighthouse, we found that third-party error logging and analytics software contributed significantly to our bundle size and load time. We removed and replaced these tools with our own client-side logging that take advantage of modern browser APIs like sendBeacon and ping. We send logging and analytics to our own Google BigQuery infrastructure where we can answer the questions we care about in more detail than any of the off-the-shelf tools could provide. This also eliminates a number of third-party cookies and gives us far more control over how and when we send logging data from clients.

Our CLS score still had the most room for improvement. The way Google calculates CLS is complicated — you're given a maximum "session window" with a 1-second gap, capped at 5 seconds from the initial page load, or from a keyboard or click interaction, to finish moving things around the site. If you're interested in reading more deeply into this, here's a great guide on the topic at the link in the footnote.[14] This penalizes many types of overlays and pop-ups that appear just after you land on a site: for instance, ads that shift content around, or upsells that might appear when you start scrolling past ads to reach content. A web.dev article by Milica Mihajlija and Philip Walton provides an excellent explanation.[15]

We are fundamentally opposed to this kind of digital clutter, so we were surprised to see how much improvement Google insisted we make. Chrome has a built-in Web Vitals overlay[16] that you can access by using the Command Menu[17] to "Show Core Web Vitals overlay". To see exactly which elements Chrome considers in its CLS calculation, we found the Chrome Web Vitals extension's "Console Logging" option in settings more helpful. Once enabled, this plugin shows your LCP, FID, and CLS scores for the current page. From the console, you can see exactly which elements on the page are connected to these scores. Our CLS scores had the most room for improvement.


https://instantdomainsearch.com - 17:12:15

Metrics

●	**Largest Contentful Paint** (might change)	0.24 s
●	**First Input Delay** (waiting for input)	
●	**Cumulative Layout Shift** (might change)	0.002

Mobile performance may be significantly slower. Learn more

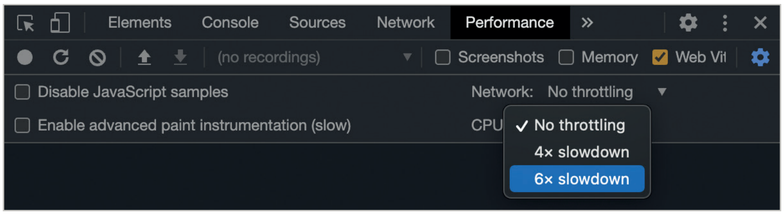
The Chrome Web Vitals extension shows how Chrome scores the current page on their web vitals metrics. Some of this functionality will be built into Chrome 90.

Of the three metrics, CLS is the only one that accumulates as you interact with a page. The Web Vitals extension has a logging op-

14 https://smashed.by/completecls
15 https://web.dev/cls/
16 https://smashed.by/cwv
17 https://smashed.by/commandmenu

tion that will show exactly which elements cause CLS while you are interacting with a product. Watch how the CLS metrics add when we scroll on Smashing Magazine's home page.[18]

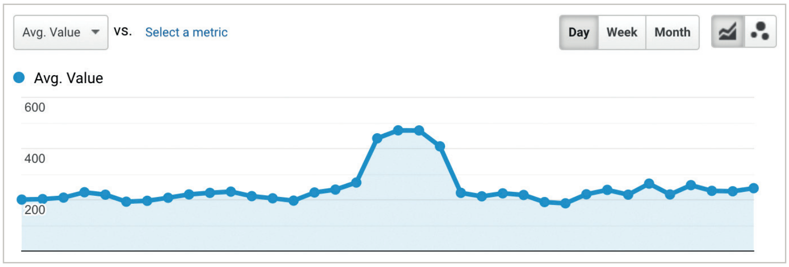
Google will continue to adjust how it calculates CLS over time, so it's important to stay informed by following Google's web development blog. When using tools like the Chrome Web Vitals extension, it's important to enable CPU and network throttling to get a more realistic experience. You can do that with the developer tools by simulating a mobile CPU.[19]



Disable JavaScript samples
Enable advanced paint instrumentation (slow)
Network: No throttling
CPU: ✓ No throttling / 4× slowdown / 6× slowdown

It's important to simulate a slower CPU and network connection when looking for web vitals issues on your site.

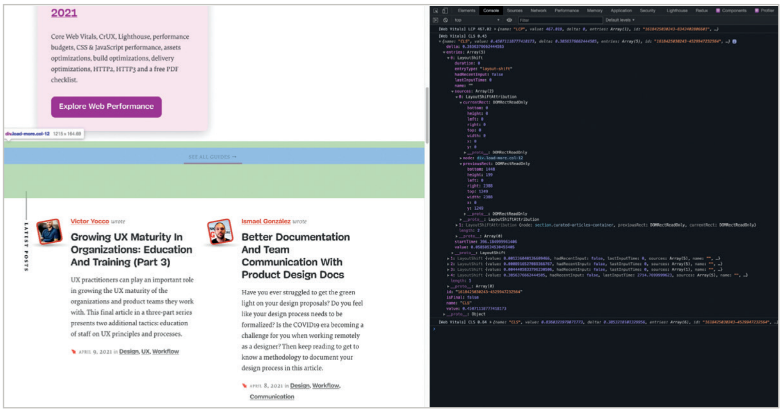
The best way to track progress from one deploy to the next is to measure page experiences the same way Google does. If you have Google Analytics set up, an easy way to do this is to install Google's web-vitals module and hook it up to Google Analytics. This provides a rough measure of your progress and makes it visible in a Google Analytics dashboard.



Google Analytics can show an average value of your web vitals scores.

18 https://smashed.by/smashingscrolling
19 https://smashed.by/mobilecpu

This is where we hit a wall. We could see our CLS score, and while we'd improved it significantly, we still had work to do. Our CLS score was roughly 0.23 and we needed to get this below 0.1 — and preferably down to zero. At this point, though, we couldn't find something that told us exactly which components on which pages were still affecting the score. We could see that Chrome exposed a lot of detail in its core web vitals tools, but that the logging aggregators threw away the most important part: exactly which page element caused the problem.


2021
Core Web Vitals, CrUX, Lighthouse, performance budgets, CSS & JavaScript performance, assets optimizations, build optimizations, delivery optimizations, HTTP2, HTTP3 and a free PDF checklist.
Explore Web Performance
div.load-more.col-12 1218 × 164.88
SEE ALL GUIDES →
LATEST POSTS
Victor Yocco wrote
Growing UX Maturity In Organizations: Education And Training (Part 3)
UX practitioners can play an important role in growing the UX maturity of the organizations and product teams they work with. This final article in a three-part series presents two additional tactics: education of staff on UX principles and processes.
April 9, 2021 in Design, UX, Workflow
Ismael González wrote
Better Documentation And Team Communication With Product Design Docs
Have you ever struggled to get the green light on your design proposals? Do you feel like your design process needs to be formalized? Is the COVID19 era becoming a challenge for you when working remotely as a designer? Then keep reading to get to know a methodology to document your design process in this article.
April 8, 2021 in Design, Workflow, Communication
Elements Console Sources Network Performance Memory Application Security Lighthouse Redux Components Profiler
top
Filter
Default levels

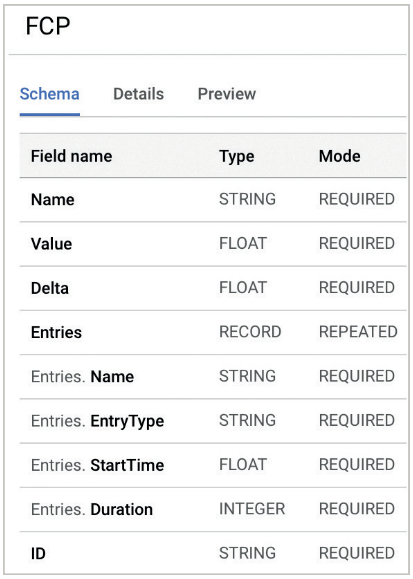
```
[Web Vitals] LCP 467.02 ▸{name: "LCP", value: 467.019, delta: 0, entries: Array(1), id: "1618425030243-8342402006601", …}
[Web Vitals] CLS 0.45
▾{name: "CLS", value: 0.45071118777418173, delta: 0.3856376662444585, entries: Array(5), id: "1618425030243-4529947232564", …}
   delta: 0.3856376662444585
  ▾entries: Array(5)
   ▾0: LayoutShift
      duration: 0
      entryType: "layout-shift"
      hadRecentInput: false
      lastInputTime: 0
      name: ""
     ▾sources: Array(2)
      ▾0: LayoutShiftAttribution
       ▾currentRect: DOMRectReadOnly
          bottom: 0
          height: 0
          left: 0
          right: 0
          top: 0
          width: 0
          x: 0
          y: 0
         ▸__proto__: DOMRectReadOnly
       ▸node: div.load-more.col-12
       ▾previousRect: DOMRectReadOnly
          bottom: 1448
          height: 199
          left: 0
          right: 2388
          top: 1249
          width: 2388
          x: 0
          y: 1249
         ▸__proto__: DOMRectReadOnly
       ▸__proto__: LayoutShiftAttribution
      ▸1: LayoutShiftAttribution {node: section.curated-articles-container, previousRect: DOMRectReadOnly, currentRect: DOMRectReadOnly}
       length: 2
      ▸__proto__: Array(0)
      startTime: 396.184999961406
      value: 0.05850524530455405
     ▸__proto__: LayoutShift
   ▸1: LayoutShift {value: 0.001236040136609466, hadRecentInput: false, lastInputTime: 0, sources: Array(5), name: "", …}
   ▸2: LayoutShift {value: 0.000891652708936767, hadRecentInput: false, lastInputTime: 0, sources: Array(5), name: "", …}
   ▸3: LayoutShift {value: 0.004440582379623050, hadRecentInput: false, lastInputTime: 0, sources: Array(1), name: "", …}
   ▸4: LayoutShift {value: 0.3856376662444585, hadRecentInput: false, lastInputTime: 2714.7699999623, sources: Array(5), name: "", …}
    length: 5
   ▸__proto__: Array(0)
   id: "1618425030243-4529947232564"
   isFinal: false
   name: "CLS"
   value: 0.45071118777418173
  ▸__proto__: Object
[Web Vitals] CLS 0.84 ▸{name: "CLS", value: 0.8360321979871773, delta: 0.3853210181329956, entries: Array(6), id: "1618425030243-4529947232564", …
>
```

This shows exactly which elements contribute to your CLS score.

To capture all of the detail we need, we built a serverless function to capture web vitals data from browsers. Since we don't need to run real-time queries on the data, we stream it into Google BigQuery's streaming API for storage.[20] This architecture means we can inexpensively capture about as many data points as we can generate.

After learning some lessons while working with web vitals and BigQuery, we decided to bundle up this functionality and release these tools as open-source at *vitals.dev*.

20 https://smashed.by/streamingapi


FCP

Field name	Type	Mode
Name	STRING	REQUIRED
Value	FLOAT	REQUIRED
Delta	FLOAT	REQUIRED
Entries	RECORD	REPEATED
Entries. Name	STRING	REQUIRED
Entries. EntryType	STRING	REQUIRED
Entries. StartTime	FLOAT	REQUIRED
Entries. Duration	INTEGER	REQUIRED
ID	STRING	REQUIRED

One of our BigQuery schemas.

Using Instant Vitals is a quick way to get started tracking your web vitals scores in BigQuery. Here's an example of a BigQuery table schema that we create:

Integrating with Instant Vitals is easy. You can get started by integrating with the client library to send data to your back-end or serverless function:

```
Unset

import { init } from "@instantdomain/vitals-client";
init({ endpoint: "/api/web-vitals" });
```

Then, on your server you can integrate with the server library to complete the circuit:

```
Unset

import fs from "fs";
import { init, streamVitals } from "@instantdomain/vitals-
server";
// Google libraries require service key as path to file
const GOOGLE_SERVICE_KEY = process.env.GOOGLE_SERVICE_KEY;
process.env.GOOGLE_APPLICATION_CREDENTIALS = "/tmp/goog_
creds";
fs.writeFileSync(
  process.env.GOOGLE_APPLICATION_CREDENTIALS,
  GOOGLE_SERVICE_KEY
);
const DATASET_ID = "web_vitals";
init({ datasetId: DATASET_ID }).then().catch(console.error);
// Request handler
export default async (req, res) => {
  const body = JSON.parse(req.body);
  await streamVitals(body, body.name);
  res.status(200).end();
};
```

Simply call `streamVitals` with the body of the request and the name of the metric to send the metric to BigQuery. The library will handle creating the dataset and tables for you.

After collecting a day's worth of data, we ran a query like this one:

```
Unset

SELECT
  '<project_name>.web_vitals.CLS'.Value,
  Node
FROM
  '<project_name>.web_vitals.CLS'
JOIN
  UNNEST(Entries) AS Entry
JOIN
  UNNEST(Entry.Sources)
WHERE
  Node != ""
ORDER BY
  value
LIMIT
  10
```

This query produces results like this:

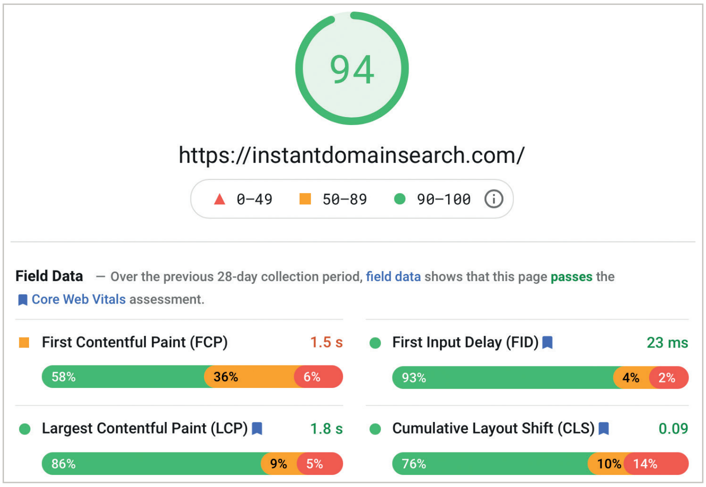
VALUE	NODE
4.6045324800736724E-4	/html/body/div[1]/main/div/div/div[2]/div/div/blockquote
7.183070668914928E-4	/html/body/div[1]/header/div/div/header/div
0.031002668277977697	/html/body/div[1]/footer
0.035830703317463526	/html/body/div[1]/main/div/div/div[2]
0.035830703317463526	/html/body/div[1]/footer
0.035830703317463526	/html/body/div[1]/main/div/div/div[2]
0.035830703317463526	/html/body/div[1]/main/div/div/div[2]
0.035830703317463526	/html/body/div[1]/footer
0.035830703317463526	/html/body/div[1]/footer
0.03988482067913317	/html/body/div[1]/footer

This shows us which elements on which pages have the most impact on CLS. It created a punch list for our team to investigate and fix. On Instant Domain Search, it turns out that slow or bad mobile connections will take more than 500ms to load some of our search results. One of the worst contributors to CLS for these users was actually our footer.

The layout shift score is calculated as a function of the size of the element moving, and how far it goes. In our search results view, if a device takes more than a certain amount of time to receive and render search results, the results view would collapse to a zero-height, bringing the footer into view. When the results come in, they push the footer back to the bottom of the page. A big DOM element moving this far added a lot to our CLS score. To work through this properly, we needed to restructure the way the search results are collected and rendered. We decided to just remove the footer in the search results view as a quick hack that would stop it from bouncing around on slow connections.

We now review this report regularly to track how we are improving — and use it to fight declining results as we move forward. We have witnessed the value of extra attention to newly launched features and products on our site and have operationalized consistent checks to be sure core vitals are acting in favor of our ranking. We hope that by sharing Instant Vitals we can help other developers tackle their core web vitals scores too.[21]

Google provides excellent performance tools built into Chrome, and we used them to find and fix a number of performance issues.



Field Data — Over the previous 28-day collection period, field data shows that this page passes the Core Web Vitals assessment.

Metric	Value	Good	Needs improvement	Poor
First Contentful Paint (FCP)	1.5 s	58%	36%	6%
First Input Delay (FID)	23 ms	93%	4%	2%
Largest Contentful Paint (LCP)	1.8 s	86%	9%	5%
Cumulative Layout Shift (CLS)	0.09	76%	10%	14%

Google PageSpeed Insights shows that we now pass the core web vitals assessment.

We learned that the field data provided by Google offered a good summary of our p75 progress, but did not have actionable detail. We

21 https://smashed.by/instantvitals

needed to find out exactly which DOM elements were causing layout shifts and input delays. Once we started collecting our own field data — with XPath queries — we were able to identify specific opportunities to improve everyone's experience on our site. With some effort, we brought our real-world core web vitals field scores down into an acceptable range in preparation for June's page experience update. We're happy to see these numbers go down and to the right!

Instant Domain Search Key Takeaways

Use the right tools to track your progress and to identify specific areas for improvement.

Despite performance being a core feature of the company, Instant Domain Search discovered that their core web vital scores were not good for a lot of people. To rectify this issue, they started by tracking their core web vitals scores over time. This helped them to identify which areas of their website needed improvement. They then used a variety of tools to identify which elements on their page were causing CLS and FID issues.

Once they had identified the problem areas, they were able to take steps to fix them. For example, they optimized their images, reduced the number of third-party scripts, and used a caching plugin.

As a result of the efforts, Instant Domain Search was able to improve their core web vitals scores significantly, gaining worthwhile insights into improving the website's user experience.

1
2
3
1
2
3
CAPABILITIES

CAPABILITIES

Introduction

For many years, the powerful capabilities that enabled developers to build great user experiences on mobile devices were restricted to native applications. This meant the features mobile developers needed, such as offline functionality, installability, and notifications just weren't available on the web. This forced developers to invest in building native apps even if their teams would have preferred to just extend their existing web experience.

Bridging the gap between native capabilities and the modern web is made possible through a combination of progressive web application (PWA) technologies and APIs that can access device and operating system features more readily. In this section, we will discuss PWAs and device APIs and then look at some case studies where adding these capabilities helped businesses.[1]

Web Apps, Native Apps, and PWAs

Browser vendors have been working to bridge the native–web gap for some time now, allowing businesses to enjoy the universality of the web with many of the features that were once only available to native platforms.. The web app manifest enables offering an "installed" experience via the web. Service workers allow the delivery of a reliable offline experience that users expect from an installed experience. WebAssembly can get you closer to the metal for complex code. There are plenty more.

A more complete list of features that were earlier out of reach for web apps but have become accessible are as follows:

1 https://smashed.by/deviceapis

1. You can access the camera and microphone on your device through your web app, capture or record audio and video,[2] use them for real-time communication[3] or shape detection.[4]

2. Web apps can detect your location using GPS to serve you location-specific results or redirect to a country-specific website. Apps are also able to detect the position and motion of the device.

3. Web apps support different types of input like touch gestures and clipboard input, and can even analyze them for speech or music recognition[5]

4. Web apps support native app behaviors like web push notifications, similar to push notifications sent by native apps but available via desktop or mobile web. This feature brings the benefit of real-time, personalized communication to web apps. Additionally, native app functions like task scheduling (calendar access) and permissions for features like GPS, camera, and microphone are also supported by web apps.

5. Web apps can be built as progressive web apps installed and accessed like a native app. They are capable of providing a seamless experience and integrating with device features.

Furthermore, it is still as easy to start using a new web app as it was earlier. You can use web apps without going through the download, install, set up, and upgrade process. Thus, web apps provide better reach to businesses that use them.

Native apps are still preferred by users, especially on mobile phones, as they sync with the device ecosystem. They can be easily searched

2 https://www.awesomescreenshot.com/
3 https://zoom.us/
4 https://lens.google/
5 https://www.shazam.com/

and accessed on the device and pinned to a specific location like a taskbar, dock, or home screen. Native apps can access system resources like file systems, contacts, location, and Bluetooth with manageable permissions. They can also work offline and usually have an inbuilt mechanism to sync data when online. However, native apps need to be downloaded and installed from a specific app store, which can be daunting if you are not going to use the app frequently and tend to forget your app store password easily.

PROGRESSIVE WEB APPLICATIONS (PWAS)

Mobile web has generated more traffic than desktop web over the last few years.[6] Web apps built for the desktop do not respond well to mobile device constraints. Introduced by Google in 2016, progressive web applications (PWAs) try to combine the capabilities of native apps with the reach of web apps, thus enabling a best-of-both-worlds experience, especially on mobile. For a web app to be progressive, we enhance it with modern APIs to make it more capable, reliable, and installable.

PWAs are an attractive option for both app developers and users, and have the potential to replace native apps in the future because:

1. **Easy distribution:** They offer easier distribution options. You don't go through the native app store review and release process and can fully control the release and update cycle for the app. The same PWA can be used by all web users, irrespective of the mobile or desktop device that they are on.

2. **Ease of use:** Users can open and start using it immediately without going through the app store's search, download, install, and open process. The level of loyalty and commitment expected from the user is higher in the case of native apps.

6 https://smashed.by/trafficstats

3. **Native app behavior:** PWAs can emulate native app behavior by allowing for offline browsing and the ability to be accessed from the home screen. You can combine this with better performance through caching and predictive prefetching of resources.

4. **Smaller footprint:** They have a smaller footprint on device storage as they are mostly served online. The caching mechanism can be designed to balance offline capabilities, data consumption, and storage requirements.

5. **Device access:** Depending on the platform support, PWAs can support features like web push notifications, file system access, media controls, app badging, full clipboard, and so on. Many of these features are already available through modern APIs (discussed later), while others are under construction.

PWAs can drive business success because they put the user first by providing a fast, installable, reliable, and engaging (FIRE) experience.[7] PWAs can result in users spending more time on the app, reduced bounce rates, and more returning visitors, thereby yielding a higher conversion rate. Thus, adding PWA capabilities also allows businesses to tap into the growing smartphone market. One of the greatest success stories of PWA's has been the Twitter Lite PWA.[8] Twitter saw a reduction in data consumption as well as time-to-interactive with the PWA. With all Twitter mobile web users migrating to PWA in 2017, there was a substantial increase in usage and engagement.

PWAs can result in users spending more time on the app, reduced bounce rates, and more returning visitors, thereby yielding a higher conversion rate.

Note that there are use cases where native applications are and will still be most suitable and preferred. While distributing native apps

7 https://smashed.by/businesssuccess
8 https://smashed.by/twitterlitepwa

through app stores may be cumbersome, it does allow you to monetize the app through the app store. Native apps are especially suitable for high fidelity games, which come with in-app purchase options and may have "close-to-the-metal" requirements from the device. PWA's are also not ideal for functions that need access to API's not yet supported by web apps.

PWAS: IMPLEMENTATION GUIDELINES

The initial set of requirements to make a web app installable[9] as a PWA:

1. Should include a web app manifest, a JSON file that tells the browser about your PWA and how it should behave when installed on the user's desktop or mobile device. For example, app name, icons, colors, shortcuts used by the PWA are all defined in the web app manifest.

2. Should register a service worker to allow the app to work offline or under poor network conditions. This improves the reliability of the web application and ensures smooth and steady user interactions.

3. Should be served over HTTPS for security.

PWA support is not uniform across browsers and platforms. Here are some critical differences in how different platforms support progressive features.

1. WebAPK's: Available mainly through Chrome and Samsung browsers on Android devices, WebAPK's[10] are Android APKS (Android Packages) corresponding to a PWA installed on the device. Chrome automatically generates and installs the WebAPK for the PWA.

9 https://smashed.by/installableexperience
10 https://smashed.by/appmanifest

2. Trusted Web Activities: Available through the Play Store on Android devices, TWAs[11] provide a new way of accessing web app (including PWA) content on your Android app using a protocol based on Chrome Custom Tabs.[12] You can also launch a PWA from a TWA if you have an excellent PWA and wish to bring the same user experience to your Android app.

3. WKWebView: This is the iOS/macOS equivalent of TWA available through the Apple Appstore. WKWebView is a "platform-native view that you use to incorporate web content seamlessly into your app's UI."[13] They allow you to reuse parts of your website inside your app.

Additionally, PWAs are accessible through other platforms/browser combinations, but support for different capabilities may vary. Apple/iOS devices especially have some limitations when it comes to PWA's. Noteworthy among these are

- Install prompt is not automatically available through any browser on iOS, but instead, users can use the ***Add to Home Screen*** option available through Safari.
- Web push notifications are not supported. You can include native push notifications through the use of hybrid platforms like Cordova.[14]
- iOS PWAs may not be able to preserve state when switching between apps. The PWA is reloaded every time and may show a blank image. One can prevent the blank screen by linking splash startup images for all resolutions.

Despite the few limitations discussed, PWA's provide the base for introducing web app capabilities that are at par with native apps. Let us now look at these capabilities and the efforts to bring them to web apps, including PWA's.

11 https://smashed.by/twa
12 https://smashed.by/customtabs
13 https://smashed.by/wkwebview
14 https://cordova.apache.org/

Accessing Hardware Devices with Device APIs

Modern web apps are getting increasingly sophisticated when it comes to accessing device-level hardware and software. Device APIs make such access possible. Device APIs are browser-supported APIs that allow us to access a wide range of devices directly through our web apps without a third-party library. Thus, access to features like barcode and QR code scanners, phone contacts, Bluetooth, GPS, etc., is now available to web apps. Therefore, device APIs allow us to build faster solutions for many different use cases within our web apps.

Following are some compelling examples for using device APIs to build native app capabilities in web apps.

- Shape-detection API allows apps to use the highly optimized shape detection features already available in Android, iOS, and macOS.[15] It currently supports face detection and barcode detection. Everyday use cases include face tagging in social networking apps and QR code recognition for social distancing/mobile payment apps.

- Web NFC brings near field communication (NFC) technology to Chrome on Android.[16] NFC is the technology that powers Apple Pay and Android Pay on phones by allowing interaction with NFC tags. Web NFC enables this communication through web apps. Currently, you can use Web NFC in situations where NFC Data Exchange Format (NDEF) is supported.

- Contact picker API provides access to the contacts list on the mobile device.[17] WhatsApp is probably the best example of how access to phone contacts contributed to the app's success. Contact picker API allows web apps to access limited details of selected entries from the contact list. This feature would be helpful to web apps that provide communication and social networking functions.

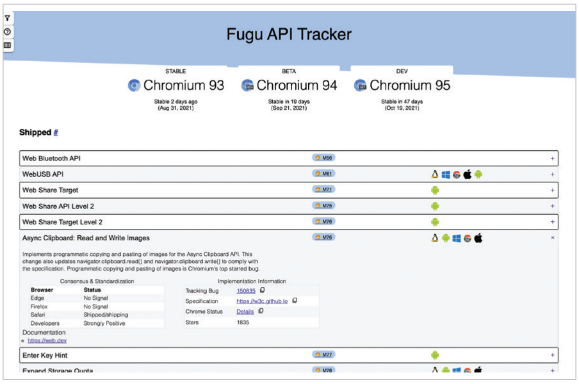
15 https://smashed.by/shapedetection
16 https://smashed.by/nfc
17 https://smashed.by/contactpicker

The Capabilities Project: Fugu

We have described a few device APIs in the previous section, but they are just the tip of the iceberg. The need for new and enhanced APIs keeps growing as we try to enable web apps to do anything that native apps can do.

Project Fugu, a Chromium project, aims to achieve this by providing developers a way to identify opportunities for enhancement and then discuss, develop and introduce them on the open web.[18] The project's mission is to "enable new experiences on the web while preserving the web's core benefits of security, low-friction, and cross-platform delivery."

The project is open to all Chromium contributors and organizations, including Microsoft, Intel, Samsung, and Google. The Fugu API tracker maintains the complete list of capabilities that are part of this project with the current status, description, and other details, including browser support:[19]



Fugu API Tracker

STABLE	BETA	DEV
Chromium 93	Chromium 94	Chromium 95
Stable 2 days ago (Aug 31, 2021)	Stable in 19 days (Sep 21, 2021)	Stable in 47 days (Oct 19, 2021)

Shipped #

Web Bluetooth API	M56	+
WebUSB API	M61	+
Web Share Target	M71	+
Web Share API Level 2	M75	+
Web Share Target Level 2	M76	+
Async Clipboard: Read and Write Images	M76	×

Implements programmatic copying and pasting of images for the Async Clipboard API. This change also updates navigator.clipboard.read() and navigator.clipboard.write() to comply with the specification. Programmatic copying and pasting of images is Chromium's top starred bug.

Consensus & Standardization

Browser	Status
Edge	No Signal
Firefox	No Signal
Safari	Shipped/shipping
Developers	Strongly Positive

Implementation Information

Tracking Bug	150835
Specification	https://w3c.github.io
Chrome Status	Details
Stars	1835

Documentation
- https://web.dev

Enter Key Hint	M77	+
Expand Storage Quota	M78	+

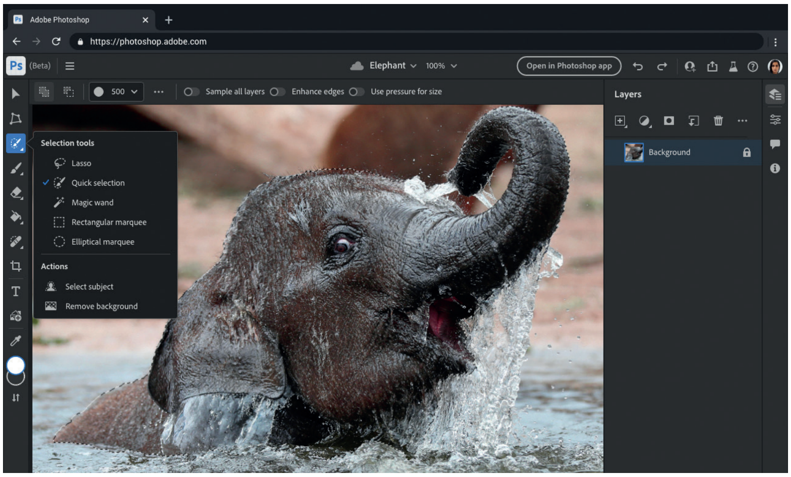
With this background on capabilities, we can move on to some case studies of organizations that have enriched their web applications with additional capabilities and the benefits gained.

18 https://smashed.by/fugus
19 https://fugu-tracker.web.app/

Photoshop's Journey to the Web

By Nabeel Al-Shamma & Thomas Nattestad

Over the last three years, Chrome has been working to empower web applications that want to push the boundaries of what's possible in the browser. One such web application has been Photoshop. The idea of running software as complex as Photoshop directly in the browser would have been hard to imagine even just a few years ago. However, by using various new web technologies, Adobe has now brought a public beta of Photoshop to the web.[1]



In this write-up, we'd like to share for the first time the details of how our collaboration with Chrome is extending Photoshop to the web. You can use all the APIs Adobe used and more in your own apps as well. Be sure to check out our web capabilities related blog posts[2] for inspiration and watch our API tracker[3] for the latest and greatest we're working on.

1 The original version of this case study was published in October 2021: https://smashed.by/psweb
2 https://smashed.by/capabilities
3 https://smashed.by/fugus

Why Photoshop Came to the Web

As the web has evolved, one thing that hasn't changed are the core advantages that websites and web apps offer over platform-specific applications. These advantages include many unique capabilities such as being linkable, ephemeral, and universal, but they boil down to enabling simple access, easy sharing, and great collaboration.

The simple power of a URL is that anyone can click it and instantly access it. All you need is a browser. There is no need to install an application or worry about what operating system you are running on. For web applications, that means users can have access to the application and their documents and comments. This makes the web the ideal collaboration platform, something that is becoming more and more essential to creative and marketing teams.

Google Docs was a pioneer of this simplified access. Most of us know how easy it is to start a document, send the link to someone, and immediately jump into not only the application, but the specific document or comment as well. Since then, a plethora of amazing applications, such as those we've shown off in the past,[4] have adopted this model, and now Photoshop too will benefit.

How Photoshop Came to the Web

The web started out as a platform only suited for documents, but has grown dramatically throughout its history. Early apps like Gmail showed that more complex interactivity and applications were at least possible. Since then, we've seen impressive codevelopment where web apps push the boundaries of what's possible, and

4 https://smashed.by/adobespark

browser vendors respond by further expanding web capabilities. The latest iteration of this virtuous loop is what has enabled Photoshop on the web.

Adobe previously brought Spark[5] and Lightroom[6] to the web and had been interested in bringing Photoshop to the web for many years. However, they were blocked by the performance limitations of JavaScript, the absence of a good compilation target for their code, and the lack of web capabilities. Read on to learn what Chrome built in the browser to solve these problems.

WebAssembly Porting with Emscripten

WebAssembly and its C++ toolchain Emscripten[7] have been the key to unlocking Photoshop's ability to come to the web, as it meant that Adobe would not have to start from scratch, but could leverage its existing Photoshop codebase. WebAssembly (Wasm) is a portable binary instruction set shipping in all browsers that was designed as a compilation target for programming languages. This means that applications such as Photoshop that are written in C++ can be ported directly to the web without requiring a rewrite in JavaScript. To get started porting yourself, check out the full Emscripten documentation,[8] or follow this guided example of how to port a library.[9]

Emscripten is a fully featured toolchain that not only helps you compile your C++ to Wasm, but provides a translation layer that turns POSIX API calls into web API calls and even converts OpenGL into WebGL. For example, you can port applications that reference the local filesystem and Emscripten will provide an emulated file system to maintain functionality.[10]

5 https://smashed.by/spark
6 https://smashed.by/lightroom
7 https://emscripten.org/
8 https://smashed.by/emscriptendocs
9 https://smashed.by/emscripting
10 https://smashed.by/emscripten

Emscripten has been capable of bringing most parts of Photoshop to the web for a while, but it wasn't fast enough. We have continually worked with Adobe to figure out where bottlenecks are and improve Emscripten. Photoshop depends on multithreading. Bringing dynamic multithreading[11] to WebAssembly was a critical requirement

Also, exception-based error handling is very common in C++, but wasn't well supported in Emscripten and WebAssembly. We have worked with the WebAssembly Community Group in[12] the W3C to improve the WebAssembly standard and the tooling around it to bring C++ exceptions to WebAssembly.

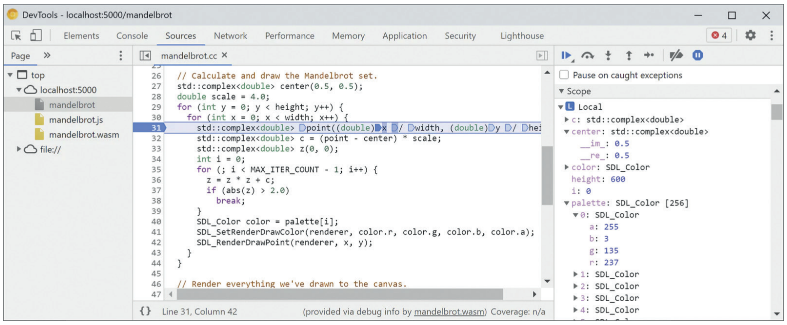
Emscripten doesn't just work on large applications but also lets you port libraries or smaller projects! For example, you can see how you can compile the popular OpenCV library[13] to the web through Emscripten.

Lastly, WebAssembly offers advanced performance primitives such as SIMD instructions,[14] which dramatically improve your web app performance. For example, Halide is essential to Adobe's performance, and here SIMD provides a 3–4× speed-up on average and in some cases an 80–160× speedup.[15]

WebAssembly Debugging

No large project can be successfully completed without the appropriate tools for the job, and it's for this reason that the Chrome team developed full-featured WebAssembly debugging support. It provides support for stepping through the source code, setting breakpoints and pausing on exceptions, variable inspection with rich type support, and even basic support for evaluation in the DevTools console!

11 https://smashed.by/pthreads
12 https://smashed.by/w3webassembly
13 https://smashed.by/opencv
14 https://smashed.by/simd
15 https://halide-lang.org/



DevTools - localhost:5000/mandelbrot

```
// Calculate and draw the Mandelbrot set.
std::complex<double> center(0.5, 0.5);
double scale = 4.0;
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    std::complex<double> point((double)x / width, (double)y / hei
    std::complex<double> c = (point - center) * scale;
    std::complex<double> z(0, 0);
    int i = 0;
    for (; i < MAX_ITER_COUNT - 1; i++) {
      z = z * z + c;
      if (abs(z) > 2.0)
        break;
    }
    SDL_Color color = palette[i];
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);
    SDL_RenderDrawPoint(renderer, x, y);
  }
}

// Render everything we've drawn to the canvas.
```

Line 31, Column 42 (provided via debug info by mandelbrot.wasm) Coverage: n/a

```
Local
  c: std::complex<double>
  center: std::complex<double>
    __im_: 0.5
    __re_: 0.5
  color: SDL_Color
  height: 600
  i: 0
  palette: SDL_Color [256]
    0: SDL_Color
      a: 255
      b: 3
      g: 135
      r: 237
    1: SDL_Color
    2: SDL_Color
    3: SDL_Color
    4: SDL_Color
```

Be sure to check out the authoritative guide on how to utilize WebAssembly Debugging.[16]

High-Performance Storage

Given how large Photoshop documents can be, a critical need for Photoshop is the ability to dynamically move data from on-disk to in-memory as the user pans around. On other platforms, this is accomplished usually through memory mapping via mmap,[17] but this hasn't been performantly possible on the web – that is until origin private file system access handles were developed and implemented as an origin trial. You can read how to leverage this new API in the documentation.[18]

P3 Color Space for Canvas

Historically, colors on the web have been specified in the sRGB color space,[19] which is a standard from the mid-1990s, based on the capabilities of cathode-ray tube monitors. Cameras and monitors have come a long way in the intervening quarter-century, and many larger and more capable color spaces have been standardized. One of the most popular modern color spaces is Display P3.[20] Photoshop uses a

16 https://smashed.by/wasmdebugging
17 https://smashed.by/mmap
18 https://smashed.by/accessingfiles
19 https://smashed.by/srgb
20 https://smashed.by/dcip3

Display P3 Canvas[21] to display images more accurately in the browser. In particular, images with bright whites, bright colors, and details in shadows will display as best as possible on modern displays that support Display P3 data. The Display P3 Canvas API is being further built on to enable high dynamic range[22] displays.

Web Components and Lit

Photoshop is a famously large and feature-rich application, with hundreds of UI elements supporting dozens of workflows. The app is built by multiple teams using a variety of tools and development practices, but its disparate parts need to come together into a cohesive, high-performing whole.

To meet this challenge, Adobe turned to Web Components[23] and the Lit library.[24] Photoshop's UI elements come from Adobe's Spectrum Web Components[25] library, a lightweight, performant implementation of the Adobe design system that works with any framework, or no framework at all.

What's more, the entire Photoshop app is built using Lit-based Web Components. Leaning on the browser's built-in component model and Shadow DOM encapsulation, the team found it easy to cleanly integrate a few "islands" of React code provided by other Adobe teams.

Service Worker Caching with Workbox

Service workers act as a programmable local proxy, intercepting network requests and responding with data from the network, long-lived caches, or a mixture of both.

21 https://smashed.by/displayp3
22 https://smashed.by/hdrcanvas
23 https://smashed.by/components
24 https://lit.dev/
25 https://smashed.by/spectrum

As part of the V8[26] team's efforts to improve performance, the first time a service worker responds with a cached WebAssembly response, Chrome generates and stores an optimized version of the code – even for multi-megabyte WebAssembly scripts, which are common in the Photoshop codebase. A similar precompilation takes place when JavaScript is cached[27] by a service worker during its `install` step.[28] In both cases, Chrome is able to load and execute the optimized versions of cached scripts with minimal runtime overhead.

Photoshop on the web takes advantage of this by deploying a service worker that pre-caches many of its JavaScript and WebAssembly scripts. Because the URLs for these scripts are generated at build time, and because the logic of keeping caches up to date can be complex, they turned to a set of libraries maintained by Google called Workbox[29] to generate their service worker as part of their build process.

A Workbox-based service worker along with the V8 engine's script caching led to measurable performance improvements. The specific numbers vary based on the device executing the code, but the team estimates these optimizations decreased the time spent on code initialization by 75%.

What's Next for Adobe on the Web

The launch of the Photoshop beta is just the beginning, and we've got several performance and feature improvements already underway as Photoshop tracks towards full launch after this beta. Adobe isn't stopping with Photoshop and plans to aggressively expand Creative Cloud[30] to the web, making it a primary platform for both creative content creation and collaboration. This will enable

26 https://v8.dev/
27 https://smashed.by/swcaches
28 https://smashed.by/installsw
29 https://smashed.by/workbox
30 https://smashed.by/cc

millions of first-time creators to tell their story and benefit from innovative workflows on the web.

As Adobe continues to push the boundaries of what's possible, the Chrome team will continue our collaboration to drive the web forward for Adobe and the vibrant web developer ecosystem in general. As other browsers also catch up on these modern browser capabilities, we're excited to see Adobe make its products available there as well. Stay tuned for future updates as we continue to push the web forward!

You can learn more about accessing Photoshop on the web (beta) in the Adobe Help Center.[31]

Photoshop Key Takeaways

Complex software such as Photoshop can be made available on the web using the latest web technologies.

Bringing Adobe Photoshop to the web is a major milestone in the world of creative software, and it opens up new possibilities for collaboration and creativity. To port a complex application like Photoshop to the web was no easy task. Adobe had to overcome a number of challenges, including the need to create a high-performance rendering engine and to support a wide range of devices and browsers.

WebAssembly and Emscripten were key to making Photoshop on the web possible. The Chrome team also made significant contributions to WebAssembly debugging and performance. This work helped to ensure that Photoshop on the web would be a smooth and

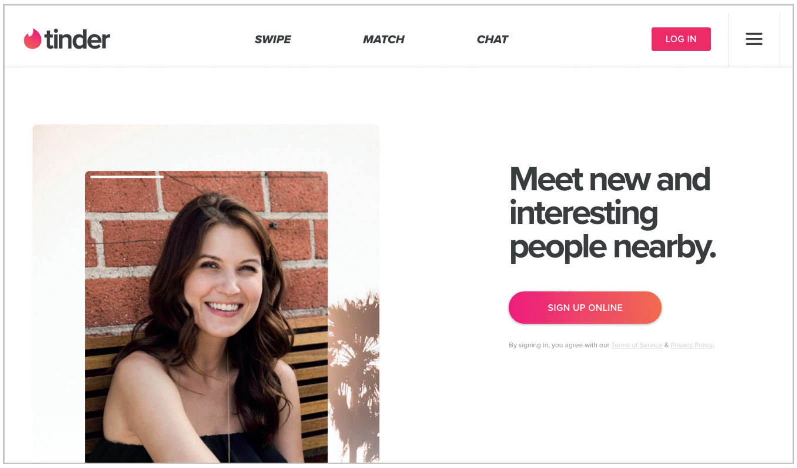
31 https://smashed.by/adobehelp

Building Tinder Online / Tinder PWA

by Roderick Hsiao & Addy Osmani

Tinder, one of the most popular online dating services, is also available on the web platform worldwide.[1] We started this journey when the company had already invested heavily in its native app experience and advanced machine learning technology.

We realize that not all users have the latest mobile devices with big storage and ultra-high network speed to run our native client. The web platform, then, serves a very good purpose – to run mostly anywhere with relatively light required resources.


tinder
SWIPE
MATCH
CHAT
LOG IN
Meet new and interesting people nearby.
SIGN UP ONLINE
By signing in, you agree with our Terms of Service & Privacy Policy.

Tinder Online

Our web team is relatively small, but we started with a great mission: to deliver a performant and smooth web experience using cutting-edge web technology.

1 The original version of this case study was published in February 2018: https://smashed.by/tinderroderick & https://smashed.by/tinderaddy

Architecture

Tinder Online is built using a React–Redux stack.

To build a highly performant and scalable web app, we created our entire user interface using React, with a focus on building reusable components that are then composed within view containers. This flexible composability facilitates rapid iteration and a maintainable codebase.

We use a Redux store to persist our application state. Our state is constructed via ImmutableJS[2] and Normalizr,[3] which allows us to carry out efficient and performant state operations. Memorized selectors make our store access highly performant.

When we first rolled out the experience to target markets, we used a serverless solution. We deployed static assets to Amazon S3 and executed the full app logic client-side. We then moved to an isomorphic Node app to serve more complicated use cases.

We construct the initial application state (feature flags and internationalization) server-side using a simple NodeJS/Express server, and render a highly cacheable app shell with a dehydrated state client-side. The full application logic and data fetching flow is then initialized after rehydrating the application state.

Side effects and asynchronous operations like API requests are handled using Redux Sagas.[4] We persist parts of our state, such as user settings, location, and application settings, with IndexDB[5] in supported browsers, and fall back to `localStorage`[6] when necessary. The persistent store greatly improves the app startup performance and user experience.

2 https://smashed.by/immutablejs
3 https://smashed.by/normalizr
4 https://smashed.by/reduxsaga
5 https://smashed.by/indexdb
6 https://smashed.by/localstorage

The app rendering logic and route settings are centralized and configured on the top level. This abstraction allows us to separate page-level logic from component-level logic, and makes it easy to handle route-level code splitting and various page transition effects. We also develop a proxy React component to implement dynamic JavaScript loading and resource preload for the next route.

The core swiping experience and animation is built on top of React Motion.[7] Internationalization is handled by React Intl.[8] We use React I13n[9] to separate instrumentation logic from UI logic by creating pluggable listeners for different tracking systems.

Performance

Our goal is to provide a seamless experience similar to our native clients for most of our users regardless of network conditions or device hardware restrictions. Therefore, performance is the top priority for us when building features. We focus on two main areas: network performance, and render performance.

NETWORK PERFORMANCE

To support users on slower networks, the web app is optimized to limit network load, document parsing time, and render time. In general, we want to load the critical assets early and fast, and defer the optional resources.

We are able to greatly improve the initial load time by assigning individual resources priorities using link preload and prefetch along with code splitting. We ship minimal resources to the client by implementing code splitting, pre-cache chunks via a service worker,

7 https://smashed.by/reactmotion
8 https://smashed.by/reactintl
9 https://smashed.by/reacti13n

and preload assets for the next anticipated route efficiently. We are using Workbox[10] to control high-level service worker caching strategies for different resources.

The critical render path is optimized by inlining most of our common CSS. We are using Atomic CSS[11] to create highly reusable and compressible style sheets. With Atomic CSS, UI theming and display logic are controlled by React props, making our code easy to share and maintain. Our core CSS, which includes theming, spacing, and responsive styling, is about 10 KB (gzip) for the whole site.

To prevent our bundle size increasing when adding new features, we set performance budgets for all of our resources. The sizes of our JavaScript and CSS bundles are audited on each commit. Setting a good performance bundle forces us to build highly shareable components. We also measure and track performance with tools such as Lighthouse and CSS stats[12] prior to each release. Real-time user monitoring metrics such as load time[13] and paint time (`PerformancePaintTiming`)[14] are collected client-side.

Our source code is compiled and polyfilled by Babel.[15] and generated by Webpack.[16] By exercising bundle analysis, we were able to identify several opportunities for performance optimization strategies, such as code splitting, tree shaking, or selecting alternative libraries. We also use `@babel/preset-env` to include only the subset of polyfills targeting our supported browsers. The total resources needed for the web app is around 3 MB, which is great for users who have limited device storage.

10 https://smashed.by/reacti13n
11 https://acss.io/
12 https://smashed.by/cssstats
13 https://smashed.by/mdnperf
14 https://smashed.by/performancepainttiming
15 https://babeljs.io/
16 https://webpack.github.io/

RENDER PERFORMANCE

We optimize rendering and animation performance by prioritizing JavaScript tasks using `requestIdleCallback`. Non-critical tasks such as instrumentation will be scheduled at idle time. We also ensure that our HTML and CSS are highly optimized and lazy-load offscreen assets via Interaction Observer[17] for fast rendering and smooth performance, even on slower devices.

We use Chrome DevTools and React developer tools heavily to identify performance bottlenecks, such as browser repaint, React rerender or high cost JavaScript operations.

What's Next

Product-wise we are seeing very positive user engagement on the web platform. In terms of technology, there are several areas we would like to focus on:

- Experiment with different approaches for code splitting, such as deferring the registration of Redux reducers and saga handlers.
- Utilize our service worker runtime caching more widely for a better offline experience.
- Offload expensive tasks, such as parsing frequently consumed API responses, to web workers.
- Improve performance among modern browsers by experimenting with new browser primitives such as the network information API.

17 https://smashed.by/reactinviewport

- Experiment with deploying ES module to supported browsers.
- Re-architect Redux store structure to enhance state management.

Tinder Key Takeaways

Tinder can reach more users after building a PWA in addition to the native app.

Tinder had invested heavily in their native app but realized that all users who wished to use their app did not have the latest mobile device and high-speed network required to run it. This drove their decision to build a web app that could run on any mobile device, including those with fewer resources.

Tinder built its PWA on a React/Redux stack. The PWA uses service workers to pre-cache chunks to ensure better network performance of the app. The app also supports push notifications to provide a more native app-like experience. The resultant Tinder PWA at an initial size of 2.8 MB is much lighter than its Android native app, which is 10 MB. A higher rate of swiping and messaging and longer sessions were observed with the PWA when compared to the native app.

Upgrading Ele.me to Progressive Web App

by Xuan Huang

Since the very first experiments that @Vue.js tweeted, we at Ele.me (the biggest food ordering and delivering company in China) have been working on upgrading our mobile website to a progressive web app (PWA).[1] We're proud to ship the world's first PWA exclusively for the Chinese market, but even prouder to collaborate with Google, UC browser and Tencent to push the boundary of web experience and browser support in China.

Multi-page, Vue.js, PWA?

There is a prevailing opinion that only structuring a web app as a single page app (SPA) can we build PWAs that deliver app-like user experience. Popular reference examples such as the following use the SPA model:

1. Twitter Lite: https://smashed.by/twitterlite
2. Flipkart Lite: https://smashed.by/flipkartlite
3. Housing Go: https://smashed.by/housinggo
4. Polymer Shop: https://smashed.by/polymer

However, at Ele.me we've come to appreciate the many advantages of a multi-page app model, and we decided more than a year ago to refactor the mobile site from an AngularJS SPA to a multi-page

1 The original version of this case study was published in May 2017: https://smashed.by/eleme

app (MPA). The most important advantage we see is the isolation and decoupling between pages, which allows us to build different parts of the mobile site as microservices. These services can then be independently iterated, embedded into third-party apps, and even maintained by different teams.

Meanwhile, we also leverage Vue.js[2] to boost our productivity. You may have heard of Vue.js as a rival of React or Angular, but Vue.js's light weight and performance make it also a perfect replacement for a traditional jQuery/Zepto + template engine stack when engineering a multi-page app. We built every component as a single-file component so they can be easily shareable between pages. The declarative-ness plus reactivity Vue.js offered helped us manage both code and data flow. Oh, did I mention that Vue.js is progressive? So things like Vuex or Vue-Router can be incrementally adopted if our site's complexity scales up, like… migrating to SPA again? (Who knows…)

In 2017, PWAs seemed to be all the rage, so we embarked on exploring how far our Vue.js-based multi-page PWAs could actually go.

Implementing PRPL with MPA

I love the PRPL pattern (preload/render/precache/lazy load) because it gives you a high-level abstraction of how to structure and design your own PWA systems. Since we were not rebuilding everything from scratch, we decided to implement PRPL as our migration goal.

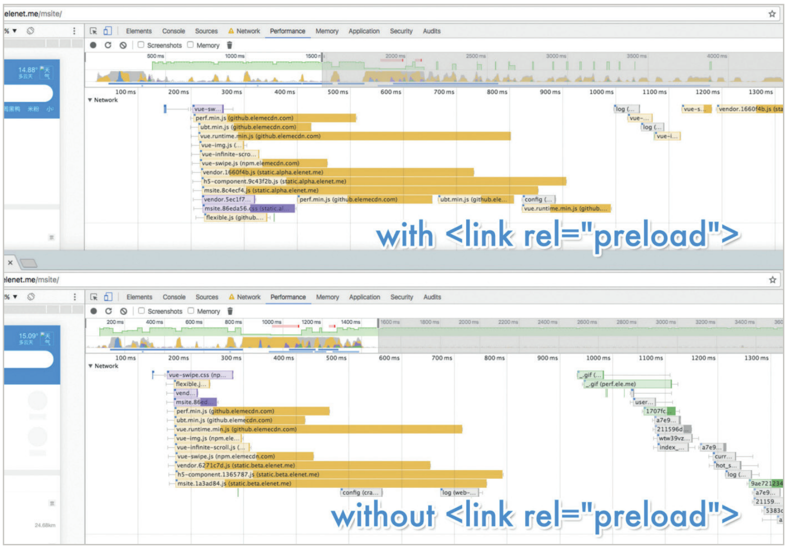
1. PUSH/PRELOAD CRITICAL RESOURCES FOR INITIAL ROUTE

The key of pushing/preloading is to prioritize resources hidden in deep dependency graphs and make the browser's network stack busy as soon as possible. Let's say you have an SPA with code splitting by route, you can push/preload chunks for the current route before the entry chunks

2 http://vuejs.org/

(e.g. Webpack manifest, Router) finish downloading and evaluating. So when the actual fetches happen, they might already be in caches.

Routes in MPAs naturally fetch code for that route only, and tend to have a flattening dependency graph. Most scripts depended by Ele.me are just `<script>` elements, so they can be found and fetched by the good old browser preloader[3] in the early parsing phase without explicit `<link rel="preload">`.



With or without explicit `<link rel="preload">`.

To take advantage of HTTP2 multiplexing, we currently serve all critical resources under a single domain (no more domain sharding), and we are also experimenting on server push.

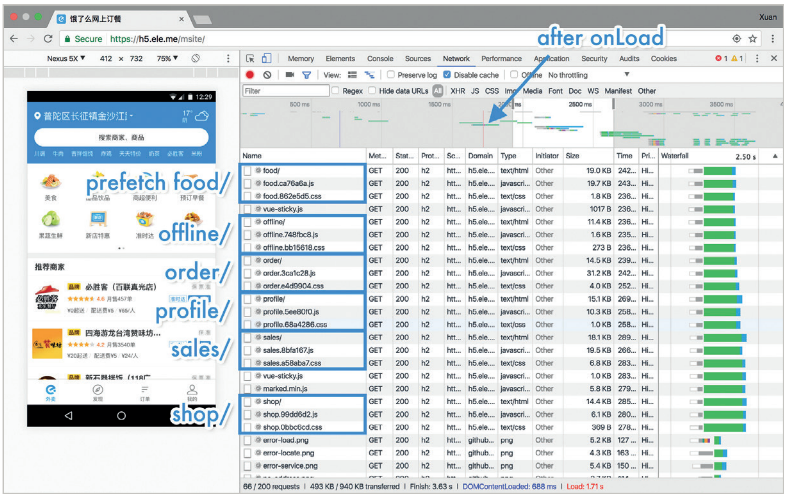
2. RENDER INITIAL ROUTE AND GET IT INTERACTIVE ASAP

This one is essentially free in MPA since there's only one route at one time. A straightforward rendering is critical for metrics such as first meaningful paint and time to interactive. MPAs gain it for free owing to the simplicity of the traditional HTML navigation they used.

3 https://smashed.by/preloader

3. PRECACHE REMAINING ROUTES USING SERVICE WORKERS

This is the part where service workers came to join the show. Service workers are known as a client-side proxy enabling developers to intercept requests and serve responses from the cache, but it can also initiate a `fetch` to prefetch then precache future resources.


after onLoad
prefetch food/
offline/
order/
profile/
sales/
shop/
Secure https://h5.ele.me/msite/
Nexus 5X 412 × 732 75%
Memory Elements Console Sources Network Performance Application Security Audits Cookies
Preserve log
Disable cache
Offline No throttling
Filter Regex Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Name	Met...	Stat...	Prot...	Sc...	Domain	Type	Initiator	Size	Time	Pri...
food/	GET	200	h2	htt...	h5.ele....	text/html	Other	19.0 KB	242...	Hi...
food.ca76a6a.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	19.7 KB	243...	Hi...
food.862e5d5.css	GET	200	h2	htt...	h5.ele....	text/css	Other	1.8 KB	236...	Hi...
vue-sticky.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	1017 B	236...	Hi...
offline/	GET	200	h2	htt...	h5.ele....	text/html	Other	11.4 KB	236...	Hi...
offline.748fbc8.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	1.6 KB	235...	Hi...
offline.bb15618.css	GET	200	h2	htt...	h5.ele....	text/css	Other	273 B	236...	Hi...
order/	GET	200	h2	htt...	h5.ele....	text/html	Other	14.5 KB	239...	Hi...
order.3ca1c28.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	31.2 KB	242...	Hi...
order.e4d9904.css	GET	200	h2	htt...	h5.ele....	text/css	Other	4.0 KB	252...	Hi...
profile/	GET	200	h2	htt...	h5.ele....	text/html	Other	15.1 KB	269...	Hi...
profile.5ee80f0.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	10.3 KB	258...	Hi...
profile.68a4286.css	GET	200	h2	htt...	h5.ele....	text/css	Other	1.0 KB	258...	Hi...
sales/	GET	200	h2	htt...	h5.ele....	text/html	Other	18.1 KB	289...	Hi...
sales.8bfa167.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	19.5 KB	266...	Hi...
sales.a58aba7.css	GET	200	h2	htt...	h5.ele....	text/css	Other	6.8 KB	283...	Hi...
vue-sticky.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	1.0 KB	283...	Hi...
marked.min.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	5.8 KB	279...	Hi...
shop/	GET	200	h2	htt...	h5.ele....	text/html	Other	14.4 KB	285...	Hi...
shop.99dd6d2.js	GET	200	h2	htt...	h5.ele....	javascri...	Other	6.1 KB	280...	Hi...
shop.0bbc6cd.css	GET	200	h2	htt...	h5.ele....	text/css	Other	369 B	278...	Hi...
error-load.png	GET	200	h2	htt...	github...	png	Other	5.2 KB	127 ...	Hi...
error-locate.png	GET	200	h2	htt...	github...	png	Other	4.3 KB	163 ...	Hi...
error-service.png	GET	200	h2	htt...	github...	png	Other	5.4 KB	150 ...	Hi...

66 / 200 requests | 493 KB / 940 KB transferred | Finish: 3.63 s | DOMContentLoaded: 688 ms | Load: 1.71 s

Prefetching and precaching future routes.

We already used Webpack[4] in the build process to do *.vue* compilation and asset versioning, so we created a Webpack plugin to help us collect dependencies into a precache manifest and generate a new service worker file after each build. This is pretty much like how SW-Precache works.[5]

In fact, we only collect dependencies of routes we flagged as being critical routes. You can think of them as the app shell or installation package of our app. Once they are cached/installed successfully, our web app can boot up directly from cache and be available offline. Routes that are not critical would be incrementally cached at runtime during the first visit. Thanks to the least recently used

4 https://webpack.github.io/
5 https://smashed.by/swprecache

(LRU) cache policies and time-to-live (TTL) invalidation mechanisms provided by SW-Toolbox,[6] we have no worries of hitting the quota in the long run.

4. LAZY-LOAD AND INSTANTIATE REMAINING ROUTES ON DEMAND

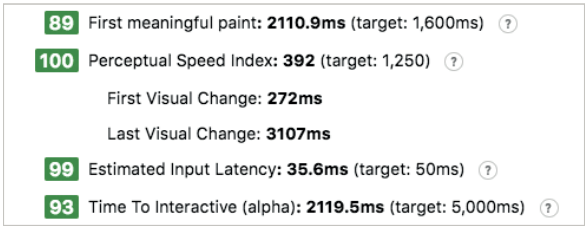
Lazy loading and lazily instantiating the remaining parts of the app are relatively challenging for an SPA to achieve. It requires both code splitting and async importing. Fortunately, this is also a built-in feature of the MPA model, in which routes are naturally separated.

Note that the lazy loading can be done instantly if the requested route is already precached in the service worker cache, no matter whether an SPA or MPA is used. #ServiceWorkerAwesomeness

The Result

Surprisingly, we found that a multi-page PWA is kinda naturally PRPL! The MPA has already provided built-in support for P, R, and L, and the second P involving service workers can be easily fulfilled in any PWA.

So what about the end result?


89 First meaningful paint: **2110.9ms** (target: 1,600ms)
100 Perceptual Speed Index: **392** (target: 1,250)
First Visual Change: **272ms**
Last Visual Change: **3107ms**
99 Estimated Input Latency: **35.6ms** (target: 50ms)
93 Time To Interactive (alpha): **2119.5ms** (target: 5,000ms)

In Lighthouse benchmarking, we made time to interactive (TTI) around 2 seconds, and this was benchmarked on our HTTP1 server.

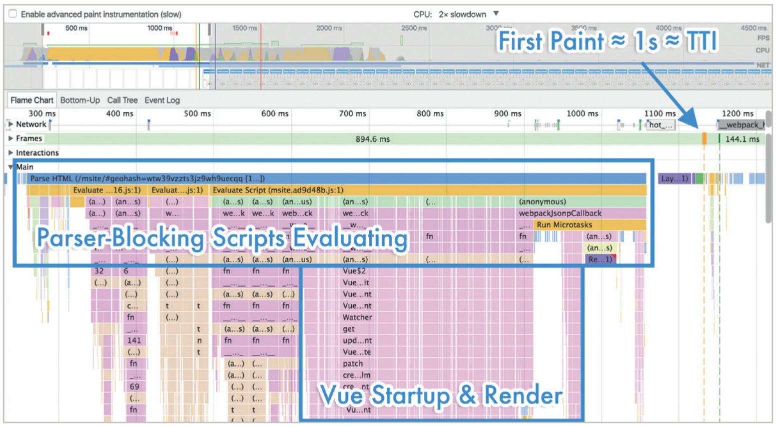
6 https://smashed.by/swtoolbox

The first visit is fast. The repeat visit with service workers is even faster. You can check out this video to see the huge difference between with or without service workers: https://smashed.by/elemesw

The video is recorded on desktop but with a highly latent server to make the gap even more apparent. Did you see that? No, I mean the annoying blank screen. Even with the service worker, the blank screen is still conspicuous during navigating. How can that be?

Multi-Page Pitfall: Redo Everything!

Unlike SPAs, changing routes in MPAs means actual browser navigation happens: the previous page is discarded completely and the browser needs to redo everything for the next route: download resources, parse HTML, evaluate JavaScript, decode image data, layout the page, and paint the screen, even if many of them could be shared across routes. All of this work combined requires significant computing power and time.



Profile of entry page (before optimization).

So here is the profile (2× slower CPU simulated) of our entry page (the heaviest one). Even if we can make TTI around 1 second in repeat visits, that might still feel too slow to our users for just switching a tab.

HUGE JAVASCRIPT RE-STARTUP COST

According to the profile, most of the time (900 ms) before hitting the first paint is spent on evaluating JavaScript. Half is on dependencies including Vue.js runtime, components, libraries, and so on. The other half is on Vue.js starting up and mounting. Because all UI rendering is dependent on JavaScript/Vue.js, all of the critical scripts remain guiltily parser-blocking. I'm by no means blaming JavaScript or Vue.js overheads here. It's just a trade-off when we need this layer of abstraction in engineering.

As an SPA, JavaScript start-up cost is amortized during the whole life cycle. Parsing/compiling for each script happens only once, and much heavy executing can be done only once. The big JavaScript objects, like Vue.js's ViewModels and virtual DOM can be kept in memory and reused as much as you want. This is not the case with MPAs, however.

COULD BROWSER CACHES HELP?

Yes and no.

Google's V8 introduced code caching, a way to store a local copy of compiled code so fetching, parsing, and compilation could all be skipped next time. As Addy Osmani mentioned in "JavaScript Start-up Performance," scripts stored in cache storage via service workers could trigger code caching in just the first execution.

Another browser cache you might hear of is back/forward cache, or bfcache. The name varies, like Opera's fast history navigation or WebKit's page cache. The idea is that browsers can keep the previous page live in memory (i.e. DOM/JS states) instead of destroying everything. In fact, this idea works very well for MPAs. You can try every traditional multi-page website in iOS Safari and observe instantaneously loading when moving back or forward. (There's a slight difference, though, with browser UI/gesture or hyperlinks.)

Unfortunately, Chrome does not currently have this kind of in-memory bfcache, concerning memory consumption and its multi-process architecture. It just leverages HTTP disk cache to simplify the loading pipeline; almost everything still needs to be redone. More details and discussions can be read in "Faster Back/Forward Navigation in Chrome"[7] by Kinuko Yasuda.

Striving for Perceived Performance

Although the reality is dark, we don't want to give up so easily. One optimization we try to do is to render DOM nodes/create virtual DOM nodes as little as possible to improve the time to interactive, while another opportunity we see is to play tricks on perceived performance.

Owen Campbell-Moore has written a great post, "Reactive Web Design: The secret to building web apps that feel amazing,"[8] covering both "instant loads with skeleton screens" and "stable loads via predefined sizes on elements" to improve perceived performance and user experience. Yes, we actually used both.

How about showing the end result after these optimizations first, before entering technical nitty gritty? Here is a video: https://smashed.by/elmepwa

So fast that you can not see the pulsing skeleton screen clearly? Here is a video showing how it looks under a 10 times slower CPU: https://smashed.by/elmecpu

7 https://smashed.by/chromenav
8 https://smashed.by/reactivedesign

This is a much better UX, right? Even if we have slow navigation in slow devices, at least the UI is stable, consistent, and always responding. So how did we get there?

PRE-RENDERING SKELETON SCREEN WITH VUE.JS AT BUILD TIME

As you might have guessed, the skeleton screen that consists of markup, styles, and images is inlined into `*.html` of each route, so they can be cached by a service worker, be loaded instantly, and be rendered independently with any JavaScript.

We don't want to manually craft each skeleton screen for each route. It's a tedious job and we have to manually sync every change between skeleton screens and the actual UI components (yes, we treat every route as just a Vue.js component). But think about it, a skeleton screen is just a blank version of a page into which information is gradually loaded.[9] What if we bake the skeleton screen into the actual UI component as just a loading state so we can render the skeleton screen out directly from it without the issue of syncing?

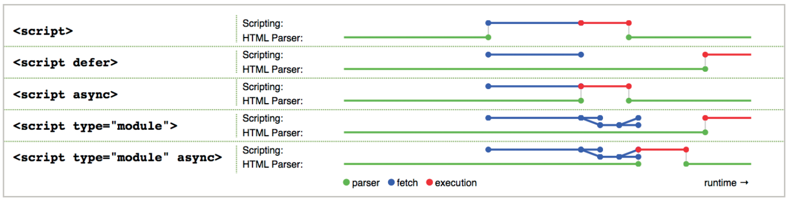
Thanks to the versatility of Vue.js, we can actually realize it with Vue.js server-side rendering.[10] Instead of using it on a real server, we use it at build time to pre-render Vue.js components to strings and inject them into HTML templates. You should write code that is "universal" to make Vue.js components that can be executed in Node. But for routes that depend heavily on some DOM/BOM-specific third-party modules, we have to make a separated `*.shell.vue` to temporarily work around it.

FAST SKELETON PAINTING

Having markups in `*.html` doesn't mean that they will be painted fast – you have to make sure the critical rendering path is optimized

9 https://smashed.by/skeletonscreen
10 https://ssr.vuejs.org/en/

for that. Many developers believed that putting script tags in the end of the body is sufficient for getting content painted before executing scripts. This might be true for browsers supporting rendering an incomplete DOM tree (e.g. streaming render), but browsers might not do that in mobile concerning slower hardwares, battery, and heat. Although we are told that script tags with async or defer are not parser-blocking, it doesn't mean we can get content painted before executing scripts in reality.



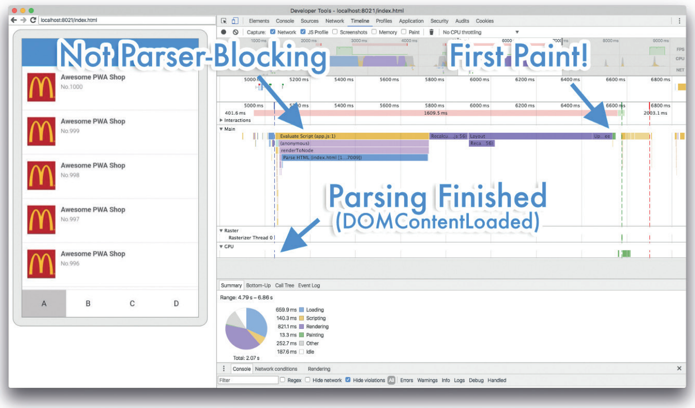
Parse, fetch, and execute for scripts with different attributes

First I want to clarify it a little bit. According to the scripting section of HTML[11] (WHATWG living standard, the W3C's same here), `async` scripts would be evaluated as soon as they are available, and thus could potentially block parsing. Only `defer` (and not inlined) is specified to never block parsing. That's why Steve Souders posted "Prefer DEFER Over ASYNC."[12] (`defer` has its own issues and we will cover that later.)

More importantly, a script not blocking parsing could still block painting nonetheless. So here is a reduced test I wrote named "minimal multi-page PWA" (MMPWA), which basically renders 1,000 list items within an `async` (and truly not parser-blocking) script to see if we can get a skeleton screen painted before scripts are executed. The profile below (over USB debugging on my real Nexus 5) shows my ignorance:

Yes, keep your mouth open. The first paint is blocked. I was surprised too. The reason, I guess, is that if we touch the DOM so quickly

11 https://smashed.by/scripting
12 https://smashed.by/defer



First paint blocked for scripts to execute.

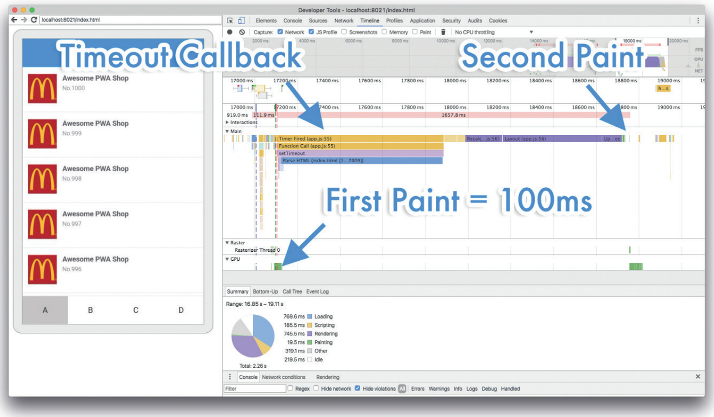
that the browser has still not finished the previous painting job, our dear browser has to abort every pixel it has drawn, and has to wait until the current DOM manipulation task finishes and redo the rendering pipeline again. And this more often happens with a mobile device with a slower CPU/GPU.

FAST SKELETON PAINTING WITH SETTIMEOUT HACK

We indeed encountered this problem when testing our new beautiful skeleton screen. Perhaps Vue finishes its job and starts to mount nodes too fast ;). But anyway we have to make it slower or, rather, lazier. So we tried putting DOM manipulation things inside `setTimeout(callback, 0)`, and it works like a charm!

You may be curious about how this change performs in the wild, so I have refined MMPWA by rendering 5,000 list items rather than 1,000 to make the differences more obvious, and by designing it in an A/B testing manner. The code is on GitHub and the demo is live on https://smashed.by/mmpwademo. Here is a link to the video: https://smashed.by/mmpwatesting

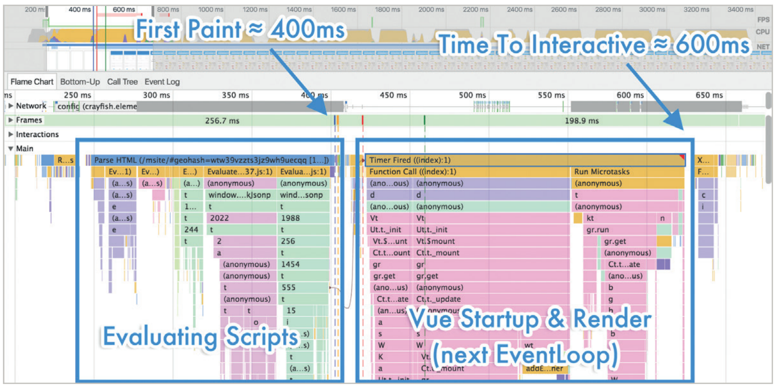
This famous `setTimeout` hack (aka zero delays) looks quite magic, but it is science™. If you are familiar with the event loop, it just



First paint unblocked for skeleton painting.

prevents this code from executing in the current loop by putting everything into the task queues with the timer callback, so the browser could breathe (update the rendering) in the main thread.

So we applied what we learned from MMPWA by putting `new Vue()` inside `setTimeout` and BOOM! We have a skeleton screen painted consistently after every navigation! Here is the profile after all these optimizations.

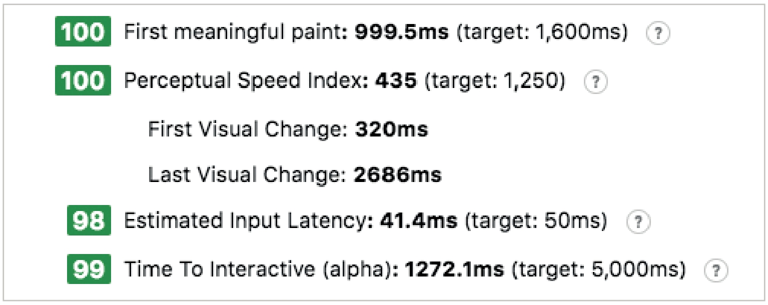


Profile of entry page (after optimization).

Huge improvements, right? This time we hit first paint (skeleton screen paint) at 400 ms and TTI at 600 ms. You should really go back to have a before–after comparison in detail.

ONE MORE THING THAT I DEFERRED

But wait, why is there still a bunch of guiltily parser-blocking scripts? Are they all async? OK, OK. For historical reasons, we do keep some parser-blocking scripts, like lib-flexible[13] – we couldn't get rid of it without huge refactoring. But most of these blocking scripts are in fact deferred. We expected that they could be executed after parsing and in order; however, the profile kinda slapped my face.


100 First meaningful paint: **999.5ms** (target: 1,600ms)
100 Perceptual Speed Index: **435** (target: 1,250)
First Visual Change: **320ms**
Last Visual Change: **2686ms**
98 Estimated Input Latency: **41.4ms** (target: 50ms)
99 Time To Interactive (alpha): **1272.1ms** (target: 5,000ms)

Possible Chrome bug.

Remember I said I would talk about one issue of `defer` previously? Yes, that's it. I have had a conversation[14] with Jake Archibald and it turns out it might be a bug of Chrome when the deferred scripts are fully cached.

Similar improvements can be seen from Lighthouse (under the same network environment but from an HTTP2 server). Pro tip: Always use Lighthouse in a variable-controlling approach.

Performance in the Real World

Alex Russell gave a very insightful talk[15] on mobile web performance at Chrome Dev Summit 2016, talking about how hard it can be to build performant web applications on mobile devices. Highly recommended.

13 https://smashed.by/libflexible
14 https://smashed.by/huxprotwitter
15 https://smashed.by/progressiveperf

Chinese users tend to have pretty powerful phones. The Xiaomi Mi4 is shipped with the Snapdragon 801 processor (slightly outperforming the Nexus 5) but only costs $100. It's affordable by at least 80% of our users so we take it as a baseline.

I made a recording[16] on my Nexus 5 showing switching between four tabs. The performance varies between tabs due to their variant scale. The heaviest one, the entry page, takes around 1 s to hit real time to interactive. This is surprisingly comparable to what I get from Chrome Simulation with 2× CPU throttling. With 5× throttling, this can take 2–3 seconds to get TTI, horribly. (To be honest, I found that even under the same throttling, the results can vary drastically depending on my Macbook's "mood".)

Final Thoughts

This case study is much longer than I expected. I really appreciate you getting here. So what can we learn from it?

MPA STILL HAS SOME WAY TO GO

Jake Archibald even said that "PWA !== SPA" at Chrome Dev Summit 2016. But the sad truth is that even though we have taken advantage of bleeding-edge technologies such as the PRPL pattern, service workers, app shell, and skeleton screens, there is still a distance between us and many single page PWAs just because we are multi-page structured.

The web is extremely versatile. Static blogs, e-commerce websites, desktop-level software, all of those different scaling things should all be first-class citizens of the web family. MPAs might have things like bfcache API and navigation transitions to catch up to SPAs in the future, but it is not today certainly.

16 https://smashed.by/elemerecording

PWA IS AWESOME NO MATTER WHAT

Hey, I am not overblowing it. Even as a multi-page PWA, Ele.me couldn't be as stunning and app-like as many single page PWAs are. The idea and technologies behind PWAs still help us deliver a much better experience to our users on the web that hasn't been possible before.

What our PWA is trying to solve are some fundamental problems of the current web application model, such as its hard dependencies to network and browser UIs. That's why a PWA can always be beneficial no matter what architecture or framework you actually use.

Ele.me Key Takeaways

A multi-page PWA helped to deliver a much better experience to web users.

PWA's usually tend to be single-page apps. However, when Ele.me, China's biggest food order and delivery app, decided to convert their mobile website to a PWA, they wanted to use a multi-page structure. This would allow them to build different parts of the mobile website as microservices.

The team chose the Vue.js platform for implementation as it is both lightweight and progressive. Additionally, the team leveraged the PRPL pattern to preload critical resources, render initial routes, pre-cache with service workers, and lazy-load remaining routes. They then used skeleton screens to improve the perceived performance of the app. A combination of multiple technologies resulted in a performant PWA for a multi-page app which otherwise seemed difficult. The PWA helped to deliver a better experience to their web app users.

Interview

Xuan Huang

Former Front-end Engineer at Ele.me

Author of **"Upgrading Ele.me to be a PWA"**

What excited you or your team the most about the work in the case study?

1. "Performance performance" and new technologies.

   Performance optimization is always attractive to any large-scale website. But due to their scale, it usually only happens in small and independent steps and evolves rather slowly.

   Adopting technologies and patterns under the PWA umbrella, however, gave us a great opportunity to do or revisit a lot of such optimizations that we've heard about and wanted or planned to do, all together at once and also with a more systematic approach. In particular, we implement the PRPL pattern and new pre-caching capability provided by service workers.

2. Perceived performance and UX

   We always want to make our website more "appy," which you could easily tell from how much our UI design is influenced by native apps (bottom navigation, for instance).

   PWAs as a broader concept provide not only technologies but also advice on designs that are particularly applicable to the web. "Skeleton screen" and "stable load" are two of the examples that we implemented, finding that they greatly improve the user experience.

> Although we were working at a time when not many users knew how to add to home screen, we're so happy and proud that it became the little thing that every one of us always does to show off to other devs.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

If we look at the data (from the developer.google.com case study), the number might not be as dramatic as many would imagine. But if we consider the scale of our user base and the browser support distribution – it was very satisfying.

There is another impact I'm very surprised about. Our work of shipping this world-first PWA for the Chinese market made a significant impact on boosting the awareness of PWA technologies on the Chinese developer community, as well as browser vendors (like Tencent and UC), and we believed that helped push the boundaries of web experience and browser support in China.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

Technology- and support-wise, PWAs have grown more mature and users are more aware of them as well.

But speaking for the Chinese market, this question might not be answered as you expect. The use of web technologies in general has gone down significantly such that most Chinese products have been transitioned to the "mini app" space.

(N.B. This is more of an objective observation of the marketplace rather than my subjective preference.)

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

I'd say it's investigating PWA technologies while maintaining the existing multipage app architectures, despite the fact that a single-page app was the prevailing way of doing PWA in the community.

First, we might not have been able to complete the PWA if we had to restructure the entire website. It would be too much of an obstacle to overcome.

Second, we were able to maintain the strengths of the MPA already appreciated by the company and team (with the reasons detailed in the two case study writings).

Finally, we came to be one of the earliest pioneers exploring the domain of integrating the PRPL pattern into MPA, which ended up being valuable to the wider community. The most applauded response under the Medium post is "We have an MPA architecture too. Really inspiring to see what you've achieved."

I'm personally really happy that our exploration can help more people on board with PWA despite how their websites are structured.

Oh, did I forget to mention the bfcache? Chrome has implemented it!

What came next after the case study was published?

In parallel with our efforts launching the Ele.me PWA, another rival technology was rising in China at the same time (2017): the so-called "mini-app"

- Wechat Mini Program[1]
- Alipay Mini Program[2]

This is essentially an enhanced and customized semi-web environment provided by some of the biggest monopoly apps that allows small businesses ("long tail") to integrate and develop their service on.

While I personally prefer and advocate the spirit of the open web, these kinds of walled gardens have been rapidly growing since 2017, and have become extremely successful, dominating the long-tail market from 2020. Many web front-end engineers are hired to exclusively develop such "mini programs."

Ele.me is not an exception participating in this trend. After roughly a year or so, the company realized the majority of its web users have been transitioned to the Ele.me mini app ("installed" in either Wechat or Alipay) and reduced to less than 1%.

Hence, unfortunately, any new development for the website is paused and reduced to minimal complexity, only maintaining status.

Do you have any advice for teams that would like to follow in your footsteps?

Yeah. I'd say "progressive enhancement, graceful degradation" is still the king of web development philosophies.

1 https://smashed.by/wechatmini
2 https://smashed.by/alipaymini

Although we faced some challenges on upgrading an MPA to a PWA, it would have not been done if we had to do everything from scratch.

Most PWA technologies can be gradually adopted as a progressive enhancement, and they are designed with degradability in mind. Therefore, do not hesitate to try them! There are many low-hanging fruits like adding a web manifest, as well as new capabilities provided by new APIs such as service workers.

The web is always evolving as a whole without losing its heritage. We all know there are bad things about it, but the good sides are such that even the oldest web technologies can collaborate and benefit from the newest web technologies.

Together, let us bring this next-generation web application model and experience to our users to demonstrate its new power.

Has the site changed significantly since the case study was published?

Yes and no.

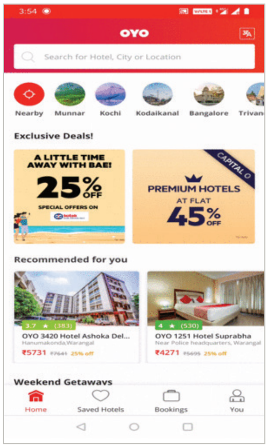
The improvements on perceived performance and UX have been preserved. You can still find some of the skeleton screens, and the UI is really stable. The web manifest has been kept as well.

However, the service worker has been dropped owing to its added complexity, which conflicts with the goal to slim down the site so it's easier to maintain.

Trusted Web Activities (or TWA) — A Complete Implementation Guide to OYO Lite

by Ankit Jain

We all know that users like to keep only those apps they use on a regular basis.[1] The primary reason for uninstalls is the size of the app. With the help of trusted web activities (TWA), users will enjoy the native app experience without having to compromise on the storage factor. OYO Lite[2] gave us three times more conversion than our mobile web (progressive web app) similar to that of the native OYO app, and three times higher logged in user percentage.



OYO Lite App.

In this case study we'll talk about how someone can use their existing web app to build an Android app with the help of TWA. Let's first see what you are signing up for:[3]

1 The original version of this case study was published in November 2019: https://smashed.by/oyotech
2 https://smashed.by/oyolite
3 https://smashed.by/oyoliteapp

What the Heck Is TWA?

Trusted web activities (TWA) are a new way to integrate your web app content with your Android app using a protocol based on Custom Tabs.[4] Although Android apps routinely include web content using a Chrome Custom Tab (with URL bar) or WebView, TWA runs your app fullscreen in the default browser (after the recent changes, TWA will open Chrome even if it is not the default) and hence can leverage the features and performance optimizations of the browser.

Note: TWA shares the browser data, like cookies and localStorage, inside the app, so if you are logged in inside a browser then you'll automatically be logged in the TWA app as well.

Why You Should Care about TWA

Generally speaking, the native app has better conversion, a more loyal user base than the web app, but it has some drawbacks also as mentioned below. TWA tries to reduce the gap between web and native experience, and it can solve problems such as updating content on the fly, while solving the storage problem for your end users.

OYO Lite[5] (our TWA app) is ~850 KB (7% compared to our main app), so it doesn't have storage issues and can be used to target low-end devices.

CAPABILITIES AVAILABLE IN TWA SIMILAR TO A NATIVE APP

- **Available at the tap of a button**: Since a TWA app will be treated similar to any other app in the Android system, it'll also have a launcher icon.
- **Works offline**: With the help of service worker caching, the app will work in offline mode also.

4 https://smashed.by/customtabs
5 https://smashed.by/oyolite

- **Fast loading**: Native apps come with all the assets bundled in them; the web app can cache its assets (JS, CSS, etc.) in the browser or service worker cache. In general, a web app's time to interactive (TTI) should be around 5 seconds for a great loading experience.

- **Keep users engaged**: Apps use push notifications for re-engagement; the same can be achieved by web apps with service workers. Native push notifications support is also planned for TWA.

- **Deep linking**: Any of your domain links can be opened in your TWA app by digital asset links[6] pairing and an intent filter in your manifest file.

- **Run fullscreen**: TWA apps can also run fullscreen with the help of digital assets link verification. The app and the site it opens are expected to come from the same developer.

CAPABILITIES IN TWA BETTER THAN NATIVE APPS

- **Can update on the fly**: If buggy code is shipped in a native app, you can't do anything but wait for your users to update the app. This is not the case with TWA apps since they are just the web app wrapped inside an app, so the code can be updated anytime just like the web.

- **Backward compatibility is not a problem**: There are no version checks in APIs built for TWA apps as they all will be running the same code.

- **Size**: Since native apps ship with all the machinery needed to run, their size usually reaches a few megabytes. TWA apps internally run a browser and request for a webpage, no code is shipped with the Android package kit (APK), and hence the whole app size gets reduced to a few hundred KB.

In conclusion, TWA is giving the best of both worlds, isn't it? Now let's try to see what it takes to build one.

6 https://smashed.by/assetlinks

Criteria for a Web App to Be Turned into a TWA App

There are currently no qualifications for content opened in the preview of trusted web activities. It means any web app can be used to build a TWA app, but users won't like your app if it shows a URL bar inside an app, displays a "not connected to internet" message when offline, takes too much time to load, or transitions between pages are not smooth like a native app. So to make a decent TWA app, bare minimum qualifications should be:

- To be accessible and operable even when offline.
- To have digital asset links set up.
- To work as a reliable, fast, and engaging standalone component within the launching app's flow.

If the web app is already a progressive web app (PWA) with a good score on Lighthouse, then you just have to set up digital asset links.

A Very Basic TWA

To build a basic TWA app, we followed the steps mentioned in the official Google documentation.[7] Our project underwent the following changes:

1. Created an Android manifest file containing the `DEFAULT_URL` (i.e https://www.oyorooms.com), and intent filters to define that this activity is the launcher, and an intent filter that says that this app can handle oyorooms URLS.

2. The URL bar was removed with the help of digital asset links verification.

7 https://smashed.by/usingtwa

3. Launch icon was created.

There is a long white screen between launching the app and getting anything on the screen. This is the time when the browser is getting initialized and your web app is getting the HTML document. We can't avoid it completely but we can serve something which the user is familiar with: a splash screen.

Adding the Splash Screen: The Right Way

A splash screen is the screen that generally every native app shows till it loads so that users understand that the app is starting up.

Starting on Chrome 75, trusted web activities have support for splash screens and we just have to provide background color and an image. This is sufficient, but it won't suffice for all users as:

- Some users will be using browsers other than Chrome.
- All users won't have Chrome 75 or newer.

So we went with the splash screen provided by TWA and also wrote our own custom splash activity, which handled the use cases discussed above.

First, we created another activity and made it the launcher, which means this activity will be started on clicking the app icon.

AndroidManifest.xml:

```
<activity
  android:name=".SplashActivity"
  android:theme="@style/AppTheme.NoActionBar">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>


  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT"
/>
    <category android:name="android.intent.category.BROWSABLE"
/>
    <data
      android:host="@string/website_host"
      android:scheme="@string/website_scheme"
      />
  </intent-filter>

</activity>
<activity    android:name="android.support.customtabs.trusted.
LauncherActivity"
android:theme="@style/AppTheme.NoActionBar">
.
.
<meta-data
android:name="android.support.customtabs.trusted.STATUS_BAR_
COLOR"
android:resource="@color/colorPrimary" />
<meta-data
  android:name="android.support.customtabs.trusted.SPLASH_
IMAGE_DRAWABLE"
android:resource="@drawable/ic_oyo_lite_white" />
<meta-data android:name="android.support.customtabs.trusted.
SPLASH_SCREEN_BACKGROUND_COLOR"
android:resource="@color/colorPrimary"/>
<meta-data
  android:name="android.support.customtabs.trusted.SPLASH_
SCREEN_FADE_OUT_DURATION"
android:value="500" />
<meta-data
  android:name="android.support.customtabs.trusted.FILE_
PROVIDER_AUTHORITY"
android:value="oyo.consumerlite.authority" />
</activity>
```

Now the newly created `SplashActivity` should do the following tasks:

1. Check if the splash screen is supported or not. This can be achieved by comparing the installed Chrome version with the Chrome 75 version.

2. If the splash screen is supported, just launch the trusted launcher activity with the URL. TWA will handle the splash screen. Metadata about the splash screen is provided to TWA in the Android manifest.

3. If splash screen is not supported, then show the custom splash screen layout for some time (somewhere around 400 ms seems decent) and then launch the trusted launcher activity.

In this way, users having old or newer Chrome versions will get the splash screen. Although the handling of splash screens by TWA is much better than custom handling as in the latter, we are putting a delay in launching the activity, whereas TWA shows the splash screen till the page is rendered behind the scenes and then fades it out, which gives a nice experience.

Let's Talk Numbers

- Conversion: 3× of PWA
- Play Store rating: 4.1
- Logged-in %: 3× of PWA
- Realization: 1.5× of PWA

Apart from the above stats, TWA helps in product building/marketing:

- Multiple presence on play store, leading to a higher opportunity for user acquisition.

- Higher retention rates and stickiness.
- Useful in markets with low internet penetration, especially helpful for OYO, which has a presence across 80 countries.
- Helpful in personalized marketing (to push the most relevant product for the right audience).
- Provides a better platform for customer relationship management activities as compared to normal mobile web.
- No incremental releases required around this – easier incremental changes.

OYO Lite Key Takeaways

Three times more conversions than PWA and three times higher logged in user percentage achieved by converting to TWA.

Users tend to delete the apps they don't use regularly. Companies like OYO, a hotel booking service, found that OYOLite, their web app which combines native app features, is more beneficial from a user engagement perspective. They used trusted web activities (TWA) to create the OYOLite app.

TWA provides a new way of integrating web app content to your Android app using a protocol based on Chrome Custom Tabs. TWA tries to bridge the gap between the web and native experience as they can be accessed like a native app and open full screen but are fast and small in size like a web app. They also allow native app features like splash screens, deep links, and android shortcuts. In the case of OYO, their TWA helped them to achieve better user conversions and logged-in user percentage and also reach users in areas with low internet access.

Interview

Ankit Jain

Former SDE-3 at OYO

Author of **"A Complete guide to Trusted Web Activities with OYO"**

What excited you or your team the most about the work in the case study?

They say web is available for everybody, but native apps have a dedicated user base who like to access our product in a single tap. But they also uninstall it after their usage to save memory. We wanted to solve these problems, and TWA gave us the best of both worlds. We are continuing working on it to target the correct user base.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

Yes, it gave us a three-times better conversion than our mobile web (progressive web app) and one-and-a-half times better realization.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

We can still think of putting a nudge on the main app page in Play Store that says that the Lite version is available. It will help us reduce cannibalization and the choice will be for users completely.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

When we built it, there was very little support for splash screen, webview as a fallback, etc. If we had chosen to ignore these features in OYO Lite, then users wouldn't have felt the app-like experience and we didn't want that. So we handled all these necessary features and shipped them in OYO Lite.

What came next after the case study was published?

We want to grow OYO Lite more, and for that we are working on targeting the correct user base and flows.

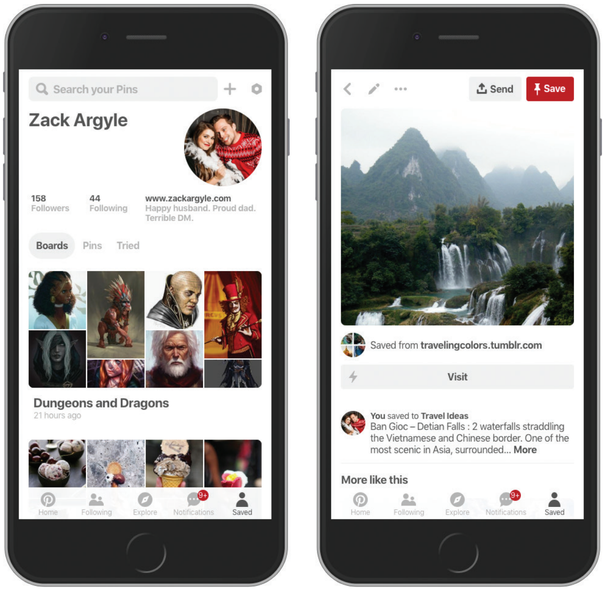
Do you have any advice for teams that would like to follow in your footsteps?

Lite apps built with TWA are way smaller than their native alternatives (OYO Lite is only 7% of its native app). So if our PWA is performant, then we should really try to explore the TWA world.

A Year into the Pinterest PWA

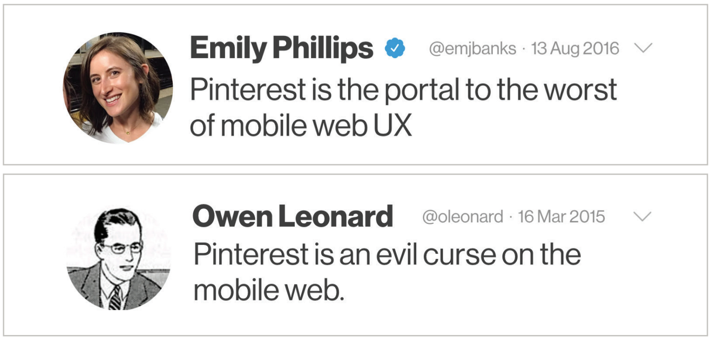
by Zack Argyle

The idea of building a progressive web app (PWA) is not new,[1] but its definition has changed with the emergence of key technologies like service workers. Now it's finally possible to build great experiences in a mobile browser. Being an early adopter can be scary, so we'd like to share a brief overview of our experience building one of the world's largest progressive web apps.


Search your Pins
Zack Argyle
158 Followers
44 Following
www.zackargyle.com
Happy husband. Proud dad. Terrible DM.
Boards
Pins
Tried
Dungeons and Dragons
21 hours ago
Home
Following
Explore
Notifications
Saved
Send
Save
Saved from travelingcolors.tumblr.com
Visit
You saved to Travel Ideas
Ban Gioc – Detian Falls : 2 waterfalls straddling the Vietnamese and Chinese border. One of the most scenic in Asia, surrounded... More
More like this

Three years ago we looked at the state of our website on mobile browsers and groaned at the obvious deficiencies. Metrics pointed to an 80% higher engagement rate in our native apps, so the decision was made to go all-in on our apps for iOS and Android. Despite increasing our app downloads substantially, there were some obvious downsides.

1 The original version of this case study was published in July 2018: https://smashed.by/pinterestpwa


Emily Phillips @emjbanks · 13 Aug 2016
Pinterest is the portal to the worst of mobile web UX

Owen Leonard @oleonard · 16 Mar 2015
Pinterest is an evil curse on the mobile web.

Emily. Owen. We would like to take this moment to offer an apology. You were right. It was terrible.

In July 2017 we brought a team together to rewrite our mobile website from scratch as a PWA. This was the culmination of several years of conversation, months of metrics investigation, and one large hypothesis: mobile web can be as good as a native app. The results are quite… pinteresting.

Why Did We Do It?

There were two main reasons why we reinvested so heavily in our mobile web. The first was our users. Our mobile web experience for people in low-bandwidth environments and limited data plans was not good. With more than half of all Pinners based outside the United States, building a first-class mobile website was an opportunity to make Pinterest more accessible globally and, ultimately, improve the experience for everyone.

The second reason was data-driven. Because the experience wasn't great, a very small percentage of the unauthenticated users that landed on our mobile web site either installed the app, signed up or logged in. It was not a good funnel. Even if we weigh the native app users more heavily for higher engagement than mobile web users, it's not the type of conversion rate anyone strives for. We thought we could do better.

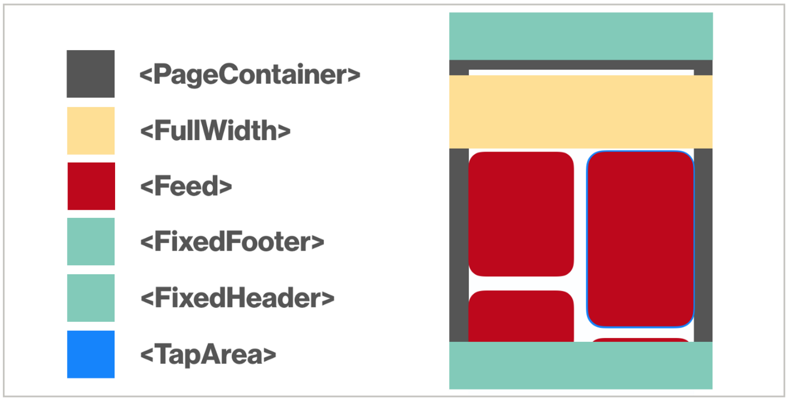
How Did We Do It?

In July 2017 we formed a team that combined engineers from our web platform and growth teams. Internally, we called it "Project Duplo," inspired by simplicity and accessibility. At the time, the mobile website accounted for less than 10% of our total sign-ups (for context, the desktop website drove five times that).

TIMELINE

- July 2017: Begin "Project Duplo"
- August 2017: Launch new mobile site for percentage of logged-in users
- September 2017: Ship new mobile site for logged-in users
- January 2018: Launch new mobile site for percentage of logged-out users
- February 2018: Ship new mobile site for logged-out users

Part of the reason we were able to create and ship a full-featured rewrite in three months was thanks to our open-source UI library,



In addition to Gestalt, we also used React, React Router 4, Redux, Redux Thunk, React Redux, Normalizr, Reselect, Flow, and Prettier.

Gestalt. At Pinterest, we use React 16 for all web development. Gestalt's suite of components are built to encompass our design language, which makes it very easy to create consistently beautiful pages without worrying about CSS. We created a suite of mobile web-specific layout components for creating consistently spaced pages throughout the site. `FullWidth` breaks out of the default boundaries of `PageContainer`, which breaks out of the boundaries of a `FixedHeader`. This kind of compositional layout led to fast, bug-free UI development.

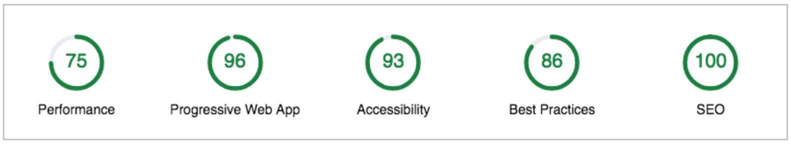
How We Made It Fast!

Performance was baked into the goals and process because of how tightly correlated it is to engagement, and how sensitive it is on a mobile connection. In fact, our home page JavaScript payload went from about 490 KB to around 190 KB. This was achieved through code-splitting at the route level by default, encouraging use of a `<Loader>` component for component-level code-splitting. An easy-to-use route preloading system was built into our client-side router, which creates a fast experience for initial page load as well as client-side route changes. For more details on how we made it fast, check out the performance case study we did with Addy Osmani.[2]

After one year, there are about 600 JavaScript files in our mobile web codebase, and all it takes is one ill-chosen import to bloat your bundle. It's really hard to maintain performance! We share code extensively across subsites for **.pinterest.com*, and so we have certain measures set up to ensure that mobile web's dependencies stay clean. First is a set of graphs reporting build sizes with alerts for when bundles exceed permitted growth rates. Second is a custom ESLint rule that disallows importing from files and directories we

2 https://smashed.by/pwaretrospective

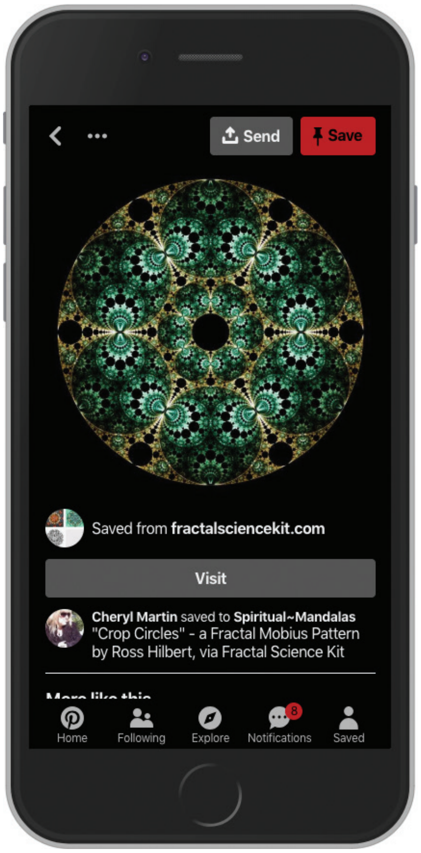
know are dependency-heavy and will bloat the bundle. For example, mobile web cannot import from the desktop web codebase, but we have a directory of "safe" packages that can be shared across both. There's still work to do, but we're proud of where we are:



While the case study deals mostly with page load, we also cared deeply about a fast, native-like experience while browsing. The biggest driver of client-side performance was our normalized Redux store which allows for near-instant route changes. By having a single source of truth for models, like a Pin or user, it makes it trivial to show the information you have while waiting for more to load. For example, if you browse a feed of Pins, we have information about each Pin. When you tap on one, it takes you to a detailed view. Because the Pin data is normalized, we can easily show the limited details we have from the feed view until the full details finish being fetched from the server. When you click on a user avatar, we show that user's profile with the information we have while we fetch the full user details. If you're interested in the structure of our state or the flow of our actions, the Redux devtools extension is enabled in production for our mobile web site.

At the heart of the new site was our attempt at building a truly progressive web app. We support an app shell, add to homescreen, push notifications, and asset caching. The service worker caches a server-rendered, user-specific app shell that's used for subsequent page loads and creates near-instant page refreshes. We're excited that Apple is building support for service workers in Safari so that all users can have the best "native-like" experience.

And what kind of "native" experience would it be without a "night mode"?



Saved from **fractalsciencekit.com**

Cheryl Martin saved to **Spiritual~Mandalas**
"Crop Circles" - a Fractal Mobius Pattern by Ross Hilbert, via Fractal Science Kit

The Verdict

Now for the part you've all been waiting for: the numbers. Weekly active users on mobile web have increased 103% year-over-year overall, with a 156% increase in Brazil and 312% increase in India. On the engagement side, session length increased by 296%, the number of Pins seen increased by 401%, and people were 295% more likely to save a Pin to a board.

Those are amazing in and of themselves, but the growth front is where things really shone. Logins increased by 370% and new sign-ups increased by 843% year-over-year. Since we shipped the new experience, mobile web has become the top platform for new signups. And for fun, in less than 6 months since fully shipping, we already have 800,000 weekly users using our PWA like a native app (from their home screen).

Looking back over one full year since we started rebuilding our mobile web, we're so proud of the experience we've created for our users. Not only is it significantly faster, it's also our first platform to support right-to-left languages and "night mode." Investing in a full-featured PWA has exceeded our expectations. And we're just getting started.

Pinterest Key Takeaways

An overhaul of the web app to improve performance and make it progressive leads to an 843% increase in new sign-ups.

The Pinterest web app in 2017 was very different from what it is now. Users found the experience very poor, and as a result visiting users did not end up signing up or downloading the app. The app was especially slow for around 50% of its users who accessed it from low bandwidth regions. Pinterest decided to fix the issues and revamp the app in mid-2017, and the new web app launched early in 2018.

Besides addressing the performance issues, the app was converted to a PWA with capabilities like an app shell, add to the home screen, push notifications, and asset caching. The PWA improved the user experience as the number of active users year-on-year increased by 103%, with a 312% increase in India. The biggest gain seen was for new sign-ups, which increased by 843% year-on-year.

Building Spotify's New Web Player

By José M. Pérez

The purpose of this case study is to tell the story of the new Spotify web player: how and why it came to be.[1] We will focus on what the steps were that led to a complete rewrite, and how the lessons learned influenced the experience and the tech decisions of the new web player for desktop browsers.[2]

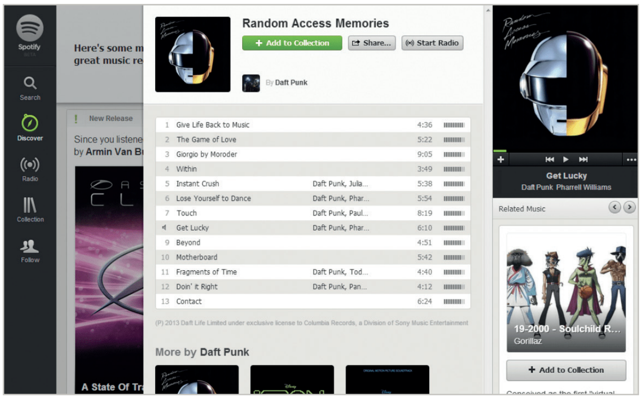
Using the Web to Implement Spotify Applications at Spotify

Spotify has been using web technologies for a long time. Before tools like Electron[3] became a reality for building hybrid applications, Spotify started using Chromium Embedded Framework (CEF)[4] in 2011 to embed web views on the desktop application. This made it easier to build and iterate on different parts of the application without having to perform full releases. It was also the foundation used to integrate a myriad of third-party apps built using web technologies, what we called Spotify Apps.

Spotify's web player was released in 2012 and complemented the experience on desktop devices. It made it possible for users to play music from Spotify as quickly as possible, without needing to download and install any application.

The architecture of the web player followed the same approach as the desktop application. The views were isolated from each other

1 The original version of this case study was published in March 2019: https://smashed.by/spotifyengineering
2 https://open.spotify.com/
3 https://electronjs.org/
4 https://smashed.by/cef



1	Give Life Back to Music		4:36
2	The Game of Love		5:22
3	Giorgio by Moroder		9:05
4	Within		3:49
5	Instant Crush	Daft Punk, Julia...	5:38
6	Lose Yourself to Dance	Daft Punk, Phar...	5:54
7	Touch	Daft Punk, Paul...	8:19
	Get Lucky	Daft Punk, Phar...	6:10
9	Beyond		4:51
10	Motherboard		5:42
11	Fragments of Time	Daft Punk, Tod...	4:40
12	Doin' it Right	Daft Punk, Pan...	4:12
13	Contact		6:24

(P) 2013 Daft Life Limited under exclusive license to Columbia Records, a Division of Sony Music Entertainment

An early version of Spotify's web player.

using iframes, and this allowed the teams to iterate on and release them without interfering with the rest of the application.

In addition, the code for the views was identical on both desktop and web player. Thus, the team working on the playlist view would implement a new feature and make it available on the desktop application and the web player without having to care about the underlying infrastructure.

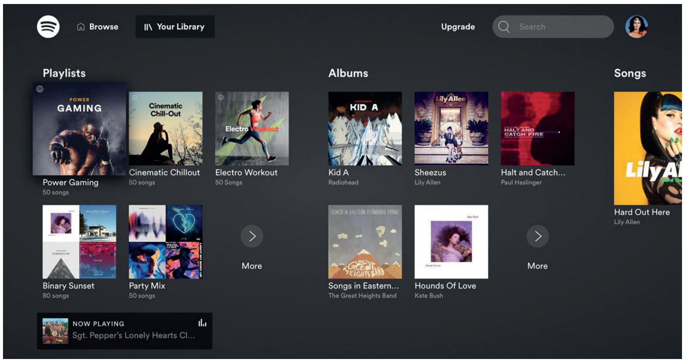
The architecture of the web player was ideal for consistency between platforms, and fit how the company was organized in feature teams. It also had its drawbacks.

Having iframes for every feature and having that feature load its own JavaScript and CSS might have worked well for the desktop application, which the user downloads bundled with all the resources that it needs. The web player, on the other hand, had to download many resources every time the user navigated between views, which resulted in long load times, which impacted user experience.

Considering a New Web Player

Over the years, we got better at prioritizing a core set of features. With the rise of smartphones, we learned how to strive for removing clutter, to properly A/B test features, and to better understand what was really needed to deliver a good user experience.

In the summer of 2016 we decided to improve the web player. We realized that the architecture of isolated views was difficult to maintain and was preventing us from building a better product. We wanted to go back to basics and support a set of core features (e.g. playback, library management, and search) and work our way from there.



Spotify for TV.

We found inspiration in the Spotify application for TV and video consoles.[5] This application is a web-based single page application, and uses the Spotify Web API,[6] which combines the access to lots of micro services to create a unified interface to manipulate Spotify data. It represented a good example of a light client being built by a single team leveraging existing libraries at Spotify. We researched the feasibility of upgrading the web player, rewriting it view by view. In parallel, we started working on a prototype following a similar architecture to the TV application. After considering the two approaches, we decided on the latter.

5 https://smashed.by/spotifytv
6 https://smashed.by/spotifyapi

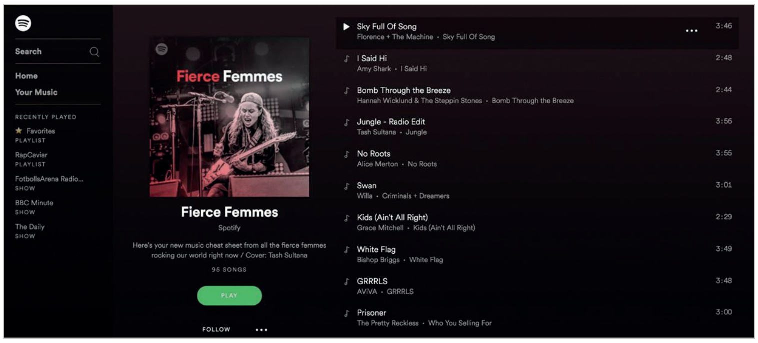
As a company we usually try to improve existing systems iteratively instead of completely replacing systems with new ones. There were a few key points behind the decision to rewrite the new web player from scratch versus improving the existing one:

- The system to deliver the code for the views, which worked in isolation from one another, wasn't used by the desktop application anymore, and it was too complex for the web player use case.
- The web player was based on lots of libraries and frameworks that were quite outdated. Giving every team an isolated environment to run their code also resulted in them choosing different client-side stacks to build their views.
- The web player was built by multiple teams with over 40 developers but now would be maintained by a dedicated team of five developers.
- It was very slow to iterate on and experiment, especially when it came to making changes across multiple views, like updating the visual style.

The Birth of a New Web Player

We decided not to repeat the mistakes of the past, so before deciding the feature set that the new web player should have, we ran A/B tests on the existing web player. For some users we removed certain features and we measured their impact in user engagement. After getting the results, we decided on the bare minimum feature set that we would feel comfortable with releasing and that our users would enjoy.

We built a minimum viable product (MVP) in a few weeks, using our new infrastructure based on Spotify's Web API. During the following months, we carried out extensive user testing and improved the pro-



Sky Full Of Song	Florence + The Machine • Sky Full Of Song	3:46
I Said Hi	Amy Shark • I Said Hi	2:48
Bomb Through the Breeze	Hannah Wicklund & The Steppin Stones • Bomb Through the Breeze	2:44
Jungle - Radio Edit	Tash Sultana • Jungle	3:56
No Roots	Alice Merton • No Roots	3:55
Swan	Willa • Criminals + Dreamers	3:01
Kids (Ain't All Right)	Grace Mitchell • Kids (Ain't All Right)	2:29
White Flag	Bishop Briggs • White Flag	3:49
GRRRLS	AViVA • GRRRLS	3:48
Prisoner	The Pretty Reckless • Who You Selling For	3:00

The new web player.

totype based on the feedback. Once we felt comfortable, we released it to a small percentage of users side-by-side with the existing web player, and checked the performance among them closely.

Our hypothesis was proved. The simpler and faster web player outperformed the old web player in all key metrics.

The Tech Architecture

The new web player is in line with the overall Spotify look and feel, and is built on HTML5 standards. It drops Flash in favor of encrypted media extensions (EME)[7] for music playback, which is supported natively by most modern browsers. It is fast, even on spotty connections, and responsive, and we have focused on making it enjoyable to use.

The architecture is based on React + Redux, which has made it easier for us to share components between the views, to have a clear data flow, and to improve debuggability and testability. Although the components are not shared with other Spotify clients, we see a trend in other Spotify web development teams who are also embracing a similar approach to building web experiences.

Making the decision to embrace well-known open source solutions and avoiding using Spotify custom libraries allowed us to onboard

7 https://smashed.by/eme

new developers quickly. This has led to numerous contributions from web developers from all over the company.

Having a simpler architecture allowed us to experiment faster and add features that didn't exist in the old web player, like daily mixes, video and audio podcasts, and Connect.[8] On top of that, we were also able to build fast CI/CD pipelines. Now with every commit the latest version of the web player is reaching our users immediately. Finally, we have a web player leveraging today's technologies. As an example, we added support for progressive web apps on Chrome OS,[9] so the web player is installed and run as a regular desktop application.

Spotify Key Takeaways

A simpler and faster web player for desktop users using modern technology and based on user preferences outperforms the old web player.

Spotify had released a web app for desktop users to complement its desktop app as early as 2012. However, this web player reused most of the code and features of the desktop app by loading similar content to iframes on the web app. Over the years, Spotify realized that the architecture had become challenging to maintain and decided to build a simpler app based on their single-page apps for TV and video consoles using Spotify Web API.

Spotify built the new web player by considering user preferences after performing A/B testing. It was built on the HTML5 standard and uses Encrypted Media Extensions for music playback instead of Flash. The new design and architecture make it fast, responsive, and enjoyable to users and allow developers to release new features quickly. The web player can also run as a PWA on Chrome OS.

8 https://smashed.by/connect
9 https://smashed.by/pwachrome

Interview

José M. Pérez

Former Engineering Manager at Spotify

Author of "**Building Spotify's New Web Player**"

What excited you or your team the most about the work in the case study?

The new Spotify web player was born out of necessity. The previous version mimicked Spotify's organization, where many feature teams could deploy mini sites run within iframes. With the change of focus towards mobile and a native desktop application, the web player had become slow and challenging to maintain. We wanted to build a product that was cohesive and delightful, and that could work well on any device and network condition.

We decided to build a single page application (SPA) with a shared data store. Navigating between pages was instantaneous. We would render a skeleton page with the header in its final state, and render the rest of the page through additional data fetching.

We also included lazy loading for images through IntersectionObserver, which reduced the data consumption without penalizing the user experience.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

After the release of the new web player we soon started seeing an increase in traffic from countries with slower network connections. Usage from devices like Chromebooks rocketed, as the web player didn't require installing any application and it offered a similar experience.

Spotify had traditionally considered the web player as a gateway to drive desktop app installs, since users who had downloaded the app were more engaged. This proved to be wrong, and we saw a considerable increase in users and retention soon after releasing the new player.

It's important to be present where the user is and give them choices. With features like push notifications, service workers, picture-in-picture, or File API, the web doesn't have to envy native applications.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same?
Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

I think today I would have taken a similar approach, but be even more metric-driven. I've grown to think that metrics are important and "data wins arguments." Metrics remove part of the bias, and are especially important when proposing rebuilding a product. Lots of stakeholders will think these decisions are made by developers because they want to have fun and play with new technology. The data that proved that building from scratch was the best way forward prevented many discussions and gave a clear path towards execution.

I learned that you need to make sure to spend the right amount of time analyzing the problem and wondering – from the very beginning – how you are going to prove that the project is successful. This lets you monitor the right metrics and be more analytical and less sentimental.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

I liked that we had constraints and we wanted to focus on building a product in a few months' time. Having a timeline with planned milestones and deliverables makes everyone involved focus on the outcome, avoiding bikeshedding on technical details that are insignificant.

What came next after the case study was published?

The technical approach we followed, using lazy loading and shared data stores to improve the site speed, were showcased at Google IO 2018 and 2019. This was accompanied by the post published on Spotify's engineering blog, explaining the history of the project and why we had made some decisions.

The feedback we got was really positive and gave the team more confidence to share our thinking process with the world. It also paved the way for other projects that adopted a similar tech stack and ideas around data fetching, loading of assets, and navigation.

> We think we can create the definitive product with a stack that will remain forever. However, we make decisions based on the current state of the technology and business insights. Those will change over time, and a good architecture makes it possible to introduce changes over time, replacing parts of it with better alternatives and removing sections that are not needed anymore.

We learned that many companies, small or large, have similar challenges. Being open about what worked and what didn't can be seen as a weakness, but it's quite the opposite. You help other teams going through a similar journey, and they share ideas back with you. The whole community benefits.

Do you have any advice for teams that would like to follow in your footsteps?

Measure. When working on performance optimization it is easy to find two extremes: those who don't do anything about it, and those who do too much. Find a sweet spot, where you make sure you are delivering a good experience and are aware when you are reaching diminishing returns.

Finally, write code that is easy to remove. We think we can create the definitive product with a stack that will remain forever. However, we make decisions based on the current state of the technology and business insights. Those will change over time, and a good architecture makes it possible to introduce changes over time, replacing parts of it with better alternatives and removing sections that are not needed anymore.

Has the site changed significantly since the case study was published?

The web player received new features without incurring additional data usage. It became a PWA and could be installed on Chromebooks providing a more app-like experience. For a long time, we had maintained another set of pages with a similar content and design, built in a different stack. These pages were server-side rendered and optimized for SEO. The new web player made it possible to merge both projects, simplifying the codebase and removing lots of custom logic to decide what version should be rendered.

In summary, it helped reduce duplication and made the engineering team move faster.

Mainline Menswear's Success Building a PWA

By Charis Theodoulou, Natasha Kosoglov, & Thomas Steiner

Mainline is an online clothing retailer that offers the biggest designer brand names in fashion.[1] The UK-based company entrusts its team of in-house experts, blended strategically with key partners, to provide a frictionless shopping experience for all. With market presence in over 100 countries via seven custom-built territorial websites and an app, Mainline will continue to ensure the ecommerce offering is rivaling the competition.

Challenge

Mainline Menswear's goal was to complement the current mobile-optimized website with progressive features that would adhere to its "mobile first" vision, focusing on mobile-friendly design and functionality with a growing smartphone market in mind.

Solution

The objective was to build and launch a PWA that complemented the original mobile-friendly version of the Mainline Menswear website,[2] and then compare the stats to their hybrid mobile app, which is currently available on Android and iOS.

Once the app launched and was being used by a small section of Mainline Menswear users, they were able to determine the difference in key stats between PWA, app, and web.

1 The original version of this article was published in April 2021: https://smashed.by/mainline
2 https://www.mainlinemenswear.co.uk/

The approach Mainline took when converting its website to a PWA was to make sure that the framework they selected for their website (Nuxt.js, utilizing Vue.js) would be future-proof and enable them to take advantage of fast moving web technology.

Results

139% More pages per session in PWA vs. web.	**161%** Longer session durations in PWA vs. web.	**10%** Lower bounce rate in PWA vs. web
12.5% Higher average order value in PWA vs. web	**55%** Higher conversion rate in PWA vs. web.	**243%** Higher revenue per session in PWA vs. web.

Technical Deep Dive

Mainline Menswear is using the Nuxt.js framework[3] to bundle and render its site, which is a single page application (SPA).

GENERATING A SERVICE WORKER FILE

For generating the service worker, Mainline Menswear added configuration through a custom implementation of the `nuxt/pwa` Workbox module.[4]

The reason they forked the `nuxt/pwa` module was to allow the team to add more customizations to the service worker file that they weren't able to or had issues with when using the standard version. One such optimization was around the offline functionality[5] of the site; for example, serving a default offline page and gathering analytics while offline.

3 https://nuxtjs.org/
4 https://smashed.by/nuxtpwa
5 https://smashed.by/nuxtpwa

ANATOMY OF THE WEB APP MANIFEST

The team generated a manifest with icons for different mobile app icon sizes and other web app details like `name`, `description` and `theme_color`:

```
{

  "name": "Mainline Menswear",
  "short_name": "MMW",
  "description": "Shop mens designer clothes with Mainline
Menswear. Famous brands including Hugo Boss, Adidas, and
Emporio Armani.",
  "icons": [
   {
    "src": "/_nuxt/icons/icon_512.c2336e.png",
    "sizes": "512x512",
    "type": "image/png"
   }
  ],
  "theme_color": "#107cbb"
 }
```

The web app, once installed, can be launched from the home screen without the browser getting in the way. This is achieved by adding the `display` parameter in the web app manifest file:

```
{
  "display": "standalone"
}
```

Last but not least, the company is now able to easily track how many users are visiting the web app from the home screen by simply appending a `utm_source` parameter in the `start_url` field of the manifest:

```
{
   "start_url": "/?utm_source=pwa"
}
```

See "Add a web app manifest"[6] for a more in-depth explanation of all the web app manifest fields.

Runtime Caching for Faster Navigations

Caching for web apps is a must for page speed optimization and for providing a better user experience for returning users.

For caching on the web, there are quite a few different approaches.[7] The team is using a mix of the HTTP cache and the Cache API for caching assets on the client side.

The Cache API gives Mainline Menswear finer control over the cached assets, allowing them to apply complex strategies to each file type. While all this sounds complicated and hard to set up and maintain, Workbox provides the team with an easy way of declaring such complex strategies and eases the pain of maintenance.

Workbox provides the team with an easy way of declaring complex strategies and eases the pain of maintenance.

CACHING CSS AND JS

For CSS and JS files, the team chose to cache them and serve them over the cache using the `StaleWhileRevalidate` Workbox strategy. This strategy allows them to serve all Nuxt CSS and JS files fast, which significantly increases the site's performance. At the same time, the files are being updated in the background to the latest version for the next visit:

```
/* sw.js */
workbox.routing.registerRoute(
```

6 https://smashed.by/addmanifest
7 https://smashed.by/cachingapproaches

```
  /\/_nuxt\/.*(?:js|css)$/,
  new workbox.strategies.StaleWhileRevalidate({
   cacheName: 'css_js',
  }),
  'GET',
 );
```

CACHING GOOGLE FONTS

The strategy for caching Google fonts depends on two file types:

- The style sheet that contains the `@font-face` declarations.
- The underlying font files (requested within the style sheet mentioned above).

```
// Cache the Google Fonts stylesheets with a stale-while-
revalidate strategy.
workbox.routing.registerRoute(
  /https:\/\/fonts\.googleapis\.com\/*/,
  new workbox.strategies.StaleWhileRevalidate({
   cacheName: 'google_fonts_stylesheets',
  }),
  'GET',
 );

 // Cache the underlying font files with a cache-first
strategy for 1 year.
 workbox.routing.registerRoute(
  /https:\/\/fonts\.gstatic\.com\/*/,
  new workbox.strategies.CacheFirst({
   cacheName: 'google_fonts_webfonts',
   plugins: [
    new workbox.cacheableResponse.CacheableResponsePlugin({
     statuses: [0, 200],
    }),
    new workbox.expiration.ExpirationPlugin({
      maxAgeSeconds: 60 * 60 * 24 * 365, // 1 year
      maxEntries: 30,
    }),
  ],
 }),
 'GET',
);
```

A full example of the common Google fonts strategy can be found in the Workbox Docs.

CACHING IMAGES

For images, Mainline Menswear decided to go with two strategies. The first strategy applies to all images coming from its CDN, which are usually product images. Their pages are image-heavy so the team is conscious of not taking too much of users' device storage.

So through Workbox, they added a strategy that is caching images coming only from the CDN with a maximum of 60 images using the `ExpirationPlugin`.[8]

The sixty-first (newest) image requested replaces the first (oldest) image so that no more than sixty product images are cached at any point in time.

```
workbox.routing.registerRoute(
  ({ url, request }) =>
   url.origin === 'https://mainline-menswear-res.cloudinary.
com' &&
   request.destination === 'image',
  new workbox.strategies.StaleWhileRevalidate({
```

8 https://smashed.by/expirationplugin

```
    cacheName: 'product_images',
    plugins: [
     new workbox.expiration.ExpirationPlugin({
      // Only cache 60 images.
      maxEntries: 60,
      purgeOnQuotaError: true,
     }),
    ],
   }),
  );
```

The second image strategy handles the rest of the images being requested by the origin. These images tend to be very few and small across the whole origin, but to be on the safe side, the number of these cached images is also limited to sixty.

```
workbox.routing.registerRoute(
  /\.(?:png|gif|jpg|jpeg|svg|webp)$/,
  new workbox.strategies.StaleWhileRevalidate({
   cacheName: 'images',
   plugins: [


     new workbox.expiration.ExpirationPlugin({
      // Only cache 60 images.
      maxEntries: 60,
      purgeOnQuotaError: true,
     }),
    ],
   }),
  );
```

Objective
Even though the caching strategy is exactly the same as the previous one, by splitting images into two caches (`product_images` and `images`), it allows for more flexible updates to the strategies or caches.

Providing Offline Functionality

The offline page is precached right after the service worker is installed and activated. They do this by creating a list of all offline dependencies: the offline HTML file, and an offline SVG icon.

```
const OFFLINE_HTML = '/offline/offline.html';
const PRECACHE = [
{ url: OFFLINE_HTML, revision:
'70f044fda3e9647a98f084763ae2c32a' },


{ url: '/offline/offline.svg', revision:
'efe016c546d7ba9f20aefc0afa9fc74a' },
];
```

The precache list is then fed into Workbox, which takes care of all the heavy lifting of adding the URLs to the cache, checking for any revision mismatch, updating, and serving the precached files with a `CacheFirst` strategy.

```
workbox.precaching.precacheAndRoute(PRECACHE);
```

HANDLING OFFLINE NAVIGATIONS

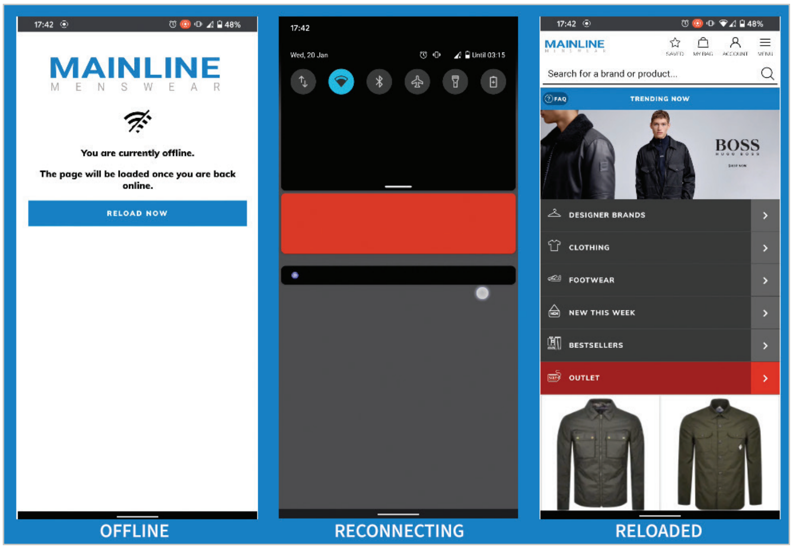
Once the service worker activates and the offline page is pre-cached, it is then used to respond to offline navigation requests by the user. While Mainline Menswear's web app is an SPA, the offline page shows only after the page reloads, the user closes and reopens the browser tab, or when the web app is launched from the home screen while offline.

To achieve this, Mainline Menswear provided a fallback to failed `NavigationRoute`[9] requests with the precached offline page:

9 https://smashed.by/navigationroute

```
const htmlHandler = new workbox.strategies.NetworkOnly();
const navigationRoute = new workbox.routing.NavigationRoute(({
event }) => {
 const request = event.request;
 // A NavigationRoute matches navigation requests in the
browser, i.e. requests for HTML
 return htmlHandler.handle({ event, request }).catch(() =>
caches.match(OFFLINE_HTML, {
   ignoreSearch: true
 }));
});
workbox.routing.registerRoute(navigationRoute);
```

Results



Offline page example as seen on www.mainlinemenswear.co.uk.

REPORTING SUCCESSFUL INSTALLS

Apart from the home screen launch tracking (with `"start_url": "/?utm_source=pwa"` in the web application manifest), the web app also reports successful app installs by listening to the `appinstalled` event on `window`:

```
window.addEventListener('appinstalled', (evt) => {
  ga('send', 'event', 'Install', 'Success');
 });
```

Andy Hoyle, head of development, said: "Adding PWA capabilities to your website will further enhance your customers' experience of shopping with you, and will be quicker to market than a platform-specific app."

Mainline Menswear Key Takeaways

Adding PWA capabilities to the website helped to enhance customer experience and boost revenue per session by 243%.

Mainline is a UK-based online clothing retailer with a market presence in over 100 countries. Mainline wanted to add progressive features to their menswear website while focusing on a mobile-friendly design. They launched the PWA app to a small section of their users and noticed a vast improvement in key stats due to the improved customer experience provided by the PWA. Some of the key technical enhancements in the PWA were as follows.

- Implemented in Nuxt.js + Vue.js framework to take advantage of the latest technology available to them at the time.
- A service worker file helps to make the app available offline.
- The web app manifest provides the ability to launch from the home screen.
- HTTP Cache and Cache api provide client-side caching of css, JS, fonts, and images.

Deprecating Excalidraw for Electron

By Thomas Steiner

On the Excalidraw project,[1] we have decided to deprecate Excalidraw Desktop, an Electron[2] wrapper for Excalidraw, in favor of the web version that you can – and always could – find at *excalidraw.com*. After a careful analysis, we have decided that a progressive web app (PWA) is the future we want to build on. Read on to learn why.

How Excalidraw Desktop Came into Being

Soon after Christopher Chedeau created the initial version of Excalidraw in January 2020 and blogged about it,[3] he proposed the following in Issue #561:

> *Would be great to wrap Excalidraw within Electron (or equivalent) and publish it as a [platform-specific] application to the various app stores.*

The immediate reaction by Guillermo Peralta Scura was to suggest:

> *What about making it a PWA instead? Android currently supports adding them to the Play Store as Trusted Web Activities and hopefully iOS will do the same soon. On Desktop, Chrome lets you download a desktop shortcut to a PWA.*

The decision that Chedeau took in the end was simple:

> *We should do both :)*

1 https://excalidraw.com/
2 https://www.electronjs.org/
3 https://smashed.by/excalidrawblog

While work on converting the version of Excalidraw into a PWA was started by Scura and later others, Panayiotis Lipiridis went ahead independently and created a separate repo[4] for Excalidraw Desktop.

To this day, the initial goal set by Chedeau – that is, to submit Excalidraw to the various app stores – has not yet been reached. Honestly, no one has even started the submission process to any of the stores. But why is that? Before I answer, let's look at Electron, the platform.

What Is Electron?

The unique selling point of Electron is that it allows you to "build cross-platform desktop apps with JavaScript, HTML, and CSS." Apps built with Electron are "compatible with Mac, Windows, and Linux"; that is, "Electron apps build and run on three platforms." According to the homepage, the hard parts that Electron makes easy are automatic updates, system-level menus and notifications, crash reporting, debugging and profiling, and Windows installers. Turns out, some of the promised features need a detailed look at the small print.

- For example, automatic updates "are [currently] only [supported] on macOS and Windows. There is no built-in support for auto-updater on Linux, so it is recommended to use the distribution's package manager to update your app."

- Developers can create system-level menus by calling `Menu.setApplicationMenu(menu)`. On Windows and Linux, the menu will be set as each window's top menu, while on macOS there are many system-defined standard menus, like the Services[5] menu. To make menus standard menus, developers should set their menu's `role` accordingly, and Electron will recognize them and make them become standard menus. This means that a lot

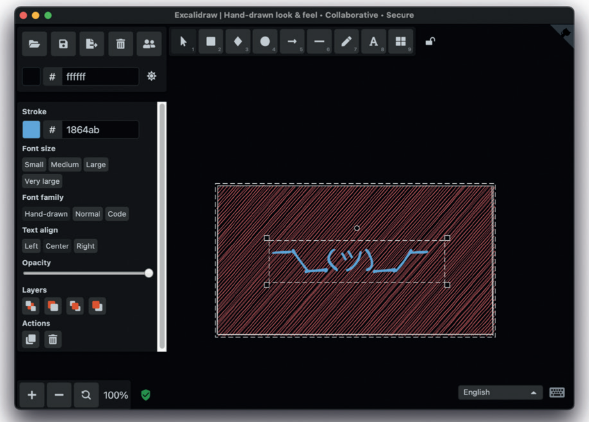
4 https://smashed.by/excalidrawdesktop
5 https://smashed.by/servicesmenu

of menu-related code will use the following platform check: `const isMac = process.platform === 'darwin'.`

- Windows installers can be made with windows-installer.[6] The README of the project highlights that "for a production app you need to sign your application. Internet Explorer's SmartScreen filter will block your app from being downloaded, and many anti-virus vendors will consider your app as malware unless you obtain a valid certificate."

Looking at just these three examples, it is clear that Electron is far from "write once, run everywhere." Distributing an app on app stores requires code signing, a security technology for certifying app ownership. Packaging an app requires using tools like electron-forge and thinking about where to host packages for app updates. It gets complex relatively quickly, especially when the objective truly is cross-platform support. I want to note that it is absolutely possible to create stunning Electron apps with enough effort and dedication. For Excalidraw Desktop, we were not there.

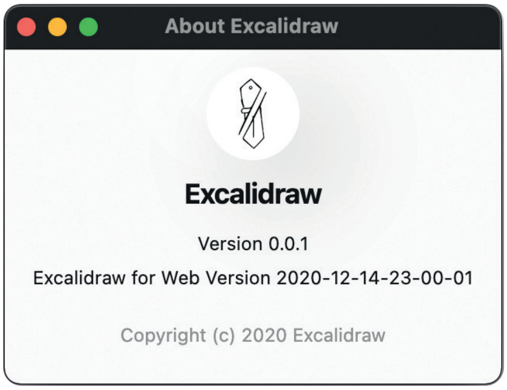
Where Excalidraw Desktop Left Off



Excalidraw Desktop is almost indistinguishable from the web version.

6 https://smashed.by/windowsinstaller

Excalidraw Desktop so far is basically the Excalidraw web app bundled as an *.asar* file with an added **About Excalidraw** window. The look and feel of the application is almost identical to the web version.


About Excalidraw

Excalidraw

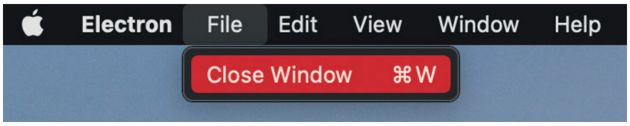
Version 0.0.1

Excalidraw for Web Version 2020-12-14-23-00-01

Copyright (c) 2020 Excalidraw

*The **About Excalidraw** menu providing insights into the versions.*

On macOS, there is now a system-level menu at the top of the application, but since none of the menu actions – apart from **Close Window** and **About Excalidraw** – are hooked up to anything, the menu is, in its current state, pretty useless. Meanwhile, all actions can, of course, be performed via the regular Excalidraw toolbars and the context menu.



The menu bar of Excalidraw Desktop on macOS.

We use electron-builder, which supports file type associations. By double-clicking an *.excalidraw* file, ideally the Excalidraw Desktop app should open. The relevant excerpt of our *electron-builder.json* file looks like this:

```
{
 "fileAssociations": [
  {
   "ext": "excalidraw",
   "name": "Excalidraw",
   "description": "Excalidraw file",
   "role": "Editor",
   "mimeType": "application/json"
  }
 ]
}
```

Unfortunately, in practice this does not always work as intended, since, depending on the installation type (for the current user, for all users), apps on Windows 10 do not have the rights to associate a file type to themselves.

These shortcomings and the pending work to make the experience truly app-like on all platforms (which, again, with enough effort is possible) were a strong argument for us to reconsider our investment in Excalidraw Desktop. The way bigger argument for us, though, was that we foresee that for our use case, we do not need all the features Electron offers. The grown and still growing set of capabilities of the web serves us equally well, if not better.

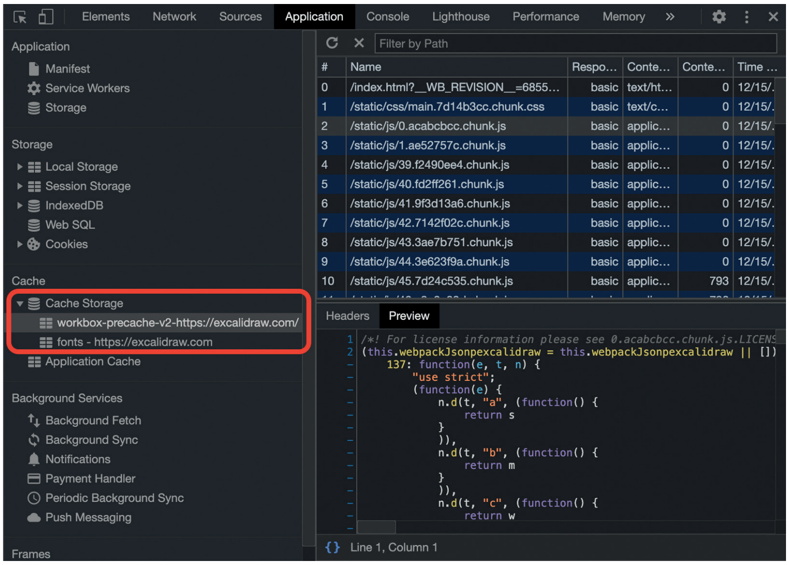
How the Web Serves Us Today and In the Future

Even in 2020, jQuery[7] is still incredibly popular. For many developers it has become a habit to use it, despite the fact that today they might not need jQuery.[8] There is a similar resource for Electron, aptly called You Might Not Need Electron.[9] Let me outline why we think we do not need Electron.

7 https://jquery.com/
8 http://youmightnotneedjquery.com/
9 https://youmightnotneedelectron.com/

INSTALLABLE PROGRESSIVE WEB APP

Excalidraw today is an installable progressive web app with a service worker and a web app manifest. It caches all its resources in two caches, one for fonts and font-related CSS, and one for everything else.

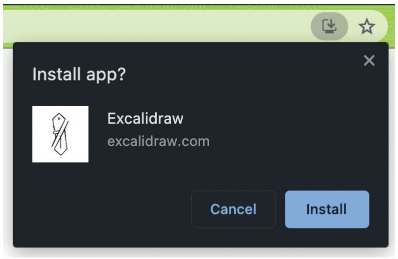


#	Name	Respo…	Conte…	Conte…	Time …
0	/index.html?__WB_REVISION__=6855…	basic	text/ht…	0	12/15/.
1	/static/css/main.7d14b3cc.chunk.css	basic	text/c…	0	12/15/.
2	/static/js/0.acabcbcc.chunk.js	basic	applic…	0	12/15/.
3	/static/js/1.ae52757c.chunk.js	basic	applic…	0	12/15/.
4	/static/js/39.f2490ee4.chunk.js	basic	applic…	0	12/15/.
5	/static/js/40.fd2ff261.chunk.js	basic	applic…	0	12/15/.
6	/static/js/41.9f3d13a6.chunk.js	basic	applic…	0	12/15/.
7	/static/js/42.7142f02c.chunk.js	basic	applic…	0	12/15/.
8	/static/js/43.3ae7b751.chunk.js	basic	applic…	0	12/15/.
9	/static/js/44.3e623f9a.chunk.js	basic	applic…	0	12/15/.
10	/static/js/45.7d24c535.chunk.js	basic	applic…	793	12/15/.

```
/*! For license information please see 0.acabcbcc.chunk.js.LICENS
(this.webpackJsonpexcalidraw = this.webpackJsonpexcalidraw || [])
    137: function(e, t, n) {
        "use strict";
        (function(e) {
            n.d(t, "a", (function() {
                return s
            }
            )),
            n.d(t, "b", (function() {
                return m
            }
            )),
            n.d(t, "c", (function() {
                return w
```

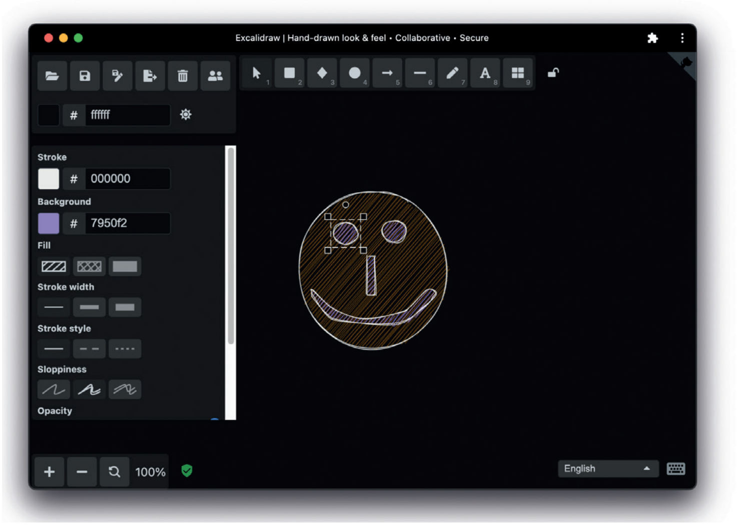
Excalidraw's cache contents.

This means the application is fully offline-capable and can run without a network connection. Chromium-based browsers on both desktop and mobile prompt the user to install the app. You can see the installation prompt in the screenshot below.

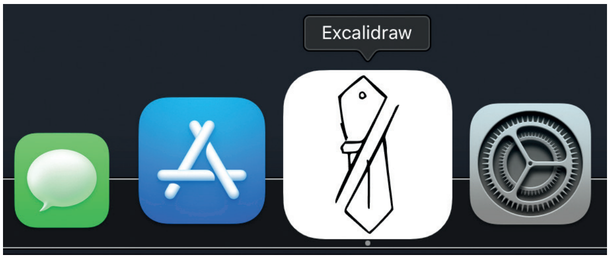


The Excalidraw install dialog in Chrome.

Excalidraw is configured to run as a standalone application, so when you install it you get an app that runs in its own window. It is fully integrated in the operating system's multitasking UI and gets its own app icon on the home screen, dock, or taskbar, depending on the platform where you install it.



The Excalidraw PWA in a standalone window.



The Excalidraw icon on the macOS dock.

FILE SYSTEM ACCESS

Excalidraw uses browser-fs-access[10] for accessing the file system of the operating system. On supporting browsers, this allows for a true **Open > Edit > Save** workflow and actual over-saving and Save As ...

10 https://smashed.by/browserfsaccess

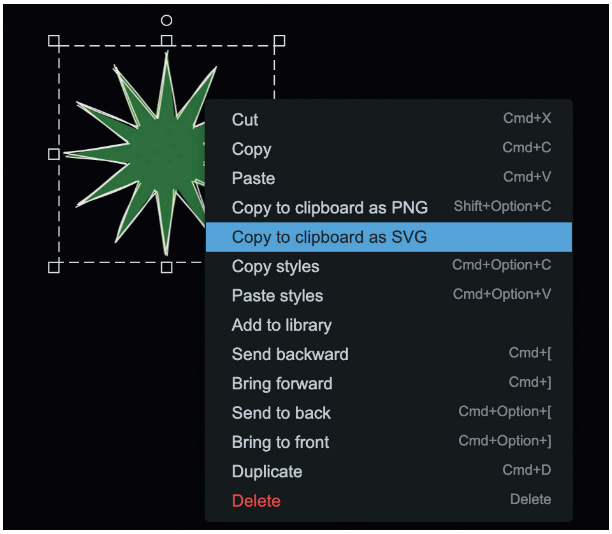
with a transparent fallback for other browsers. You can learn more about this feature in my blog post "Reading and writing files and directories with the browser-fs-access library."[11]

DRAG-AND-DROP SUPPORT

Files can be dragged and dropped onto the Excalidraw window just as in platform-specific applications. On a browser that supports the File System Access API, a dropped file can be immediately edited and the modifications saved to the original file. This is so intuitive that you sometimes forget you are dealing with a web app.

CLIPBOARD ACCESS

Excalidraw works well with the operating system's clipboard. Entire Excalidraw drawings or also just individual objects can be copied and pasted in *image/png* and *image/svg+xml* formats, allowing for an easy integration with other platform-specific tools like Inkscape,[12] or web-based tools like SVGOMG.[13]



The Excalidraw context menu offering clipboard actions.

11 https://smashed.by/fsaccesslibrary
12 https://inkscape.org/
13 https://smashed.by/svgomg

FILE HANDLING

Excalidraw already supports the experimental File Handling API, which means *.excalidraw* files can be double-clicked in the operating system's file manager and opened directly in the Excalidraw app, since Excalidraw registers as a file handler for *.excalidraw* files in the operating system.

DECLARATIVE LINK CAPTURING

Excalidraw drawings can be shared by link. Here is an example: https://smashed.by/excalidrawsharing

In the future, if people have Excalidraw installed as a PWA, such links will not open in a browser tab, but launch a new standalone window. Pending implementation, this will work thanks to declarative link capturing,[14] a bleeding-edge proposal for a new web platform feature (at the time of writing).

Excalidraw is at the forefront of what is possible in the browser, all while acknowledging that not all browsers on all platforms support each feature we use.

Conclusion

The web has come a long way, with more and more features landing in browsers that only a couple of years or even months ago were unthinkable on the web and exclusive to platform-specific applications. Excalidraw is at the forefront of what is possible in the browser, all while acknowledging that not all browsers on all platforms support

14 https://smashed.by/declarativecapturing

each feature we use. By betting on a progressive enhancement strategy, we enjoy the latest and greatest wherever possible, but without leaving anyone behind. Best viewed in any browser.

Electron has served us well, but in 2020 and beyond, we can live without it. Oh, and for that objective of @vjeux: since the Android Play Store now accepts PWAs in a container format called trusted web activity and since the Microsoft Store supports PWAs too, you can expect Excalidraw in these stores in the not too distant future. Meanwhile, you can always use and install Excalidraw in and from the browser.

Excalidraw Key Takeaways

A desktop PWA instead of a platform-specific app that leaves no one behind and is best viewed in any browser.

The team decided to release the Excalidraw desktop app by wrapping the web app using an electron wrapper so that it may be released to different app stores. Another team simultaneously went to work on a PWA. However, the desktop app did not take off and was soon deprecated in favor of the PWA as it was easier to maintain.

The Excalidraw PWA comes with a service worker and web app manifest that allows caching and makes it installable through Chromium-based browsers on both desktop and mobile. It supports various native app features like stand-alone application mode, launch from the home screen, dock, or taskbar, file-system access through browser-fs-access, clipboard access, and many more. Each feature is available to users based on browser and platform support.

Interview

Christopher Chedeau

Creator, Excalidraw

Author of **"Deprecating Excalidraw for Electron"**

What excited you or your team the most about the work in the case study?

What's amazing about the browser environment nowadays is that it provides all the APIs to build a product that would have needed a desktop app environment in the past. This enabled Excalidraw, a virtual whiteboard, to be one URL away from being used by hundreds of thousands of people.

The full list is probably too long for this answer but the highlights are:

- the file system API for interacting with the actual file system on the machine
- the web crypto API for enabling the product to be end-to-end encrypted
- the canvas API for high performance rendering of the diagrams
- the web socket API to be able to implement real time collaboration
- the progressive web app API to be able to install it locally…

I've worked in the space long enough to have seen each of those APIs come to life, and it feels like cheating to have access to all of them.

Were you surprised by the mpact your work had on the overall user experience, business, team, or other metrics?

This project started as a way to procrastinate from writing peer feedback; it wasn't meant to actually turn into a real product. But after hacking on this for a few days in the open on GitHub and Twitter, many people started contributing and it was starting to be useful.

In retrospect, two product decisions that were made were key to the early success:

1. There's no login flow: you pop in the website and are able to start drawing instantly.

2. We used Rough.js and the Virgil font for a hand-drawn effect to give the drawings a very distinctive look that begs people to ask how it was made.

What really changed the trajectory was that three months after the inception, Covid hit. Everyone was sent home and many people started looking for a virtual whiteboard – and they found excalidraw. The team scrambled to implement real-time collaboration during that weekend and the growth continued till today.

I had absolutely no idea that this would end up being used by hundreds of thousands of people and being the full-time job of multiple people. This is pretty incredible: the time we live in where this kind of thing can happen!

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to

put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

Weirdly enough, all the technical decisions that were made on *excalidraw.com* really stood the test of time. The codebase is in a healthy state and the product is very good. I still use it daily for a wide variety of tasks!

The one thing I wish I had done differently is start the work on the SaaS product sooner. I used to think that having a paid product would degrade the open source experience, but it was actually the opposite that happened.

Because Excalidraw is dealing with very private and confidential information, even though the open source product is local first and – when collaboration is active – end-to-end encrypted (meaning that the team and hosting providers don't have access to any of the drawings), many companies were not comfortable having their employees use it. They wanted to have a real company backing the product and pay for it instead.

The SaaS product offering fills a specific need that the open source product could not and is used to pay actual people to work on the open source project full-time. This is one of those situations where every party is winning! It took us more than a year to eventually convince ourselves to do it and could have happened way sooner.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

Quality over features. Excalidraw has a relatively small number of features and customization compared to the hundreds of projects that have been in the space over the years, but we obsess over all the small interaction details and pride ourselves for having zero production exceptions.

Let me give you four examples out of hundreds of small details that are really important for the user experience:

1. If you select a really small object on screen, we're going to remove the middle handles so that the click target is not confusing.

2. We update the selected elements when you drag in real time so that you are not surprised by what you select.

3. We have visible hotkeys on every action you can make so you can minimize your mouse movement.

4. When you zoom, we wait until you stop zooming to properly redraw the scene and give you a fast resize of the existing details in the meantime so that you're not dropping frames.

This has been a value that I've learned through my career working at Meta and its open source projects. As I built the initial team that is now running Excalidraw, this is something I kept pushing for and is now in the DNA of the company.

Do you have any advice for teams that would like to follow in your footsteps?

Excalidraw was really made possible thanks to open source. It started as a hack on which dozens of people implemented significant pieces. The most incredible part is that everyone who contributed brought their unique skills and talent to the table.

There isn't any single person who would have been able to do all the following things that made Excalidraw successful:

- Someone was able to get the product translated into many languages, rallying many translators together.
- Someone implemented an end-to-end encrypted live collaboration setup.
- Someone implemented deep integration with the browser to be able to interact with the device file system.
- Someone implemented an intuitive way to attach arrows to shapes.

And the list goes on. When you are starting up a project, having access to so many talented people really makes a difference and open source makes it possible!

Has the site changed significantly since the case study was published?

It did significantly change as we improved the experience in hundreds of small ways. It's hard to see the effect on *excalidraw.com* since we ship them continuously to users but we were able to see it "by accident" within Meta.

We did a one time import of the Excalidraw source code to make it available to Meta employees and integrate it with internal tooling. A year later we updated it to use the npm package designed for integrations like this one, not thinking twice about it since there were no new big features. The usage ramped up a lot faster after that. It's a testament that all the small things do add up.

1
2
3
888
ACCESSIBILITY

Introduction

Accessibility is a critical quality of products as it directly impacts how users with disabilities can perceive, understand, navigate, and interact with them equally without barriers. For a website, it means that the site's content and functionality are readily available to all who land on it.

Accessibility (sometimes abbreviated to a11y) refers to the experience of users who might be outside the narrow range of the "typical" user and who might access or interact with things differently than you expect. Accessibility addresses the experience for users with some impairment or disability – the disability might be non-physical or temporary. For example, not being able to read something on the screen in bright daylight is a temporary disability. Although we tend to center our discussion of accessibility on users with physical impairments, we can all relate to the experience of using an interface that is not accessible to us for other reasons.

Accessibility in web apps makes them viable for diverse users, including people with temporary or permanent disabilities. Accessible web apps are built with an inclusive mindset and are considerate of users who may be using their computers or mobile devices differently. When using computers and phones, people with visual, auditory, physical, or cognitive impairments may rely on assistive technologies such as screen readers, captions, or eye-tracking software. By following accessibility standards, websites can ensure support for commonly used devices.

This section discusses how addressing accessibility issues in a broader and more general sense almost always improves the user experience for everyone. We will discuss standard practices that help you develop accessible web apps and present case studies on how different apps built accessibility into their solutions.

Why Accessibility Is Important

The web is an important resource that lets us connect and helps businesses, governments, and other organizations reach people with diverse skills, abilities, and needs. Web applications should be accessible to all, irrespective of their physical or social abilities. Accessibility should be built-in as part of the user experience. Accessibility provides equal access and opportunities to people with diverse abilities, and supports the social inclusion of people with temporary or permanent disabilities, and older people. There is also a strong business case for accessibility.[1]

1. Designing for accessibility improves the overall user experience and customer satisfaction in a variety of situations, across different devices, and for older users. There is an overlap between mobile web best practices[2] and Web Content Accessibility Guidelines.[3] Thus, designing to meet these guidelines helps make web apps more accessible to everyone regardless of their situation, environment, or device.

2. Accessible solutions can enhance brand value and extend market reach.

3. Web accessibility is often required by law,[4] and implementing it can reduce potential legal risk.

Many aspects of accessibility are pretty easy to understand and implement. Some accessibility solutions are more complex and take more knowledge to implement.

1 https://smashed.by/businesscase
2 https://smashed.by/mobilebp
3 https://smashed.by/wcag
4 https://smashed.by/waipolicies

It is most efficient and effective to incorporate accessibility from the very beginning of projects, so you don't need to go back and rework.

Barriers to Accessibility

An accessibility barrier is anything that restricts you from accessing web content. The barrier could be permanent (blindness, for example), temporary (such as an inability to use your hands after an accident), or momentary (like difficulty typing in a packed commuter train). These barriers can be categorized based on the type of function affected.

- **Vision impairments:** These can range from low vision to complete blindness. Tools and techniques used by this group of people to access web content include screen magnification, text-to-speech, high-contrast themes, screen readers, braille displays, and so on.
- **Motor/dexterity impairments:** These can affect users' ability to use keyboards, mouse, touchscreens, and other regular input devices. They may rely on voice access, head or eye-tracking software, or switch devices to communicate with web apps.
- **Auditory impairments:** These are hearing impairments where users may rely on captions as an alternative to speech and sound on the web application interface.
- **Cognitive impairments:** These could include ADHD, dyslexia, and autism. Generally, users in this group would prefer minimal distractions and avoid websites with flashy content, animations, or layout shifts.

Understanding the limitations and needs of each of the above categories helps design solutions that eliminate accessibility barriers.

With this background, let us see how you can make your website accessible for some key input and output devices.

How Do I Make My UI Components Accessible?

When designing accessible components for your website, keep the following questions in mind and ensure that you can confidently answer them with a "Yes."

- Can I use this component without sight or sound?
- Is the user experience good, even with impairments; for example, are buttons large enough to click without fine motor control?
- If I rely on keyboard actions, are they standard? If not, are they documented?
  - Generally, you will want to avoid creating custom shortcuts, as many screen readers have their own shortcuts, and yours might disrupt those.
- Have I ensured that generic events exposed from my components do not affect its accessibility?
  - For example, a section in a tree view should fire a collapse/expand event, rather than bubbling up the click, to ensure that the user of your component does not forget to handle collapse/expand using arrow keys.

There are a lot of subtleties involved in getting accessibility right, so do not hesitate to reach out to an expert. For a few specific answers to the above questions, let us see how you can make your website accessible for some key input and output devices.

Enabling Accessibility for Devices

Accessibility for different devices is relevant to various groups of people. This section discusses a few pointers that would allow comfortable access to websites through commonly used devices.

KEYBOARDS

Users may rely exclusively on keyboards due to motor impairments or find keyboard shortcuts more efficient than mouse inputs. To make keyboard access easier for everyone, you can:

1. Arrange elements on the screen logically from left to right and top to bottom based on how the user would need them. Doing this allows users to press the **Tab** key and focus on every element they need to access in the correct sequence without using a mouse to get to it. You can use `tabindex`[5] to insert or remove an element from the natural tab order.

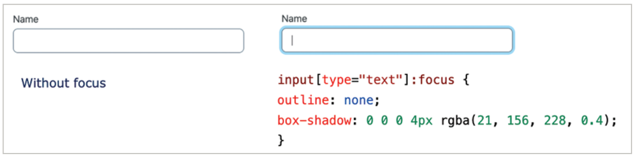
2. Ensure that hidden elements that only become visible because of some user action do not receive focus.[6] You can use one of the following CSS properties for this:
   - `display: none;`
   - `visibility: hidden;`

3. Use the correct semantic HTML element for the desired function to improve accessibility as they have built-in support for `tabindex`, interactions on mobile, and support for specific actions. For example, using a `div` or `link` to implement the button functionality is a common accessibility anti-pattern. The `div` element does not support[7] synthetic click activation (click on **Enter** or **space**) by itself, while screen readers an-

5 https://smashed.by/controlfocus
6 https://smashed.by/offscreenvisibility
7 https://smashed.by/buttonvsdiv

nounce the link element differently. Extra effort is required to make buttons using these elements accessible.

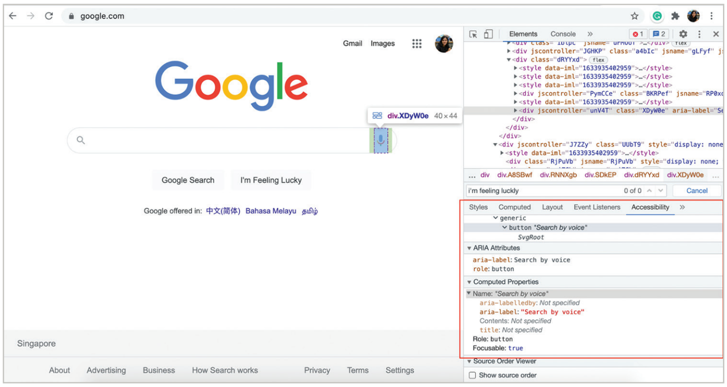
4. Use a focus indicator to highlight the currently active element to the user. This helps users who do not have a mouse pointer. Defining a style for `:focus` in the CSS for different elements helps create a focus indicator, as shown for a textbox here:



```
input[type="text"]:focus {
outline: none;
box-shadow: 0 0 4px rgba(21, 156, 228, 0.4);
}
```

SCREEN READERS

Screen readers can provide an alternative UI if developers have used semantically-rich HTML to create pages. Browsers create the accessibility tree for the page based on the semantic HTML contained in it. For example, the following figure shows the accessibility details available for the "Search by voice" button on the Google search home page, as seen in the DevTools/Elements tab.



```
▶<div jscontroller="JGHKP" class="a4bIc" jsname="gLFyf" j
▼<div class="dRYYxd"> flex
  ▶<style data-iml="1633935402959">…</style>
  ▶<style data-iml="1633935402959">…</style>
  ▶<style data-iml="1633935402959">…</style>
  ▶<div jscontroller="PymCCe" class="BKRPef" jsname="RP0xc
  ▶<style data-iml="1633935402959">…</style>
  ▶<div jscontroller="unV4T" class="XDyW0e" aria-label="Se
 </div>
</div>
</div>
▼<div jscontroller="J7ZZy" class="UUbT9" style="display: none
 ▶<style data-iml="1633935402959">…</style>
  <div class="RjPuVb" jsname="RjPuVb" style="display: none;
```

```
... div div.A8SBwf div.RNNXgb div.SDkEP div.dRYYxd div.XDyW0e ...
i'm feeling luckly    0 of 0
```

```
  ˅ generic
    ˅ button "Search by voice"
        SvgRoot
▼ ARIA Attributes
  aria-label: Search by voice
  role: button
▼ Computed Properties
▼ Name: "Search by voice"
    aria-labelledby: Not specified
    aria-label: "Search by voice"
    Contents: Not specified
    title: Not specified
  Role: button
  Focusable: true
▼ Source Order Viewer
☐ Show source order
```

*ARIA attributes available for the "**Search by voice**" button.*

The screen reader would typically read the name and role of this element: in this case, “Search by voice button.” On the other hand, if you have used a `div` for displaying a button, the semantics would be different, and the screen reader would only read the text inside the `div`: “Search by voice.” Here are a few steps that can help to make web pages accessible to screen readers.

1. Use headings to outline the page so that the user can form a mental picture as the screen reader navigates the headings. Also, ensure that you do not skip heading levels to rely on browsers default styles for different levels of headings. Missing levels break the outline, which the screen reader requires for navigation.

2. Use HTML5 landmark elements such as `main`, `section`, and `nav` to demarcate special sections of the page, to which the screen readers can jump during navigation. You can also use skip links to allow readers to jump to specific sections of the page:

```
  <a class="skip-link" href="#main">Skip to
main</a>
           [Some content]
       <main id="main">
           [Main content]
       </main>
```

3. Elements should have accessible names and text alternatives so that the screen reader can read them out. In the previous example for Google search, the icon button “Search by voice” got its name because the `aria-label` attribute was included. Similarly, you can specify accessible names or text for all HTML elements.[8]

8 https://smashed.by/missingnames

4. Video elements[9] embedded in web pages can be made accessible by using track elements to the video. The `track` element links to a web video text tracks (WebVTT) file which contains a series of cues with the video timespan to which they are applicable. Cues such as captions, subtitles, and descriptions are included in the WebVTT file so that accessibility devices such as screen readers can read them.

TOUCH SCREENS

When designing for touch screens, including mobile devices and smartphones, you need to pay special attention to the placement of input elements, especially for people with motor impairments.

1. As per Android accessibility guidelines,[10] touch targets should be at least 48 × 48 PX in size. This size corresponds to the average finger pad area of a person.

2. Touch targets should also be placed at least 8 px apart to avoid overlap when tapping on different targets.

3. You can use The pointer CSS media feature[11] to change the style of an element for different types of screens, as shown.

```
@media (pointer: fine) { /* pointer fine for mouse */
    input[type="checkbox"]
       width: 15px;
       height: 15px;
    }
}
```

9 https://smashed.by/videoelements
10 https://smashed.by/touchtarget
11 https://smashed.by/pointer

```
@media (pointer: coarse) { /* pointer coarse for touch */
    input[type="checkbox"] {
       width: 30px;
       height: 30px;
     }
}
```

DISPLAY

Some best practices ensure that content displayed is accessible to all users, including those with slight visual impairments.

Colors used in the background and foreground should be sufficiently different, resulting in high enough contrast. WebAIM guidelines recommend a minimum of 4.5:1 contrast ratio. The high contrast compensates for the loss in contrast sensitivity usually experienced by older people.

Color vision deficiency affects over 300 million people worldwide. So if you are using colors alone to convey information (for example, in validation errors or charts), these people may not perceive the content.

Responsive design[12] is a good practice for users who access the website from devices with varied viewport sizes and those with vision impairments who zoom the page to access the content.

Accessibility Standards

The Web Accessibility Initiative (WAI) within the World Wide Web Consortium (W3C) is responsible for maintaining the Web Content

12 https://smashed.by/reponsivedesign

Accessibility Guidelines (WCAG). The current standard is WCAG 2.2.[13] It allows three levels of conformance to websites, levels A, AA, and AAA, going from conformance to basic requirements for some users with disabilities to enhanced requirements to cover a larger group of users. Most websites should aim for level AA of conformance. You can find a checklist corresponding to this guideline here.[14]

Additionally, WAI-ARIA (Accessible Rich Internet Applications) specifies the standard for increasing accessibility, especially for dynamic components that are rendered using JavaScript. ARIA attributes extend HTML markup by adding textual information to HTML semantic tags. These attributes were later incorporated into HTML5.

Testing for Accessibility

You can use the Lighthouse accessibility audits during the development life cycle to determine your accessibility score[15] and meet the requirements. Lighthouse audits the page for several accessibility requirements, such as alt text for images, landmark regions, names for different elements, track elements for videos, and others. The accessibility score is a weighted average of all accessibility audits. Each audit passes only if all elements relevant to the audit satisfy the specific accessibility requirement being audited.

Additionally, there are several automated accessibility testing tools for different platforms. These include:

1. Axe[16] and Tenon.io[17] for automated testing on different frameworks/browsers.

13 https://smashed.by/wcag21
14 https://smashed.by/WCAGchecklist
15 https://web.dev/accessibility-scoring/
16 https://smashed.by/axe
17 https://tenon.io/

2. eslint-plugin-jsx-a11y[18] for testing react components.

3. codelyzer[19] for testing in-editor testing for Angular.

4. tota11y[20] to visualize how your site performs with assistive technology.

5. Accessibility Inspector[21] and VoiceOver[22] utility on macOS.

6. Windows Automation API Testing Tools,[23] NVDA,[24] and AccProbe[25] on Windows.

However, manual testing is required to identify some issues. W3C recommends that you plan and design for accessibility early in the development life cycle. Teams should also evaluate accessibility early to address issues.

Case Studies

Let us now look at some websites that have successfully built accessibility as an integral part of their user experience.

18 https://smashed.by/eslint
19 https://smashed.by/codelyzer
20 https://smashed.by/tota11y
21 https://smashed.by/accessibilityinspector
22 https://smashed.by/voiceover
23 https://smashed.by/testtools
24 http://www.nvaccess.org/
25 http://accessibility.linuxfoundation.org/a11yweb/util/accprobe/

The Story of Making Wix Accessible

By Ohad Laufer

One of Wix's mottos is "The Way the Web was Meant to Be."[1] We believe that everyone should have a website, and of course, anyone should be able to access and use websites built using Wix tools.

Wix has many users, and these users are very diverse. Users have various devices, browsers, internet connections, and more. Users might also not be able to use all interfaces, due to different physical or cognitive impairments: they may have dyslexia, be blind or deaf, or have motor control impairments that make it difficult – or impossible – to use a mouse or trackpad.

In 2017, we at Wix decided to make sure our sites were accessible. When we say a site is accessible, it means that the site's content is available to and all of its functionality can be operated by anyone.

The goal was to "enable our users to create an accessible site." This is different from modifying existing sites to be accessible. Wix can't prevent our users from using low-contrast colors or small fonts, so we decided that the first phase of the project should focus on enabling users to build sites that are accessible using the existing Wix tools.

Our team took on this project after we saw the "Accessibility – Sady"[2] video by Apple and another one by Microsoft. We wanted to take on this challenge and believed in its importance and ability to make an actual difference. We had two months and a team of six developers

1 The original version of this case study was published in December 2017: https://smashed.by/wixcasestudy
2 https://smashed.by/sady

split between Israel and Ukraine. In this limited time we knew we couldn't cover all aspects of accessibility, but tried to make the most impact. The scope of the project was to:

- enable navigation by keyboard,
- provide screen reader compatibility,
- build infrastructure for accessibility development and testing.

Other assistive technologies and additional aspects of accessibility will be supported in future phases.

Enable Keyboard Navigation

Developers sometimes assume that all our users can interact with the sites using mouse, keyboard, and touch. People with some disabilities might not be able to use all these interfaces and depend on the most basic one: the keyboard. Keyboard navigation is usually done with the **Tab** or arrow keys. The user can use **Tab** to move forward in the site to the next interactive element, and **Shift + Tab** to move back. Interaction with focused elements is usually done using **Enter** or **Space** keys.

Focusable Elements

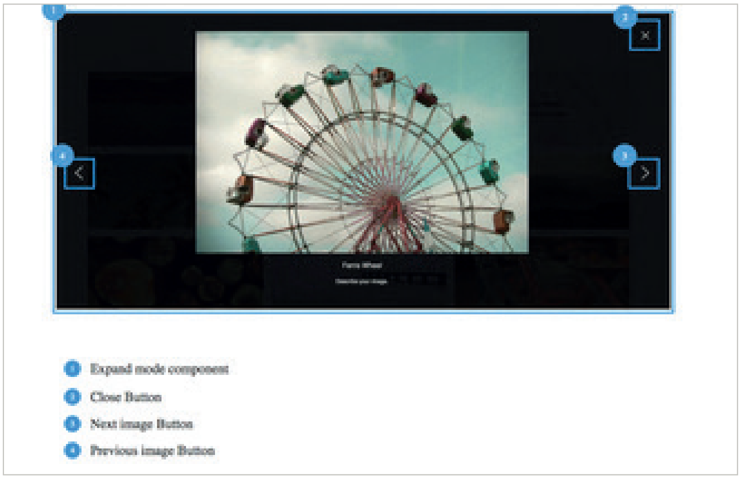
We wanted to make sure that all of the site's features were accessible via keyboard interactions: the user should be able to access and operate all interactive elements of the site. This means that all links, buttons, menus, and so on need to work well when using a keyboard. Not only do "simple" elements need to behave properly when used with a keyboard, but more complex elements, such as galleries, contact forms, and strips (a common Wix element that acts as a full-width container), need to have a logical behavior that supports keyboard interaction too.

With the help of our product manager and UX expert, and also an accessibility expert, we created a document that listed the desired behavior of each Wix component. There are over 150 such components, each with several display options. Our developers then went through all these and applied the desired behavior.

The best method was to use the proper semantic HTML elements. For instance, if we have a button component, it should use a `<button>` HTML tag. A page's navigation menu should use a `<nav>` tag. This may sound obvious, but often developers choose to mimic a native element behavior rather than simply use the most semantic element for the task.

If, for any reason, we couldn't change the HTML element in use, we had to resort to other, less semantic methods, such as using HTML's `tabindex` attribute. This overrides the native behavior of an element when using the keyboard by telling the browser to skip an element, to focus on it (even if it's not focusable by default), and can even control the order of the tabbing.

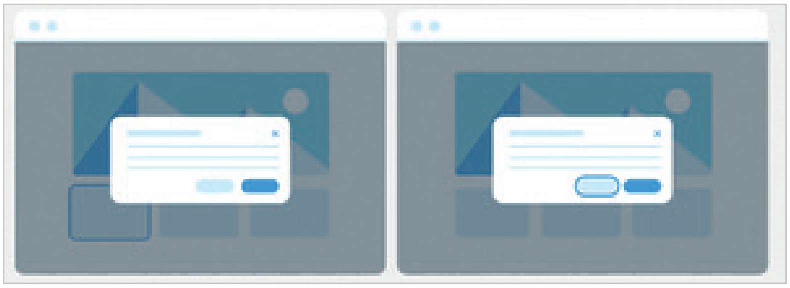
Once users can get to all the interactive elements, they need to get there in a meaningful order. For example, when navigating through matrix gallery images, the first **Tab** should land on the first image, then allow the user to use their arrow keys to move through the collection. The next **Tab** should move the focus to the "next page" button, and another **Tab** key press should move the focus to the "previous page" button. All these adjustments had to be made for each possible layout of the component.



Tab order in a gallery.

An additional aspect of keyboard navigation we had to handle were “focus traps.” We needed to make sure that elements that open as a modal window on top of the page (such as lightbox) allow navigation inside, but don’t let the focus out until the modal window is closed.

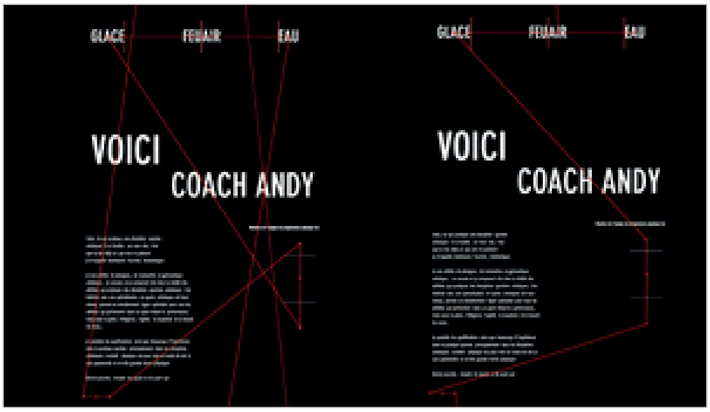
Conversely, we needed to make sure that only components that we deliberately made focus traps acted as such to prevent other components (usually external iframes) from getting the user’s focus and not releasing it as the user keeps on tabbing.



Focus trap: focus should be retained within the modal (right) and not leak outside it (left).

DOM Order

The last aspect of keyboard navigation is the need for the navigation path to follow the way the site looks. This means that if element X is directly before element Y on the site, the user should pass from element X to element Y when tabbing. We want to make this flow as intuitive as possible to the user. When navigating, the browser uses only the HTML document object model (DOM) order, so we had to make sure the DOM order reflected the actual order of components on screen. Luckily for us, we used an existing algorithm developed at Wix that analyzes the elements' position on the site and generates the proper order. We now allow users to explicitly run this algorithm on the site. In the future, this will be done behind the scenes for all users.



Wix site DOM order: the red line depicts the order in which tabbing occurs, before (left) and after (right).

Visual Focus

In addition to it being keyboard accessible, the element should display an indication when focused. This indication (visual focus) behaves as an alternative to the mouse cursor and lets users know which element they're interacting with.

At Wix, like many other websites, the visual focus was actively disabled up until now, rendering the sites inaccessible. The reason for that was a design one: the focus indication "didn't look good." In this project we now enabled users to toggle visual focus for their site on or off. We also tried to use the new `:focus-visible` standard[3] that displays the visual focus only when using the keyboard to navigate, hoping that it will make it less intrusive for mouse-using users.

Having completed all of the above, handling many components and lots of edge cases, we finally had a site that is fully navigable and functional using a keyboard only.

Make the Site Usable by Screen Readers

People with impaired vision use assistive technology tools such as a screen magnifier or screen reader to interact with their computer. A screen reader synthesizes a voice to read the site out loud to the user. Examples are VoiceOver, TalkBack, JAWS and NVDA.

Screen readers use an "accessibility tree"[4] created by the operating system and browser, and enable various interactions with the site. Screen readers read all elements, including non-actionable elements like text paragraphs and descriptions of images, using the DOM, which makes the DOM treatment discussed in the last section even more valuable.

Document Landmarks

We applied HTML document landmarks wherever we could, marking up the site header as `<header>`, the menu as `<nav>`, and more. These semantics tell the screen reader what each section does, thus enabling it to create a more accurate accessibility tree with more rel-

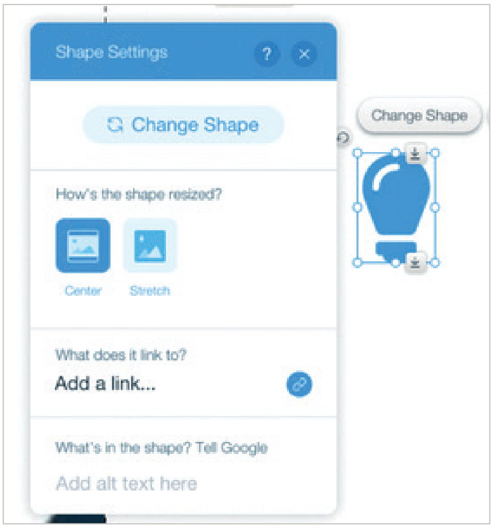
3 https://smashed.by/focusring
4 https://smashed.by/accessibilitytree

evant information about the site content. Many screen readers allow the user to jump past repeated content, such as header and site-wide navigation, and get straight to the main content.

ARIA Roles

Roles give information to assistive technologies about the behavior of an element, its current state, and how to interact with it, in case the native HTML can't be used or is not enough. One example is `aria-hidden` which makes a screen reader ignore an element. This is most commonly used when an element is heavily nested and we only want that element to be read, omitting all its predecessors. Another common use is `aria-live` which is used when something changes on the page and we want to notify the screen reader. We used it for such components as the contact form (for invalid field indications), lightboxes and image zoom mode.

We also used roles such as button to tell screen readers that an element should be read and operated as a button, when a different HTML element is used to mimic a button. As far as possible, we tried to limit using roles as much as possible, favoring the built-in behaviors of native HTML semantics to minimize the chance of unwanted side effects.



New shape settings includes `alt` text.

Titles and Alternative Text

It's simple – but crucial – to add titles, labels, and headings – and to all components. Wix has many graphic components that may only make sense when visible; for example, an icon shaped as a house. To a sighted person, it's obvious that it's a link to the homepage, but a person who doesn't see won't know that. Alternative text is a well-known best practice for images that convey meaning (photos, charts, and so on – not purely decorative ones), but descriptive text is needed for other graphic components as well, so we added the ability for the user to set it for any such kind of component.

The same goes for types of buttons and iframes that can be used for various purposes: adding a title or `aria-label` to these items tells the screen reader user the functionality of the specific element.

As a very desirable side effect, these additions also make the site more SEO-friendly: bots don't "see" the graphics either, but only text content, so in this case it's totally a win–win.

Build Infrastructure for Accessibility Development and Testing

When working on the project, we always kept in mind that accessibility is not a one-time effort. Accessibility needs to become an inseparable aspect of product development, just like we already consider performance, UX, mobile, and localization. Therefore, we didn't only work on the tasks already described, but also aspired to create tools for the future and for easier development, testing, and maintenance.

We built a collection of utility classes for handling keyboard interaction on components, getting "Tab-able" (that is, actionable) elements for easier navigation, and creation of focus traps. We also introduced testing tools and methods to address accessibility validation. We used both third-party libraries (axe-core)[5] and our own tools for easier component creation and validation.

All the knowledge we acquired during the process is continuously shared with other developers who work on different accessibility projects at Wix and is curated in different forms (readme files, presentations, this case study) for future developers joining our team.

Technical Scope

While working on this project, we used the WCAG 2.0 guidelines.[6] We found that there are no clear guidelines on how some elements (mainly complex ones) should behave, so we tried to rely on WCAG 2.0 as reference and guide while creating the specifications that best fit our needs.

Different operating systems, browsers and screen readers treat accessibility aspects very differently. As a rule of thumb, we set our scope on the following support matrix:

- Win 7/10 + Firefox + NVDA[7]
- macOS + Safari + VoiceOver

To verify the outcome of our project we were helped by an organization that employs people with disabilities who tested our websites and gave us vital feedback.

Summary

In this case study, I told the story of the work we did over two months making Wix sites accessible. Apart from the main goal of

5 https://smashed.by/axecoregit
6 https://smashed.by/WCAG20
7 http://www.nvaccess.org/

this project, we also had a greater mission: to create a new culture of development that considers accessibility as one of the cornerstones of our product.In the web world, as in the real world, there's always more than one way to achieve a goal. We aspired to set a new mindset of always preferring the native solution over a proprietary one.

Another important outcome of this project was the introduction of accessibility, its language, conversations, and practices to Wix developers, by creating a small but dedicated community within Wix to advocate this cause.

Wix Key Takeaways

Enhanced the product to create a culture for accessibility and enable users to build accessible websites using Wix.

Wix is a product that enables developers to build websites quickly with its library of customizable components. To ensure that sites built using Wix are accessible, they had to enhance each of their 150 components to support navigation using keyboard and screen readers.

They also had to create a framework that allows the development and testing of accessible components and websites. This effort involved reengineering all the components (including complex components such as galleries and contact forms) to support standard accessibility requirements like correct tab orders, focus indicators, semantic HTML elements with ARIA attributes, and others. They also introduced tools and methods for easier component creation and accessibility validation in the future.

Interview

Ohad Laufer

Former Engineering Manager at Wix

Author, **The Story of Making Wix Accessible**

What excited you or your team the most about the work in the case study?

There were two things that made this project special.

The thing that caught the team's attention initially was the mission. Wix had decided to make all websites created by the platform accessible. Up until that point, accessibility wasn't something we were considering as important or relevant. However, as we were about to start working on the project, we took a few days to get inspired by other companies' approach to accessibility. We watched some videos and talked to people with visual impairments and got a sense of how they experience the web. This was a mind-boggling experience that motivated us throughout the project and made us understand we were making an actual difference. This project gave us a glance into the world of differently-abled people, an issue I took to heart and have advocated ever since.

The other exciting part was the opportunity to touch and guide the entire company on the technical approach to accessibility. We were empowered by Wix engineering leadership to provide a complete solution. As a team focused on a specific feature within the Wix product, we now had a chance to dive deep into the architecture beyond our product scope, understand how things are built, and design a solution that fits all current and future product features. This was the kind of engineering challenge that we were super excited to take on.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

When we started, we didn't expect to have such an impact. We thought we were about to fix a specific pain that our users experienced. It was supposed to include lots of coding and even more testing. Little did we know that the fix would require changing development methodologies and mindsets throughout the company. Furthermore, the end result of a well-structured, accessible website proved valuable for SEO purposes as well. That was another valuable and unexpected win.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

In hindsight, the most important outcome of this project was creating a group of accessibility advocates within the company. It was a by-product of the effort and something that we started to establish late in the project.

If I was to do something differently, this would be it: I would focus only on creating the guidelines and tools and then make an effort to establish a group of advocates throughout the company who would apply these guidelines to their respective products. This would allow for better scaling and continuity of the initiative.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

The most critical decision was to document our process. Early on it was clear this was a project unlike any other we'd recently worked on as a team in terms of the breadth and depth we had to get into and the significant time constraints. It was quite obvious that our usual practices wouldn't fit the project's needs and we had to create new ones.

We created a new execution and tracking process, focused on communication and quick decision-making with all stakeholders and amended it as we progressed. We also documented the entire decision-making process as a single source of truth and for future reference. Luckily enough, this decision later made it easier to share our key learnings across the company and to create this lovely case study about the project.

Do you have any advice for teams that would like to follow in your footsteps?

Getting help and guidance from a subject matter expert was key.

Product teams often mistakenly assume they "get" their users and their needs. In our case, there was no way for us to do that. As people who don't use assistive technologies regularly, we lacked the knowledge and expertise of how to use those as intended, so defining desired behavior and performing verification of our solution was not something we could do on our own. Having an accessibility expert join us for consultation enabled us to make the right decisions and make sure we were building the right thing.

The Understood: How We Improved Web Accessibility

By Catherine Houle & Ilknur Eren

The *Understood.org* front-end team[1] has a special focus on removing barriers for people who learn and think differently. Our core users have ADHD, dyslexia, and other common challenges. But we are committed to creating products that meet the needs of all people. To do this, we combine accessibility and usability in ways that increase ease of use for everyone. This case study outlines the *why* and *how* of our process. We also include basic steps on how to fix common accessibility issues.

Why is Digital Accessibility Important?

Understood.org serves the one in five people who learn and think differently, which translates to approximately 70 million people in the U.S. alone. Learning and thinking differences can include the areas of memory, attention, and reading, as well as language and math, among others.

The Understood front-end team specializes in serving users who learn and think differently. But we are committed to creating products that meet the needs of all people.

Despite how common disabilities are, an article in AdWeek[2] cited the fairly shocking fact that only 2% of all websites meet accessibility standards. Those guidelines are set out by the Web Content Accessibility Guidelines (WCAG), known and accepted worldwide as the minimum requirements to meet digital accessibility.

1 The original version of this case study was published in August 2022: https://smashed.by/understoodcasestudy
2 https://smashed.by/adweek

The WCAG are essential to our work, but they serve as a floor, not a ceiling. These standards should underlie each website and app but also be woven throughout the fabric of every developer's process. Building and maintaining coding configurations that ensure error-free and equal access is the clarion call for all developers and designers.

Ethically, culturally, financially, and legally,[3] expanding accessibility to include neurodivergent people and those with other disabilities is an intelligent and highly relevant business strategy.

DEEP FOCUS ON ACCESSIBILITY

The engineering team at Understood.org is working to combine accessibility and usability[4] in ways that improve ease of use for everyone.

We define "accessibility" as removing barriers for people to gain equal access to information, particularly neurodivergent people, and "usability" as making products like websites and apps easy to use for all people. That includes how simple the product is to use the first time and if the experience was gratifying, one that a user would likely repeat. A physical corollary would be knowing whether you need PUSH or PULL to open a door.

> The reality is that digital accessibility is and always will be an ongoing process.

Developers and designers who are fluent in accessibility are increasingly highly sought after. *The Wall Street Journal*[5] noted job listings with 'accessibility' in the title grew a whopping 78% in 2021.

To be truly accessible, we need to implement solutions for people of all abilities, with both visible and invisible differences. Wheelchair ramps and closed captions are essential. But full access to the amazing power granted by access to goods, services, information, and

3 https://smashed.by/lawsuits
4 https://smashed.by/commitment
5 https://smashed.by/wsj

communication options provided by the Internet also needs more learning and thinking support. This includes ways to help users focus and remember key points.

To do this, we have started putting people at the center of the process. Previously, the focus was on process, data evolution, key metrics, and results. That mindset leaves out a sizable portion of the population which diminishes access for users to all websites and apps across the board – from e-commerce and media outlets (including social and traditional) to government sites, search engines, and educational interests.

What is the Role of Front-End Developers?

As developers, we play an important part in the consistency chain for coding best practices. We believe that due to timing and raising awareness, we are literally part of the process that is developing a foundational language for accessibility and usability that will be utilized by all future generations.

As such, we not only use our knowledge of programming languages to help develop the desired look and feel of our products, we ensure those products are accessible across multiple platforms.

And this is where the rubber meets the road: ensuring flawless operation when incorporating graphics, applications, audio, and video into the mix, ensuring those elements are cohesive and accessible for everyone by consistently testing for speed, usability, and accessibility.

ADDRESSING ACCESSIBILITY

We are on a continuous mission to ensure that sites are perceivable and error-free. Most industries come at the accessibility thing haphazardly. At *Understood.org*, we have found that the cleanest, most efficient way to approach it is to have "accessibility and usability" as prime factors in the initial development process.

It may seem like a basic statement, but as front-end developers, it is crucial that we have an in-depth understanding of how people actually use their devices when they are seeking information or online services.

At *Understood.org*, we reverse the traditional site creation process by listening closely to our users and accessibility consultants rather than designing first and asking questions second. It is not an exaggeration to say that our users' insights guide our work.

The fundamentals of solid development and design practices apply doubly to accessibility and usability:

- **Basic to advanced accessibility training** for all technical teams, including front-end, back-end, and designers.
- **Attending accessibility conferences** each year to keep up with the latest advancements and expand your knowledge base.
- **Conducting surveys and tests** with 'actual' instead of theoretical users. In our case, that would be people who learn and think differently.

Working as a Team

Every industry has its own style and uses a unique flow for development. Because serving people with learning and thinking differences is top-of-mind for us, *Understood.org* begins with user research which includes creating and applying surveys. The information and insights we glean from those surveys inform the designers, who then share content and possibilities with product managers. That information gets relayed to the front-end team to update/create, and then the front-end team creates the site/product for designers who provide feedback and apply their edits.

Why does our process start with user research to inform designers? Deque Systems,[6] a provider of compliance accessibility tools and software, observed that 67% of accessibility issues originate in the design phase of development.

Evolving and maintaining open and honest communications with product managers and design teams translates to less compliance and operational issues down the road. As with any team that works together, yet asynchronously, it is sometimes easier to spot potential concerns from the other side. In our experience:

- Engineers detected accessibility flaws, and the designers found alternative solutions that also aligned with the design vision.

- Designers had top-flight guidance on crafting color contrast, character counts, and effective font styles.

All engineer tickets include accessibility, so each ticket includes an Accessibility Audit. That way, we assign time to deal with whatever issues were revealed.

In our process, we use screen readers to test our pages manually. If there is a video, we refine the closed captions and check individual elements, including headings, buttons, navigation, lists, and color contrast.

Our front-end team always works with product managers to prioritize tickets, and we make it a point to align both teams to make things work. Importantly, engineering teams are realistic when they spec out the appropriate timeline for creation and review.

6 https://www.deque.com/

Product managers then QA all the features and test for accessibility issues. This means that we have two entire teams that review all features for accessibility and errors.

In evolving the workflow, we've learned how important it is to allocate time in sprints to work on accessibility.

Example

For our mobile app:

- We examine everything page by page and log all errors into a central database.
- We then convert that content to Jira tickets, complete with descriptions, screenshots, and story points.
- If we work with a third party, and one of its tools is not accessible, we work with them to make it accessible.

THE DEVELOPER'S POV

- Will non-screen reader users have a comparable experience to that of screen reader users?
- Can users focus on every interactivity in the right order?
- Does the HTML markup make sense?
- Are we conveying helpful semantic and stating information to screen readers? For example, we don't want repeated information that isn't necessary or bad image descriptions.
- Make sure dynamic (error message for form, confirmation of login) changes are transmitted to screen reader users.
- Are our features or components keyboard-accessible?

THE DESIGNER'S POV

- Is there sufficient color contrast?
- Do we have a good font size, clean flow, and layout throughout?
- Does the light/dark mode function well and look good?
- Are all interactions reachable and executable?

Fixing Common Accessibility Errors

There's a wide range of issues that can compromise accessibility, including those the website *webaim.org* calls "mistakes, misconceptions, over-indulgences, intricacies, and generally silly aspects of modern accessibility." We find that webaim.org is an invaluable resource for understanding and then rectifying just about anything that can go haywire. In an article[7] for freecodecamp.org, Ilknur Eren, a front-end developer on our team, included a chart illustrating the most common types of WCAG 2 failures that WebAIM says comprise 96.8% of all accessibility errors:[8]

WCAG FAILURE TYPE	% OF HOME PAGES IN 2022	% OF HOME PAGES IN 2021	% OF HOME PAGES IN 2020	% OF HOME PAGES IN 2019
Low contrast text	83.9%	86.4%	86.3%	85.3%
Missing alternative text for images	55.4%	60.6%	66.0%	68.0%
Empty links	50.1%	51.3%	59.9%	58.1%
Missing form input labels	46.1%	54.4%	53.8%	52.8%
Empty buttons	27.2%	26.9%	28.7%	25.0%
Missing document language	22.3%	28.9%	28.0%	33.1%

This chart lists the most glaring failures, but read on for those that are most common and for understanding quick fixes.

7 https://smashed.by/accessibilityerrors
8 https://smashed.by/detectederrors

MISSING ALTERNATIVE TEXT FOR IMAGES

Understanding the “why” of a coding style is as important as knowing a specific guideline. For images, the context from image to image will vary and should determine the code because algorithms can’t always interpret the meaning of an image.

One classic example of this is creating alternative text[9] for an image in the `alt` attribute of an `<img>` tag.

If you don’t understand why you’re doing it, you may create something that isn’t helpful to the end user but may actually create a brand new barrier.

Say we have an image, and we add the `alt` attribute:

```
<img src="example.png" alt="image"/>
```

While this might not get flagged by automated accessibility tests, a screen reader focusing on this image will say, “image, image.” It doesn’t inform the user and precludes their ability to solve their problem of understanding how to exit a program.

LOW-CONTRAST TEXT

According to recent reports[10] from the WebAim Million, over the last three years, by far the most significant accessibility error is low-contrast text. A surprising 80% of websites have this error, but it is relatively simple to fix. Google has a free tool called Lighthouse that makes it quick and easy to check the color contrast on any web page.

9 https://smashed.by/alt
10 https://smashed.by/webaimmillion

MISSING FORM INPUT LABELS

According to WebAIM,[11] in 2021 half of the delinquent websites were missing their form input labels, which describe what the various fields in the form are for.

One of the most common missing labels is for search forms. If there is no label on a search form, screen readers won't know what the form is.

Here's how you fix that in HTML:

```
<label for="searchLabel" class="sr-only">Search</label>
<input type="text" name="search" id="searchLabel">
<input type="submit" value="Search">
```

And here's CSS coding for the screen reader portion of that HTML snippet:

```
.sr-only{
  position: absolute;
  left: -10000px;
  top: auto;
  width: 1px;
  height: 1px;
  overflow: hidden;
}
```

EMPTY LINKS

As above, almost half of the websites had empty links. This is a simple issue to identify and resolve.

11 https://smashed.by/webaim2021

For example, a Facebook logo that doesn't add a label for a screen reader user will generate an empty link accessibility issue for a non-sighted user.

Adding a label to a link is simple and straightforward:

```
<a href="/facebook-page">
  <i aria-hidden="true"></i>
  <span class="sr-only">Facebook</span>
</a>

.sr-only{
  position: absolute;
  left: -10000px;
  top: auto;
  width: 1px;
  height: 1px;
  overflow: hidden;
}
```

MISSING DOCUMENT LANGUAGE

It's essential to list the language of the page. Screen readers use document language to decide how to pronounce words.

That said, somewhere between 28% and 33% of homepages have been missing a document language for the previous three years.

Add the language to the HTML tag:

```
<html lang="en">
...
</html>
```

EMPTY BUTTONS

It can be frustrating to click on a button and have nothing happen. The user is trying to submit a form or show/hide elements, and the lack of functionality is enough to make them exit the page.

Like empty links, buttons need text for screen readers to read when on focus.

If an image is used inside a button, we should add an `alt` attribute to create a functional image:

```
<button type="submit">
  <img src="/search.svg" alt="Search" />
</button>
```

Conclusions

In this day and age, accessibility should not be an afterthought. Everybody has the right to access the benefits and power of the Internet and apps that make daily life easier and more enjoyable. When reviewing sites and apps for usability, make sure to test your products manually for accessibility. It makes a difference. Allocate specific time to focus on accessibility and maintain open communication channels with product managers and designers.

Allocate specific time to focus on accessibility and maintain open communication channels with product managers and designers.

Continuing to explore technical training and new knowledge goes hand-in-hand with interviewing and surveying people who think differently so that what you produce and put out into the world is of the highest quality and easy for everyone to use.

Understood.org Key Takeaways

Allocate specific time to focus on accessibility and test your products manually for accessibility.

Understood.org, a non-profit that supports people with learning and thinking differences, improved accessibility on their website by fixing common accessibility issues. This included fixing low contrast text, missing form input labels, empty links and buttons, and missing document language for screen reader use.

The team leveraged user research. Engineers on the team detected accessibility flaws. Designers then helped to find alternative solutions

Pinafore: What I Learned About Accessibility in SPAs

By Nolan Lawson

Over the past year or so,[1] I've learned a lot about accessibility, mostly thanks to working on Pinafore,[2] which is a single page app (SPA). In this case study, I'd like to share some of the highlights of what I've learned, in the hope that it can help others who are trying to learn more about accessibility.

One big advantage I've had in this area is the help of Marco Zehe[3], an accessibility expert who works at Mozilla and is blind himself. Marco has patiently coached me on a lot of these topics, and his comments[4] on the Pinafore GitHub repo are a treasure trove of knowledge.

So without further ado, let's dive in!

Misconceptions

One misconception I've observed in the web community is that JavaScript is somehow inherently anti-accessibility. This seems to stem from a time when screen readers did not support JavaScript particularly well, and so, indeed, the more JavaScript you used, the more likely things were to be inaccessible. I've found, though, that most of the accessibility fixes I've made have actually involved writing more JavaScript, not less. So today, this rule is definitely more myth than fact. However, there are a few cases where it holds true.

1 The original version of this case study was published in November 2019: https://smashed.by/spasaccessibility
2 https://pinafore.social/
3 https://marcozehe.wordpress.com//
4 https://smashed.by/pinaforecomment

DIVS AND SPANS VERSUS BUTTONS AND INPUTS

Here's the best piece of accessibility advice for newbies: if something is a button, make it a `<button>`. If something is an input, make it an `<input>`. Don't try to reinvent everything from scratch using `<div>`s and `<span>`s. This may seem obvious to more seasoned web developers, but for those who are new to accessibility, it's worth reviewing why this is the case.

First off, for anyone who doubts that this is a thing, there was a large open-source dependency of Pinafore (and of Mastodon) that had several thousand GitHub stars, tens of thousands of weekly downloads on npm, and was composed almost entirely of `<div>`s and `<span>`s. In other words: when something should have been a `<button>`, it was instead a `<span>` with a click listener. (I've since fixed most of these accessibility issues, but this was the state I found it in.)

This is a real problem! People really do try to build entire interfaces out of `<div>`s and `<span>`s. Rather than chastise, though, let me analyze the problem and offer a solution.

I believe the reason people are tempted to use `<div>`s and `<span>`s is that they have minimal user agent styles; that is, there is less you have to override in CSS. However, resetting the style on a `<button>` is actually pretty easy:

```
button {
   margin: 0;
   padding: 0;
   border: none;
   background: none;
}
```

Ninety-nine percent of the time, I've found that this was all I needed to reset a `<button>` to have essentially the same style as a `<div>` or a `<span>`. For more advanced use cases, you can explore this CSS-Tricks article.[5]

In any case, the whole reason you want to use a real `<button>` over a `<span>` or a `<div>` is that you essentially get accessibility for free:

- For keyboard users who tab around instead of using a mouse, a `<button>` automatically gets the right focus in the right order.
- When focused, you can press the **Space bar** on a `<button>` to press it.
- Screen readers announce the `<button>` as a button.
- etc.

You *could* build all this yourself in JavaScript, but you'll probably mess something up, and you'll also have a bunch of extra code to maintain. It's best just to use the native semantic HTML elements.

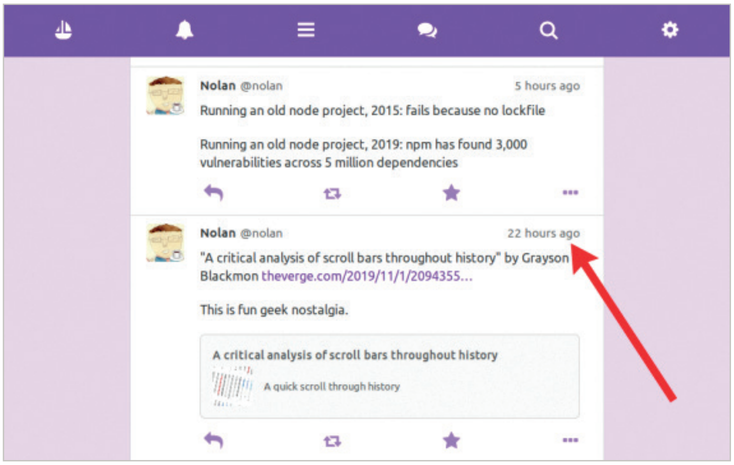
SPAs Must Manually Handle Focus and Scroll Position

There is another case where the "JavaScript is anti-accessibility" mantra has a kernel of truth: SPA navigation. Within SPAs, it's common for JavaScript to handle navigation between pages by modifying the DOM and History API[6] rather than triggering a full page load. This causes several challenges for accessibility:

- You need to manage focus yourself.
- You need to manage the scroll position yourself.

5 https://smashed.by/overridingstyles
6 https://smashed.by/historyapi

For instance, let's say I'm in my timeline, and I want to click this timestamp to see the full thread of a post:



Nolan @nolan 5 hours ago
Running an old node project, 2015: fails because no lockfile

Running an old node project, 2019: npm has found 3,000 vulnerabilities across 5 million dependencies

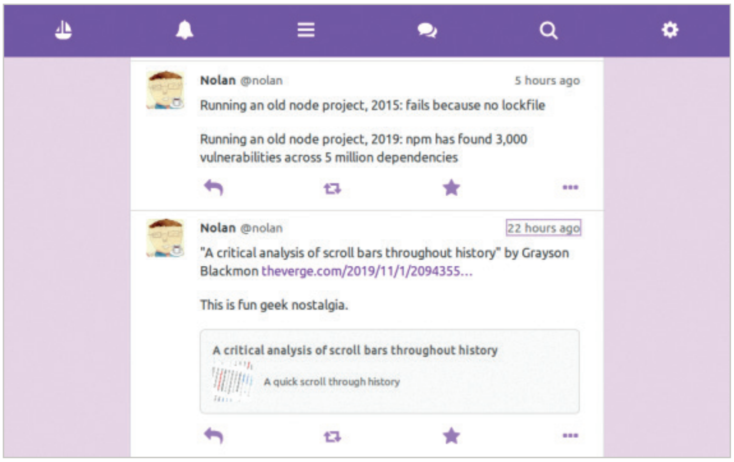
Nolan @nolan 22 hours ago
"A critical analysis of scroll bars throughout history" by Grayson Blackmon theverge.com/2019/11/1/2094355...

This is fun geek nostalgia.

A critical analysis of scroll bars throughout history
A quick scroll through history

Click timestamp to view full thread

When I click the link and then press the **Back** button, focus should return to the element I last clicked (note the purple outline):



Nolan @nolan 5 hours ago
Running an old node project, 2015: fails because no lockfile

Running an old node project, 2019: npm has found 3,000 vulnerabilities across 5 million dependencies

Nolan @nolan 22 hours ago
"A critical analysis of scroll bars throughout history" by Grayson Blackmon theverge.com/2019/11/1/2094355...

This is fun geek nostalgia.

A critical analysis of scroll bars throughout history
A quick scroll through history

Focus should return to the timestamp on click of the Back button

For classic server-rendered pages, most browser engines give you this functionality for free.[7] You don't have to code anything. But in

7 https://smashed.by/browserengines

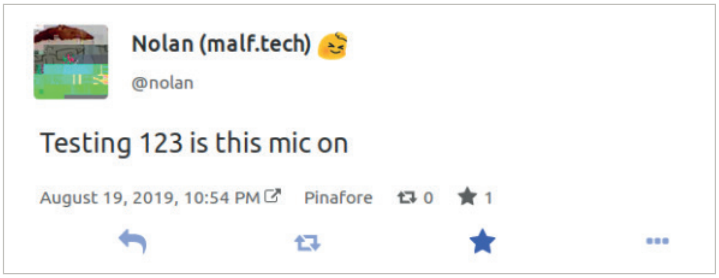
an SPA, since you're overriding the normal navigation behavior, you have to handle the focus yourself.

This also applies to scrolling, especially in a virtual list. In the screenshot above, note that I'm scrolled down to exactly the point in the page where I was when I clicked. Again, this is seamless when you're dealing with server-rendered pages, but for SPAs the responsibility is yours.

Easier Integration Testing

One thing I was surprised to learn is that, by making my app more accessible, I also made it easier to test. Consider the case of toggle buttons.

A toggle button is a button that can have two states: pressed or not pressed.[8] For instance, in the screenshot below, the "boost" and "favorite" buttons (the circular arrow and the star) are toggle buttons, because it's possible to boost or favorite a post, and they start off in unboosted/unfavorited states.



Nolan (malf.tech)

@nolan

Testing 123 is this mic on

August 19, 2019, 10:54 PM Pinafore 0 1

Distinguishing the state of a toggle (star) button using darker colors

Visually, there are plenty of styles you can use to signal the pressed/unpressed state – for instance, I've opted to make the colors darker when pressed. But for the benefit of screen reader users, you'll typically want to use a pattern like the following:

8 https://smashed.by/togglebutton

```
<button type="button" aria-pressed="false">
Unpressed
</button>
```

```
<button type="button" aria-pressed="true">
Pressed
</button>
```

Incidentally, this makes it easier to write integration tests (using TestCafe[9] or Cypress, for instance). Why rely on classes and styles, which might change if you redesign your app, when you can instead rely on the semantic attributes, which are guaranteed to stay the same?
I observed this pattern again and again: the more I improved accessibility, the easier things were to test. For instance:

- When using the feed pattern,[10] I could use `aria-posinset` and `aria-setsize` to confirm that the virtual list had the correct number of items and in the correct order.
- For buttons without text, I could test the `aria-label` rather than the background image or something that might change if the design changed.
- For hidden elements, I could use `aria-hidden` to identify them.

So make accessibility a part of your testing strategy! If something is easy for screen readers to interpret, then it'll probably be easier for your automated tests to interpret too. After all, screen reader users might not be able to see colors, but neither can your headless browser tests!

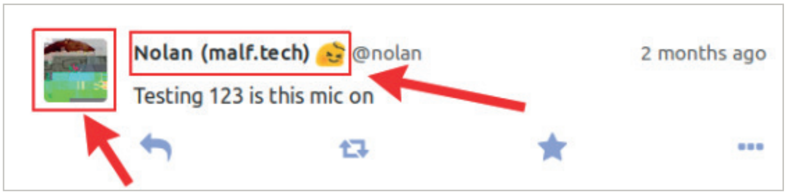
Subtleties with Focus Management

After watching a talk by Ian Forrest ("Beyond Alt-Text: Trends in Online Accessibility,"[11]), and playing around with KaiOS, I realized I could make some small changes to improve keyboard accessibility in my app.

9 https://smashed.by/testcafe
10 https://smashed.by/feedpattern
11 https://smashed.by/beyondalt

As pointed out in the talk, it's not necessarily the case that every mouse-accessible element also needs to be keyboard-accessible. If there are redundant links on the page, then you can skip them in the `tabindex` order, so a keyboard user won't have to press **Tab** so much.

In the case of Pinafore, consider a post. There are two links that lead to the user's profile page: the profile picture, and the user name:



Two links that lead to the same page

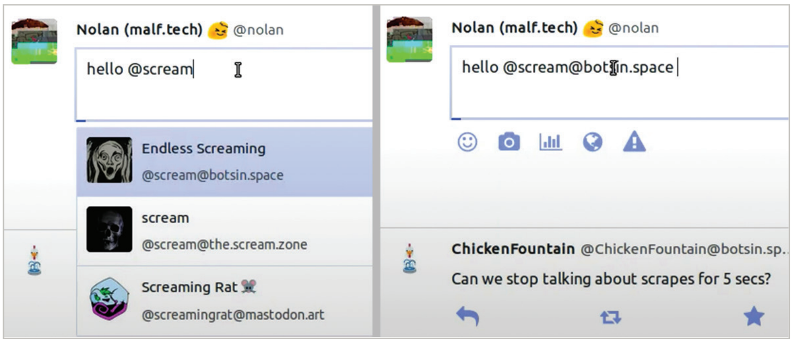
These two links lead to exactly the same page; they are strictly redundant. So I chose to add `tabindex="-1"` to the profile picture, giving keyboard users one less link to have to tab through. Especially on a KaiOS device with a tiny directional pad (d-pad), this is a nice feature! Video example: https://smashed.by/changingfocus

In the video above, note that the profile picture and timestamp are skipped in the tab order because they are redundant – clicking the profile picture does the same thing as clicking the user name, and clicking the timestamp does the same thing as clicking on the entire post. (Users can also disable the "click the entire post" feature, as it may be problematic for those with motor impairments. In that case, the timestamp is re-added to the tab order.)

Interestingly, an element with `tabindex="-1"` can still become focused if you click it and then press the **Back** button. But luckily, tabbing out of that element does the right thing as long as the other tabbable elements are in the proper order.

The Final Boss: Accessible Autocomplete

After implementing several accessible widgets from scratch, including the feed pattern and an image carousel, I found that the single most complicated widget to implement correctly was autocompletion. Video: https://smashed.by/autocompletion


Nolan (malf.tech) @nolan
hello @scream
Endless Screaming
@scream@botsin.space
scream
@scream@the.scream.zone
Screaming Rat
@screamingrat@mastodon.art
Nolan (malf.tech) @nolan
hello @scream@botsin.space
ChickenFountain @ChickenFountain@botsin.sp...
Can we stop talking about scrapes for 5 secs?

The autocomplete widget detects potential usernames while the user types, and allows them to select from a list instead of having to type the entire name.

Originally, I had implemented this widget[12] by following this design[13] by Adina Halter that relies largely on creating an element with `aria-live="assertive"`, which explicitly speaks every change in the widget state (e.g. "the current selected item is number 2 of 3"). This is kind of a heavy-handed solution, though, and it led to several bugs.

After toying around with a few patterns, I eventually settled on a more standard design using `aria-activedescendant`. Roughly, the HTML looks like this:

```
<textarea
  id="the-textarea"
  aria-describedby="the-description"
  aria-owns="the-list"
  aria-expanded="false"
  aria-autocomplete="both"
```

12 https://smashed.by/widget
13 https://smashed.by/autocompletedesign

```
  aria-activedescendant="option-1">
</textarea>
<ul id="the-list" role="listbox">
  <li
    id="option-1"
    role="option"
    aria-label="First option (1 of 2)">
  </li>
  <li
    id="option-2"
    role="option"
    aria-label="Second option (2 of 2)">
  </li>
</ul>
<label for="the-textarea" class="sr-only">
  What's on your mind?
</label>
<span id="the-description" class="sr-only">
  When autocomplete results are available, press up or down
  arrows and enter to select.
</span>
```

In broad strokes, what's happening is:

- The description and label are offscreen, using styles which make it only visible to screen readers.[14] The description explains that you can press up or down on the results and press **Enter** to select.
- `aria-expanded` indicates whether there are autocomplete results or not.
- `aria-activedescendant` indicates which option in the list is selected.
- `aria-labels` on the options allow me to control how it's spoken by a screen reader, and to explicitly include text like "1 of 2" in case the screen reader doesn't speak this information.

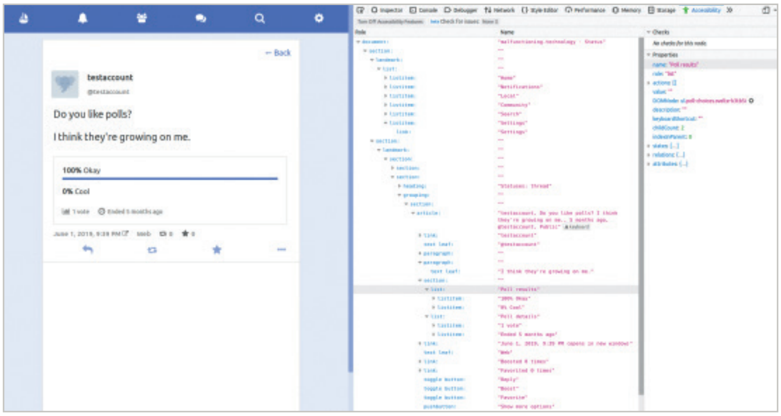
14 https://smashed.by/pinaforebugs

After extensive testing, this was more-or-less the best solution I could come up with. It works perfectly in NVDA on the latest version of Firefox, although sadly it has some minor issues in VoiceOver on Safari and NVDA on Chrome.[15] However, since this is the standards-based solution (and doesn't rely on `aria-live="assertive"` hacks), my hope is that browsers and screen readers will catch up with this implementation.

Update: *I managed to get this widget working in Chrome+NVDA and Safari+VoiceOver. The fixes needed are described in this comment.*[16]

Manual and Automated Accessibility Testing

There are a lot of automated tools that can give you good tips on improving accessibility in your web app. Some of the ones I've used include Lighthouse (which uses Axe[17] under the hood), the Chrome accessibility tools,[18] and the Firefox accessibility tools.[19] (These tools can give you slightly different results, so I like to use several so I can get second opinions!)



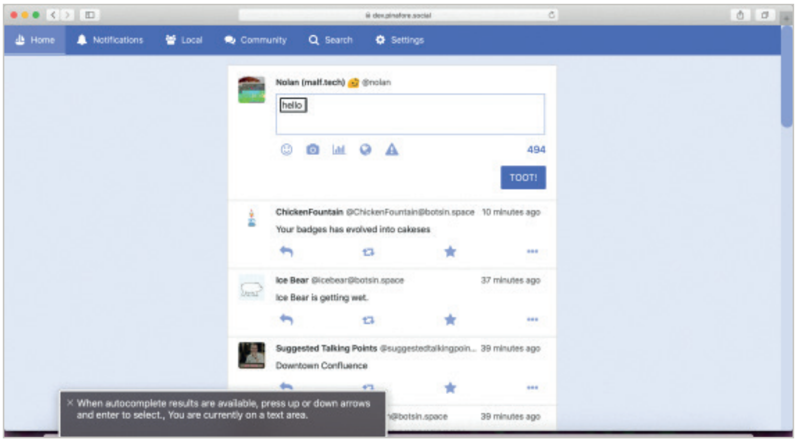
The DevTools Accessibility tab for a post

15 https://smashed.by/voiceoverissues
16 https://smashed.by/lighthouse
17 https://smashed.by/dequeaxe
18 https://smashed.by/chrometools
19 https://smashed.by/firefoxtools

However, I've found, especially for screen reader accessibility, that there is no substitute for testing in an actual browser with an actual screen reader. It gives you the exact experience a screen reader user would have, and it helps build empathy for the kinds of design patterns that work well for voice navigation – and which ones don't. Also, sometimes screen readers have bugs or slightly differing behaviors, and these are things that accessibility auditing tools can't tell you.

If you're just getting started, I would recommend watching Rob Dodson's "A11ycasts" series,[20] especially the tutorials on VoiceOver for macOS[21] and NVDA for Windows.[22] (Note that NVDA is usually paired with Firefox, and VoiceOver is optimized for Safari. So although you can use either one with other browsers, those are the pairings that tend to be most representative of real-world usage.)

Personally I find VoiceOver to be the easiest to use from a developer's point of view, mostly because it has a visual display of the assistive text while it's being spoken.



Nolan (malf.tech) @nolan

hello

ChickenFountain @ChickenFountain@botsin.space 10 minutes ago
Your badges has evolved into cakeses

Ice Bear @icebear@botsin.space 37 minutes ago
Ice Bear is getting wet.

Suggested Talking Points @suggestedtalkingpoin... 39 minutes ago
Downtown Confluence

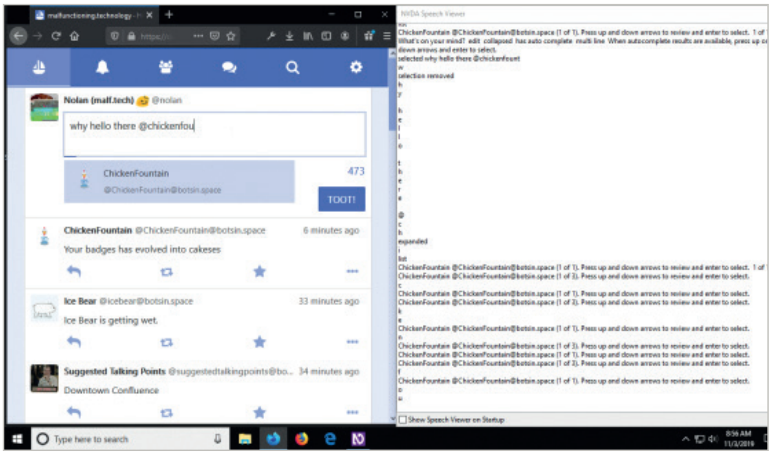
@botsin.space 39 minutes ago

When autocomplete results are available, press up or down arrows and enter to select., You are currently on a text area.

Voice over assistance on macOS

20 https://smashed.by/a11ycasts
21 https://smashed.by/voiceovertutorial
22 https://smashed.by/nvdatutorial

NVDA can also be configured to do this, but you have to go into the settings and enable the Speech Viewer option. I would definitely recommend turning this on if you're using NVDA for development!



Nolan (malf.tech) @nolan

why hello there @chickenfou

ChickenFountain @ChickenFountain@botsin.space

473

TOOT!

ChickenFountain @ChickenFountain@botsin.space 6 minutes ago

Your badges has evolved into cakeses

Ice Bear @icebear@botsin.space 33 minutes ago

Ice Bear is getting wet.

Suggested Talking Points @suggestedtalkingpoints@bo... 34 minutes ago

Downtown Confluence

NVDA Speech Viewer

```
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.  1 of
What's on your mind?  edit  collapsed  has auto complete  multi line  When autocomplete results are available, press up or
down arrows and enter to select.
selected why hello there @chickenfount
w
selection removed
h
y

h
e
l
l
o

t
h
e
r
e

@
c
h
expanded
i
list
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.  1 of
ChickenFountain @ChickenFountain@botsin.space (1 of 3). Press up and down arrows to review and enter to select.
c
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.
ChickenFountain @ChickenFountain@botsin.space (1 of 3). Press up and down arrows to review and enter to select.
k
e
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.
n
ChickenFountain @ChickenFountain@botsin.space (1 of 3). Press up and down arrows to review and enter to select.
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.
ChickenFountain @ChickenFountain@botsin.space (1 of 3). Press up and down arrows to review and enter to select.
f
ChickenFountain @ChickenFountain@botsin.space (1 of 1). Press up and down arrows to review and enter to select.
o
u
```

☐ Show Speech Viewer on Startup

Speech viewer in NVDA

Similar to testing screen readers, it's also a good idea to try tabbing around your app to see how comfortable it is with a keyboard. Does the focus change unexpectedly? Do you have to do a lot of unnecessary tabbing to get where you want? Are there any handy keyboard shortcuts you'd like to add?

For a lot of things in accessibility, there are no hard-and-fast rules. Like design or usability in general, sometimes you just have to experience what your users are experiencing and see where you can optimize.

Conclusion

Accessibility can be challenging, but ultimately it's worth the effort. Working on accessibility has improved the overall usability of my app in a number of ways, leading to unforeseen benefits such as KaiOS arrow key navigation[23] and better integration tests.

The greatest satisfaction, though, comes from users who are happy with the work I've done. I was beyond pleased when Marco Zehe had this to say:

> *Pinafore is for now by far the most accessible way to use Mastodon. I use it on desktop as well as iOS, both iPhone & iPad, too. So thank you again for getting accessibility in right from the start and making sure the new features you add are also accessible.*
> *– Marco Zehe, October 21, 2019*[24]

Thank you, Marco, and thanks for all your help! Hopefully this case study will serve as a way to pay your accessibility advice forward.

Pinafore Key Takeaways

JavaScript is not necessarily anti-accessibility, and you can build accessible SPAS using JavaScript.

Nolan Lawson built Pinafore, a web client for the Mastodon open source social network. Pinafore is a single page application and a progressive web app (PWA). Nolan was worried about making the app accessible since it used JavaScript extensively. However, he was able to achieve the desired results with workarounds for specific scenarios such as using native semantic HTML elements with ARIA attributes.

Nolan also recommends that we try to test for accessibility using actual browsers and screen readers to discover the issues that users might face.

23 https://smashed.by/arrowkey
24 https://smashed.by/marcozehe

LinkedIn's Approach to Automated Accessibility Testing

By Oliver Tse, Andrew Lee, Melanie Sumner, Renato Iwashima

Accessibility (A11Y) engineering at LinkedIn[1] aims to streamline accessible product development and maintenance. We think about how tooling can help achieve accessibility success and increase the efficiency of our fellow engineers across the organization. We design tools and the infrastructure necessary for engineers to build products that are accessible, elegant, delightful, and more easily testable.

Our overall aim is to bring together the amazing talent across all product teams at LinkedIn and to facilitate inclusive design through innovative tooling, collaborative projects, and consultations. Just as LinkedIn ensures that everyone is connected to economic opportunity, A11Y engineering ensures that every product and platform is inclusively empowered to do just that.

With this vision in mind, we have strategically embraced accessibility test automation to accelerate our detection of common issues in new features and reduce regressions in existing features. At its core, accessibility test automation involves running a suite of accessibility rules on essential user flows and user interfaces of our applications.

The main strength of automated accessibility testing is its ability to find "low-hanging fruit" caused by oversights at the code level. This frees up engineering time and covers common defects that are often overlooked.

1 The original version of this case study was published in May 2020: https://engineering.linkedin.com/blog/2020/automated-accessibility-testing (need short link)

The other key benefit is the ability to guard against regressions. Good accessibility test coverage provides a clear signal for overall A11Y health. A sudden introduction of new violations will act as a smoke signal, serving as a leading indicator for the existence of other issues. In an environment characterized by multiple releases per day, where it is not feasible to accompany each new release with manual testing, this becomes an indispensable tool.

Running accessibility test automation rules during continuous integration prevents common accessibility issues from ever reaching LinkedIn members, allowing us to address problems before we ship. While we see automated testing as an essential part of our arsenal to scale accessibility, we fully acknowledge that, depending on who you ask or reference, accessibility test automation will only identify between 20% and 30% of accessibility issues out there. ("Automated accessibility checking [...] can only auto-check about 30% of #a11y issues"[2] and "Manual Accessibility Testing: Why & How"[3].)

At LinkedIn, we integrate various open-source and licensed automation frameworks into our continuous integration pipeline, where only if the commit passes accessibility checks will it be merged into the main branch for our web, iOS, and Android applications, including:

1. Deque's axe-core for the web[4]
2. Google's Toolbox for Accessibility for iOS (GTXiLib)[5]
3. Google's Accessibility Test Framework for Android (ATF)[6]

LinkedIn web applications are built using the Ember JavaScript framework. It features an extensive ecosystem of plugins and extensions commonly referred to as Ember add-ons. Testing is a core tenet of the framework and is built in as a first-class citizen.

2 https://smashed.by/autocheck
3 https://smashed.by/manualtesting
4 https://smashed.by/dequecore
5 https://smashed.by/gtxilib
6 https://smashed.by/atf

Web Apps

LinkedIn uses Deque's axe-core accessibility testing framework for our web apps. Axe is a static analysis engine for websites and other HTML-based user interfaces. It is integrated into our Ember testing infrastructure by the Ember A11Y testing add-on.[7]

There are three types[8] of Ember tests:

1. **Unit tests** verify individual pieces of code. They are insufficient to assess A11Y but are very quick to execute.

2. **Integration tests** verify user interface components. They are examined in isolation, somewhat quick to execute, but lack the fidelity to accurately assess A11Y.

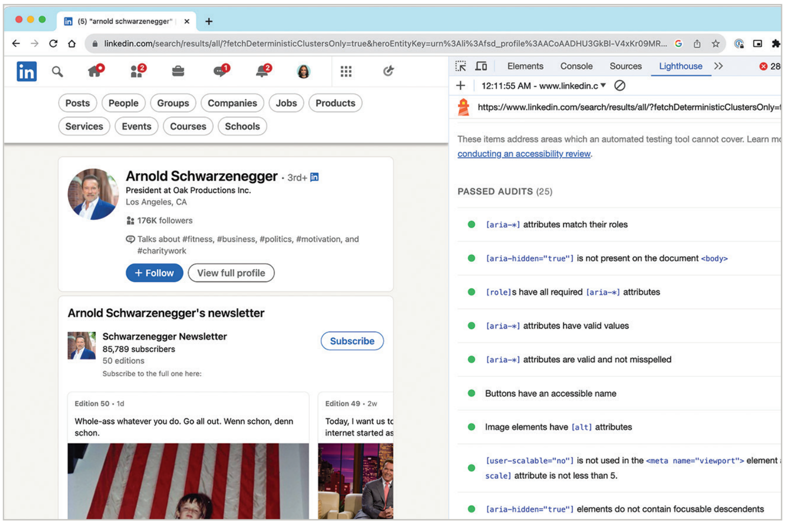
3. **Acceptance tests** are the ideal point of integration as they most closely resemble what a user would experience. However, they are much more costly in terms of runtime and resourcing.

Our continuous integration and deployment (CI/CD) process is 3×3, consisting of at least three daily deployments and three stages of validation: pre-commit, pre-merge, and post-merge.

- **Pre-commit** occurs on push-to-remote and includes static linting checks and dependency validation.
- **Pre-merge** occurs after a successful push and includes build verification and test suite execution.
- If successful, **post-merge** checks queue up associated changes for deployment.

In terms of accessibility, any resulting violations prevent changes from moving to the next stage. Specifically, invalid A11Y ember-tem-

7 https://smashed.by/embertesting
8 https://smashed.by/testtypes


(5) "arnold schwarzenegger"
linkedin.com/search/results/all/?fetchDeterministicClustersOnly=true&heroEntityKey=urn%3Ali%3Afsd_profile%3AACoAADHU3GkBl-V4xKr09MR...
Posts
People
Groups
Companies
Jobs
Products
Services
Events
Courses
Schools
Elements
Console
Sources
Lighthouse
28
12:11:55 AM - www.linkedin.c
https://www.linkedin.com/search/results/all/?fetchDeterministicClustersOnly=

Arnold Schwarzenegger · 3rd+
President at Oak Productions Inc.
Los Angeles, CA
176K followers
Talks about #fitness, #business, #politics, #motivation, and #charitywork
+ Follow | View full profile

Arnold Schwarzenegger's newsletter
Schwarzenegger Newsletter
85,789 subscribers
50 editions
Subscribe to the full one here:
Subscribe

Edition 50 · 1d
Whole-ass whatever you do. Go all out. Wenn schon, denn schon.

Edition 49 · 2w
Today, I want us tc
internet started as

These items address areas which an automated testing tool cannot cover. Learn m
conducting an accessibility review.

PASSED AUDITS (25)

- [aria-*] attributes match their roles
- [aria-hidden="true"] is not present on the document <body>
- [role]s have all required [aria-*] attributes
- [aria-*] attributes have valid values
- [aria-*] attributes are valid and not misspelled
- Buttons have an accessible name
- Image elements have [alt] attributes
- [user-scalable="no"] is not used in the <meta name="viewport"> element
  scale] attribute is not less than 5.
- [aria-hidden="true"] elements do not contain focusable descendents

LinkedIn's Lighthouse accessibility report for their main timeline experience.

plate-lint[9] checks block commits, while failed A11Y test assertions block merging and are auto-reverted.

So, we have a multifaceted approach to accessibility engineering: we use linting for static analysis; automated testing for dynamic analysis; and manual testing for the things we can't automate yet. Each of these approaches has their strengths: linting gives developers feedback right in their integrated development environment (IDE); automated testing can more robustly check the rendered code to make sure it provides what screen readers expect; and manual audits ensure usability. As such, we consider them to all be important parts of the whole.

As engineers run local tests during their day-to-day work, A11Y regressions resulting from UI changes are automatically identified by existing A11Y assertions. As such, it is imperative that all new A11Y assertions are introduced at a clean state, meaning that all identified violations are addressed before submission. Since the entire test suite runs after pushing a code commit, any A11Y failures block the associated commit from being merged.

9 https://smashed.by/templatelint

Impact on Performance

Commit-to-publish (C2P) time is an important metric signaling the overall health of our build and testing infrastructure. Unfortunately, the initial implementation of automated accessibility tests turned out to negatively impact and regress C2P. Tests with added A11Y assertions were found to more than double in execution time. Considering that there are over 8,000 acceptance tests, the additional load would equate to substantial degradation of the CI/CD pipeline.

Mitigation Strategies

Mitigating negative performance impacts required cross-collaboration with several partner teams. Based on intensive investigatory work, clear guidance and integration, best practices were formulated:

1. Indiscriminate a11y assertions cause redundancy, as multiple tests end up checking the same view.
2. To avoid such redundancy, assertions should be deliberate and scoped into specific segments of the screen.
3. Assertions should cover high-traffic, high-impact transaction paths.
4. One hundred percent accessibility test coverage is not the goal. Tests are a signal of overall accessibility health, but not the only signal. They are but a single tool in our overall arsenal.

Further investigation led to improvements from optimized configuration settings and the omission of non-performant, low-impact rules. But by far the biggest improvement came from the creation of a dedicated accessibility distributed test job. This job automatically pipes all A11Y assertions into a parallel process so that the main test execution isn't affected, thus preserving C2P time.

iOS and Android Apps

We use the same testing approach for iOS and Android that we use for the web. However, we run fewer rules and we do not run them using the same library. This is because mobile platforms have their own APIs and different ways to interact with assistive technologies. We do plan to increase the number of accessibility checks as the accessibility capability of both platforms evolves.

IOS

We have six rules. We check for whether:

1. **Label is present**: Ensure that all accessibility elements have a label.

2. **Trait is not in the label**: Ensure that elements don't redundantly describe accessibility traits such as "Button" in the label, since these roles are announced by the screen reader automatically.

3. **Label is not redundant**: Ensure that accessibility labels are unique so that they are distinguishable when using a screen reader.

4. **Traits don't conflict**: Ensure that incompatible traits such as "Button" and "Link" are not used at the same time.

5. **Touch target size**: Ensure that all interactive elements have a touch target size of at least 44pt.

6. **Contrast is sufficient**: Ensure that text contrast is at least 4.5 to 1.

For more information, check Google's GTXiLib.[10]

ANDROID

We have ten rules. We check for:

1. **Unsupported item type**: Ensure that specified accessibility class name (role) is supported by TalkBack.

2. **Clickable span**: Ensure that ClickableSpan is not being used in a TextView, since it is inaccessible because individual spans cannot be selected independently in a single TextView.

3. **Traversal order**: Ensure that the traversal order specified by the developer doesn't have any problems such as loops or constraints in the traversal.

4. **Contrast check**: Ensure that text contrast is at least 4.5 to 1.

5. **Label present**: Ensure that all accessibility elements have a label.

6. **Duplicate clickable bounds**: Ensure that clickable/touchable bounds are not overlapping each other.

7. **Duplicate speakable text**: Ensure that accessibility labels are unique so that they are distinguishable when using a screen reader.

10 https://smashed.by/gtxilib

8. **Editable content description**: Ensure that an editable TextView is not labeled by a `contentDescription`.

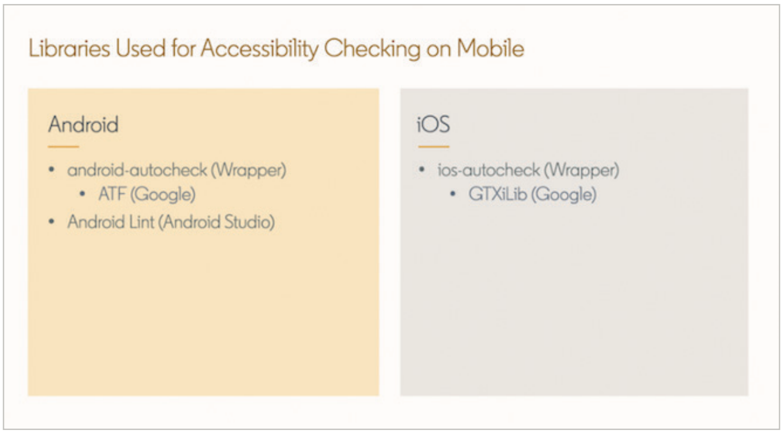
9. **Touch target size**: Ensure that all interactive elements have the touch target size of at least 48dp.

10. **Link purpose is unclear**: Ensure that the link purpose is neither unclear nor insufficiently descriptive enough.

For more information, check Google's Accessibility Testing Framework[11] and Google's Accessibility Scanner.[12]

Libraries for Accessibility Checking on iOS and Android

We use internal wrapper libraries for both Android and iOS called android-autocheck and ios-autocheck respectively.



Libraries Used for Accessibility Checking on Mobile

Android
- android-autocheck (Wrapper)
  - ATF (Google)
- Android Lint (Android Studio)

iOS
- ios-autocheck (Wrapper)
  - GTXiLib (Google)

Android/iOS libraries used for A11Y checking

11 https://smashed.by/atf
12 https://smashed.by/accessibilityscanner

These wrapper libraries use the Accessibility Testing Framework for Android and GTXiLib for iOS.

We use wrapper libraries because we can:

- add custom rules,
- override existing rules,
- define sets of rules,
- define suppression list.

Android also provides a linting tool that does some basic accessibility checks. None of the linting checks are considered errors, due to false positives inherent to static analysis checks, but they are still very helpful for engineers to detect possible issues.

Conclusion

To be effective, A11Y automated testing must be executed thoughtfully, deliberately, and in conjunction with multiple types of testing (including human manual testing, engineers doing their due diligence, linting, and so on), with each deployed at the right time in the continuous integration development process.

Automated A11Y testing is forward-looking. When executed early and often during the development process, it is a highly accurate predictor of how accessible our products will be.

Automated A11Y testing is forward-looking. When executed early and often during the development process, it is a highly accurate predictor of how accessible our products will be. While it's true that A11Y automated testing is unable to detect issues found only by manual testing, its key benefit is that it is highly efficient. Automated testing can identify issues faster than the amount of time it takes a human manual tester to file a single bug. By exploiting its efficiency, we use it as a litmus test to gain insight into the accessibility of our products.

Overall, A11Y automated testing is an essential tool that empowers our engineers to build accessible products for our members in service of our vision to create economic opportunity for everyone, including the more than one billion people around the world with disabilities.

LinkedIn Key Takeaways

Automated accessibility testing helps to find "low-hanging fruit" early when it comes to accessibility issues and is an accurate predictor of overall accessibility health.

LinkedIn has built automated accessibility testing as part of its continuous integration and deployment (CI/CD) process for its web apps and iOS and Android apps. Deque's axe-core accessibility testing framework is integrated with the Ember framework for web apps and is used for unit testing, integration testing, and acceptance testing. This is combined with linting and manual testing to ensure that accessibility is tested at different life cycle stages. Guidelines are also in place to prevent developers from adding redundant or low-impact test assertions that degrade the CI/CD pipeline.

Additionally, LinkedIn uses Google's Toolbox for Accessibility for iOS (GTXiLib) for testing its iOS apps, and Google's Accessibility Test Framework for Android (ATF) for Android apps.

Interview

Oliver Tse & Andrew Lee

Co-Authors – along with Rachel Peterson – of **LinkedIn's Approach to Automated Accessibility Testing**

What excited you or your team the most about the work in the case study?

Oliver: This definitely would be integrating accessibility automated testing into our continuous integration development pipeline.

We really wanted to enhance our code quality and ensure that our products were accessible before they reached our developers. We also wanted to make our developer experience seamless so that it was not just something additional that they had to do – not yet another process!

Integrating automated accessibility testing into the continuous integration pipeline helps to increase the efficiency of the development process. Our developers can quickly identify and address accessibility issues before they become more complex and more costly to fix later on.

This also means that accessibility can be considered throughout the development life cycle and with contributions from accessibility and inclusive designers, thus improving collaboration between our R&D teams.

Overall, integrating accessibility testing into the continuous integration process leads to a better user experience for everyone.

Andrew: This work was Accessibility Engineering's first foray into an organization-wide initiative that allowed us to meaningfully scale our impact. Prior efforts were primarily rooted in evangelism and trusting our partners in developing apps with accessibility in mind. Implementing automated accessibility testing meant that there was now a tight bond between A11Y and the development life cycle, where it is integral to the process.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

Oliver: We knew that automated testing can find at best 30% of A11Y issues. It still is amazing how many a11y nonconformances are not caught by automated testing and how much manual testing is required, and we acknowledge that automated a11y testing has limited scope; testing can only identify certain types of issues and may not be able to detect all accessibility problems.

This reminds us that automated A11Y testing is not the only method used to test for accessibility. Automated A11Y testing tools cannot yet fully replicate the user experience, and may not take into account factors such as user behavior or the impact of the issue on the user. Manual testing by humans, A11Y expert review, and user testing are also important components of a comprehensive accessibility testing strategy.

Andrew: The limitations and effectiveness of automated accessibility testing tools were generally understood, so the overall impact was not too surprising for me. However, I believe the biggest win was bringing accessibility to the forefront among front-end engineers who weren't yet directly exposed to A11Y. It suddenly became a part of the conversation, where teams were outlining their own resources and strategies for actively addressing a11y issues that were the result of the automation tools.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

Andrew: At the time, integration of web automated accessibility testing was limited owing to having to mitigate performance issues in running tests. This ultimately resulted in inadequate coverage. The process for enabling A11Y checks also didn't scale well to newly-authored tests. Most of the performance issues have since been mitigated as our infrastructure and tooling matured. This allowed us to revisit our approach and ease many of the integration pain points by leveraging some of the core APIs in our web framework. Given a do-over, we would've explored augmenting the core framework much sooner in the process, as it turned out to be the main source for the bulk of the improvements.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

Andrew: The most critical decision was adopting open-source libraries across the majority of platforms instead of building in-house solutions. This allowed coalescing across a unified set of resources and the ability for all engineers to triage issues and contribute back to the respective projects. The decision was made after carefully considering the available options in the open source landscape and weighing the cost of adoption versus maintaining custom-built solutions. Ultimately, the decision proved to be the right one, as the solutions, especially on the web platform, have been immensely improved by the contributions of others.

Do you have any advice for teams that would like to follow in your footsteps?

Oliver: Clearly set expectations on what automated A11Y testing can and cannot find. Automated A11Y testing should be one of the methods used to test for accessibility, along with manual testing by humans, expert review, and user testing.

Automated A11Y testing is a valuable tool in identifying and addressing accessibility issues. We use it as part of a comprehensive approach to accessibility testing, rather than relying on it exclusively.

Has the site changed significantly since the case study was published?

Oliver: No, the site has not significantly changed since the case study was published.

Building Dark Mode on Stack Overflow

By Aaron Shekey

On March 30, 2020,[1] we enabled folks to opt into a beta dark mode on Stack Overflow. Let's talk about the work that went into it. I'm Aaron Shekey, Stack Overflow's principal product designer on design systems. I help design all the interface components that get assembled into new features.

First, a bit of irony. I don't actually prefer dark user interfaces.

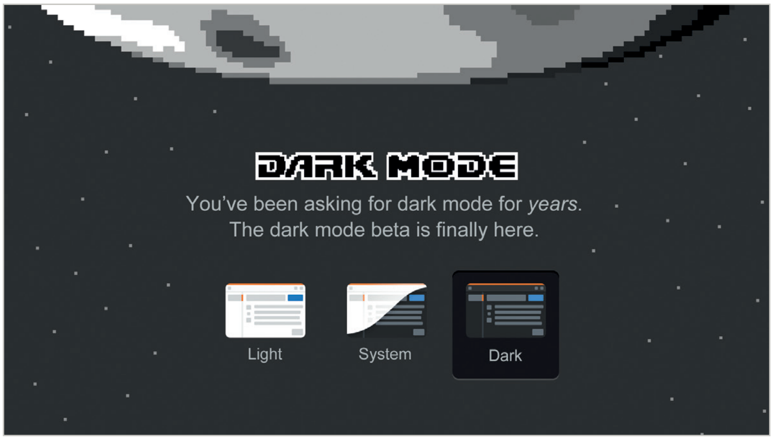
I often find the usable contrast to be way too low. It's hard to use the full spectrum of colors to express your interface. It's even harder to introduce depth with shadows and other visual cues. Light text on dark backgrounds is fatiguing to my eyes. Things that are hard to manage on light screens like simultaneous contrast are even harder to manage against dark backgrounds.

But here I am, the guy who finally shipped dark mode on Stack Overflow.

The work I'm about to talk about was never about dark mode specifically, even though countless users asked for it. By solving everything along the way to dark mode, Stack Overflow would modernize its front-end codebase, enable accessibility-conscious theming, and push for adoption of our design system[2].

We could give our users dark mode and offer future accessibility modes for free? Let's do it!

1 The original version of this case study was published in March 2020: https://smashed.by/stackoverflowdarkmode
2 https://stackoverflow.design/

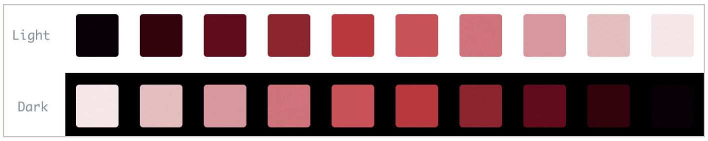

DARK MODE
You've been asking for dark mode for *years*.
The dark mode beta is finally here.
Light
System
Dark

Dark mode opt-in banner

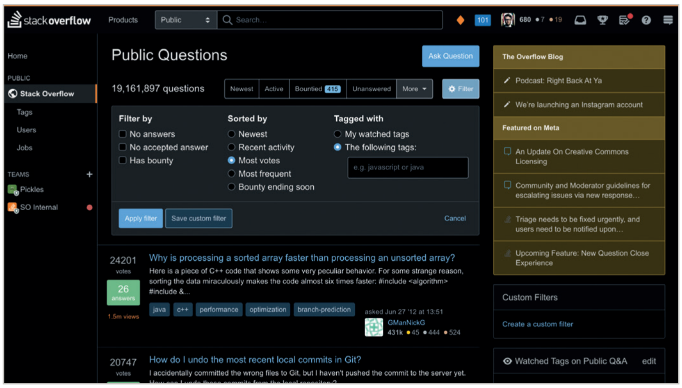
Color Exploration

When building our product's original color scales, we – perhaps naively – took a single color value and modified it using Less.js color transformations. For example, we'd define a Less variable, `@red`, and darken it by 10% a few times using `darken(@red, 10%)`. Then we'd tint to lighten a few times at the other end of the spectrum: `tint(@red, 10%)`. This would lead us to a color scale represented by `@red-050` through `@red-900` with 10% steps in between.

In my first explorations of what Stack Overflow would look like in dark mode, I wanted to simply test swapping the white background for black, and reversing the color scales. With this approach, `@red-050` became `@red-900` with the values in the middle staying pretty much the same.



This approach made everything have unusable contrast, and fell into the traps of what I dislike about dark modes in general. Pay close attention to the darkest value of red against the black background. It’s nearly indistinguishable. More on that later.



Podcast: Right Back At Ya

We’re launching an Instagram account

An Update On Creative Commons Licensing

Community and Moderator guidelines for escalating issues via new response…

Triage needs to be fixed urgently, and users need to be notified upon…

Upcoming Feature: New Question Close Experience

24201 votes · 26 answers · 1.5m views

Why is processing a sorted array faster than processing an unsorted array?

Here is a piece of C++ code that shows some very peculiar behavior. For some strange reason, sorting the data miraculously makes the code almost six times faster: #include <algorithm> #include &…

java c++ performance optimization branch-prediction

asked Jun 27 '12 at 13:51 GManNickG 431k •45 •444 •524

20747 votes

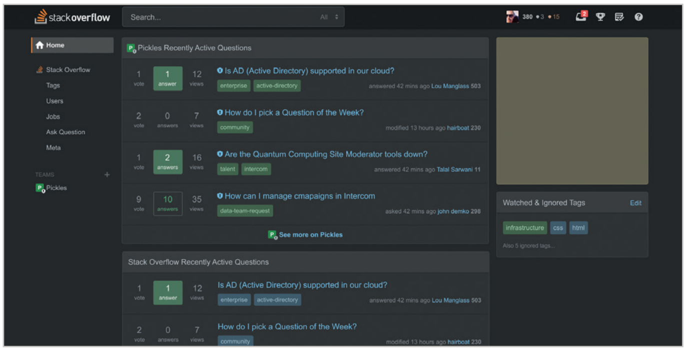
How do I undo the most recent local commits in Git?

I accidentally committed the wrong files to Git, but I haven't pushed the commit to the server yet.

We’d have to do better than this.

STARTING WITH THE MOCKUP

After just diving in technically proved to be a false start, I instead chose colors by hand in my design tool of choice, Figma.[3] I could design what Stack Overflow ought to look like without concern for how the original color values would map. Reducing the overall contrast was key to preserving depth in our interface, allowing elements to cast shadows, and displaying the full spectrum of colors.



1 vote · 1 answer · 12 views — Is AD (Active Directory) supported in our cloud? — enterprise active-directory — answered 42 mins ago Lou Manglass 503

2 vote · 0 answers · 7 views — How do I pick a Question of the Week? — community — modified 13 hours ago hairboat 230

1 vote · 2 answers · 16 views — Are the Quantum Computing Site Moderator tools down? — talent intercom — answered 42 mins ago Talal Sarwani 11

9 vote · 10 answers · 35 views — How can I manage cmapaigns in Intercom — data-team-request — asked 42 mins ago john demko 298

1 vote · 1 answer · 12 views — Is AD (Active Directory) supported in our cloud? — enterprise active-directory — answered 42 mins ago Lou Manglass 503

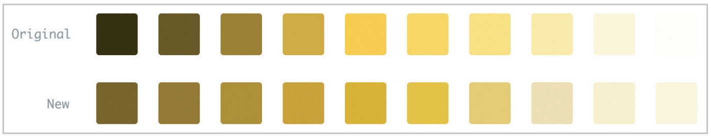
2 vote · 0 answers · 7 views — How do I pick a Question of the Week? — community — modified 13 hours ago hairboat 230

Starting with a mockup allowed us to define an aesthetic goal first, regardless of technical requirements.

3 https://figma.com/

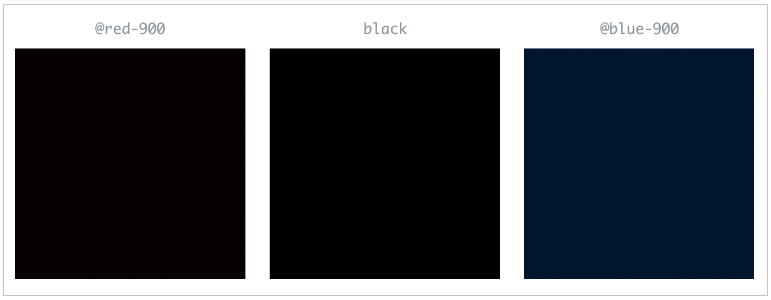
CHOOSING A BETTER ALGORITHM

After picking a lighter background for dark mode, I could then explore the color scale in a deeper manner. First, I needed to solve some of the color issues the design system inherited in light mode. At the light end of the spectrum our reds and yellows weren't as usable as I would have liked. With some colors, the lightest value was too close to white, while for others the lightest value was much too dark.



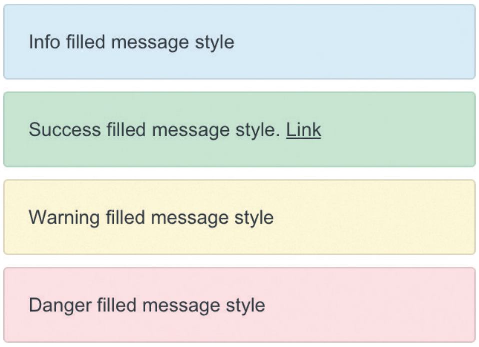
The original lightest yellow was indistinguishable from white, and the darkest yellow indistinguishable from black.

We had trouble at the darker end of the spectrum for each color. When applying `@red-900` and `@blue-900` to a background, these colors were indistinguishable from black and each other. We needed an algorithm that would provide colors that still read as their primary hue at the lightest and darkest values, allowing us to build components from these color values.



The darkest values of our colors were indistinguishable from each other and black.

When creating our notices component, we couldn't use colors from our design system. Instead, we had to eyeball custom colors.


Info filled message style

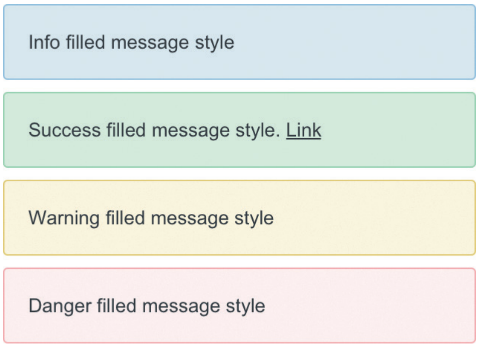
Success filled message style. Link

Warning filled message style

Danger filled message style

These colors are beautiful, but weren't based on values within our color scale.

I used Lyft's amazing ColorBox[4] to help normalize our colors. Instead of a naive linear scale at 10% increments, I used bezier curves – a vast improvement at the more extreme ends of the scale.


Info filled message style

Success filled message style. Link

Warning filled message style

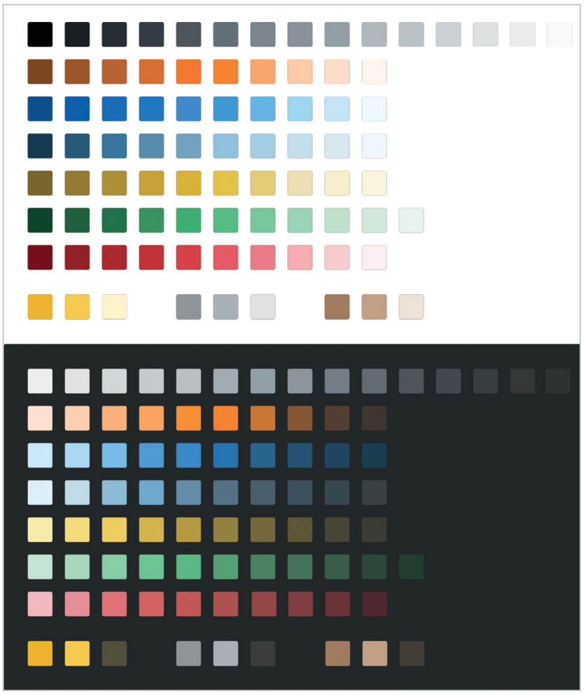
Danger filled message style

After normalizing our color values at the light end of the spectrum, I could now build our notices component using values within our color scale.

DARK VERSIONS

Once I polished our light versions, I could now explore these colors against the dark background. I would ultimately end up hand-tuning the algorithm's output to preserve long-used brand colors at certain values. This would allow me to drop the new colors into production without too jarring a shift.

4 https://www.colorbox.io/



The full normalized color gamut.

Adding the Colors to Stacks

If I had any hope of shipping dark mode to Stack Overflow, I'd first need to solve dark mode using Stacks,[5] our design system, as a sandbox.

VARIABLES

I needed to convert static, Less-compiled hex values to runtime custom CSS properties. This meant storing our color values as `var(--red-500)` instead of a static `@red-500`. This was an interesting problem in our design system and the site in general. We routinely take a single color value like `@red-500` and lighten or darken for hover and focus states, and things like backgrounds and border colors.

5 https://stackoverflow.design/

Each of our many buttons and their individual states were based on a set of transformations of a single compiled color value. It reminded me of this scene[6] in *The Big Short*: “We can transform an original 10 million dollar investment into billions of dollars,” and of course the whole thing explodes.

The problem with native CSS variables is you can’t apply any type of Less transformation to them. `darken(var(--red-500), 5%)` breaks the compiler since CSS variables are only evaluated at runtime.

This meant I’d need to refactor how all of our buttons were created. These were the existing button styles using the Less syntax:

```
.s-btn {
    color: @white;

    background-color: @blue-600;
    border: 1px solid darken(@blue-600, 5%);

    &:hover {
        background-color: darken(@blue-600, 5%);
        border-color: darken(@blue-600, 10%);
    }
}
```

I needed to translate these to their more explicit color values as defined by our color system. Instead, it ended up looking like this:

```
.s-btn {
    color: var(--white);
    background-color: var(--blue-600);
    border: 1px solid var(--blue-700);


    &:hover {
        background-color: var(--blue-700);
        border-color: var(--blue-800);
    }
}
```

6 https://smashed.by/bigshort

I needed to do this across all of our Stacks components, not just the buttons. These same concepts applied across notices, pop-overs, modals, buttons, and links to name a few.

BROWSER COMPATIBILITY

Oh, but wait a second. CSS variables aren't supported by Internet Explorer 11 (IE11), a browser we very much supported at the time of this exploration. Ultimately, we made the decision to drop support for IE11, ripping out all the CSS hacks we'd added over the years to get it to behave, and then shipping deprecation notices to users on IE11 urging them to install a new browser. This was not a decision we took lightly, and this prerequisite alone took weeks of refactoring.

CONDITIONAL CLASSES

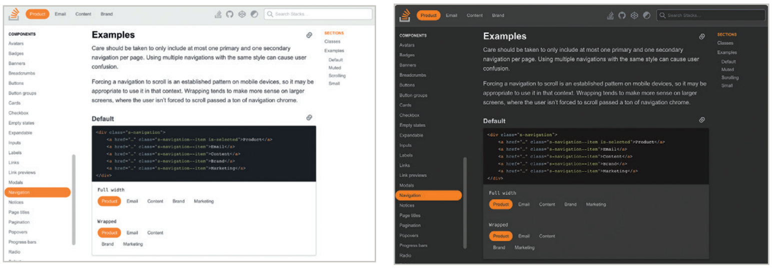
With IE11 no longer holding us back, I was able to work with our colors within Stacks. I chose to enable adding the class `.theme-system` to the `body` element. In doing so, we'd swap our light colors for their dark equivalents behind the dark mode media query. Additionally, we could skip that media query entirely and just force the dark colors by adding `.theme-dark` instead to the `body`. This would allow users to see dark mode regardless of their system's settings. My approach ended up looking like this:

```
body {
    --red-600: #c02d2e;
}


body.theme-system {
    @media (prefers-color-scheme: dark) {
        --red-600: #d25d5d;
    }
}
body.theme-dark {
    --red-600: #d25d5d;
}
```

To offer complete flexibility, Stacks provides atomic color classes that are only applied when dark mode is enabled. You can read about Stacks CSS design choices at length at my personal portfolio.[7] By adding `.d:bg-green-100` to an element, our engineers and designers can say "In dark mode, apply a background of green 100." Additional conditional classes allow us to drop borders, swap backgrounds, or change text colors in dark mode. Steve Schoger's got a really great tweet[8] demonstrating the customizations that are sometimes required for dark modes. I've taken lots of inspiration from Tailwind CSS.[9]

DOCUMENTING IT

Once Stacks was in a place to ship its own dark mode, we opted to add a button on the top of the site to quickly toggle between them. Folks from Design & Engineering need to be able to switch between both views as quickly as possible.



Examples

Care should be taken to only include at most one primary and one secondary navigation per page. Using multiple navigations with the same style can cause user confusion.

Forcing a navigation to scroll is an established pattern on mobile devices, so it may be appropriate to use it in that context. Wrapping tends to make more sense on larger screens, where the user isn't forced to scroll passed a ton of navigation chrome.

Default

```
<div class="s-navigation">
    <a href="…" class="s-navigation--item is-selected">Product</a>
    <a href="…" class="s-navigation--item">Email</a>
    <a href="…" class="s-navigation--item">Content</a>
    <a href="…" class="s-navigation--item">Brand</a>
    <a href="…" class="s-navigation--item">Marketing</a>
</div>
```

Switching from light to dark mode

Adding the Colors to Stack Overflow

I solved all these color issues on the design system side with relative ease. Our design system has inherited fewer mistakes from our past, making it easier to refactor with the new future in mind. In order to ship to Stack Overflow, I needed to maintain our original Less variables for backwards compatibility. This allowed us to enable dark mode on certain parts of our interface incrementally.

7 https://smashed.by/aaronshekey
8 https://smashed.by/steveschoger
9 https://tailwindcss.com/

Since the majority of our interfaces built after 2018 use Stacks, they get dark mode and responsive layouts for free. The majority of our site, however? Not so much.

SITE CHROME

First, I'd need to make the largest changes I could without disrupting Stack Overflow's default light mode. These tasks were mostly just replacing static Less variables with their CSS variable equivalents throughout the site. I first applied `background-color: var(--white)` to the background of the site, replacing `background-color: @white`. This would now flip most of the page appropriately. I then did this for font colors. Rinse and repeat. Mostly, this actually meant *deleting* a lot of CSS, since we often were over-specifying font colors on child elements when we could just inherit from the parent.

STAFF SHIPPING

Once I got the broad strokes down, I leaned on engineers Adam Lear and Nick Craver to provide a method to ship a preview of dark mode to Stack Overflow employees. This would allow our staff to opt into a woefully broken dark mode, allowing folks to see how much conversion was left, but hopefully motivate them to help fix the portions of our site with the most traffic. This would let me fix the biggest barriers of the site – our existing codebase.

BUTTONS

If the view you're working on is already built with Stacks, there really isn't a ton you have to do to fix things for dark mode. You might decide you don't actually need a border, or you want to select a slightly different shade of gray for the background. Unfortunately, for the widest majority of the site, we still weren't relying on Stacks.

This was most obvious when it came to our buttons. Over the years, we had various implementations of buttons. The last was the most frustrating since we targeted the button element itself for styling. This means that any `button` or `input type=”button”` on the site would get default, super-specific styling from a deprecated set of styles.

This kicked off a large refactor that's still ongoing to delete element-level references to `button` in CSS, instead replacing them with their Stacks equivalent. For example, hundreds of `input type=”submit”` would need to be replaced with `<button type=”submit” class=”s-btn s-btn__primary”>`. To complicate things, we were often wiring up JavaScript interactivity to these visual selectors. If we changed the visual classes, it often broke what the button actually did. Across thousands of buttons, I needed to first add `js-` specific classes, wire them up, and then rip out the old visuals.

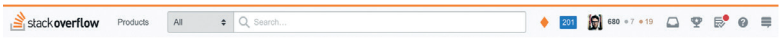
This eventually got me to the point of deleting a majority of the legacy button classes, allowing our buttons to switch colors properly when dark mode was enabled – all with few regressions to the light mode of our site.

THE SITE HEADER

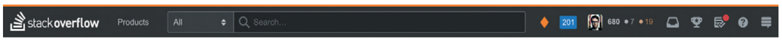
Complicating things even further, our site-wide header has several modes: light, dark, and themed. Both teams and our network sites force a dark appearance of the header. Additionally, our teams have a colored bar that's established by the team's avatar color. Like a lot of our components, the site header's CSS took a single color, measured if it was light or dark,

> We couldn't just rip this out and replace it with pre-baked CSS variables as we did on the design system. Our enterprise clients actually theme their headers entirely, using a single color to generate all the custom overrides.

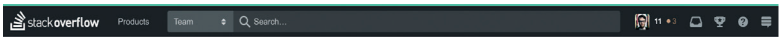
and then mutated that color through a complex set of Less functions. However, we couldn't just rip this out and replace it with pre-baked CSS variables as we did on the design system. Our enterprise clients actually theme their headers entirely, using a single color to generate all the custom overrides.



Light header



Dark header



Team header

For the light header that we ship to Stack Overflow, we needed to find a solution to measure if our color was a CSS variable or a static hex value. If it was a CSS variable, we'd skip the Less transformations entirely, building a header that would swap colors based on dark mode. If you passed a static Less variable instead, it'd then measure that color for lightness or darkness, and build the appropriate header.

Our approach ended up looking like this:

```
& when ( iscolor(@theme-topbar-background-color) ) {
     @theme-topbar-style: if(luma(@theme-topbar-background-
color) >= 50%, light, dark);
 }
 & when not ( iscolor(@theme-topbar-background-color) ) {
     @theme-topbar-style: automatic;
 }
```

I'd then build the header appropriately based on automatic, light, or dark.

TAGS

If there's one bit of advice I could give when designing a component: don't add layout to your component. In other words, your context should define how much space is between them. Don't bake it into your component. In Stack Overflow's earliest iterations, it was decided that our `post-tag` component would have outside margins applied to it. Like our buttons, tags ran into the same JS-targeting issue. To complicate things further, most tags were generated using a single helper method in our application.

Refactoring tags would mean swapping `post-tag` for our new theme-aware `s-tag` component. I'd also need to refactor our JS to target `js-tag` where appropriate. I'd also need to change our tag generator method to accept arbitrary layout classes, since, in certain contexts we might want to wrap our tags in a flex layout instead of relying on (or fighting against) pre-baked margins.

POST STYLING

The majority of Stack Overflow is user-generated posts. These posts display Markdown as the original question body as well as answers and comments. At the time of Stack Overflow's launch, Markdown was relatively new.

Over the years, the industry has coalesced on some standard ways of displaying things like headers and blockquotes. Dark mode was a perfect time to reconsider how we handled some of our post formatting – the most controversial being block quotes.

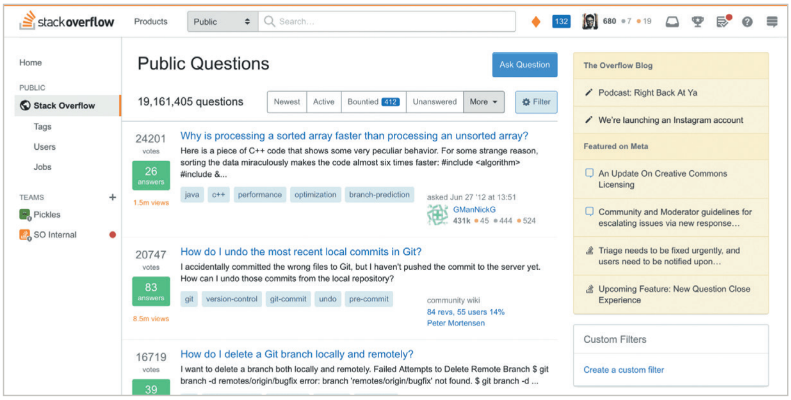
We originally implemented block quotes with an overpowering yellow background that reduced the contrast of the quote itself.

The yellow was also problematic when displayed against a dark background. Ultimately, we switched to the industry-standard single gray bar to represent block quotes.

CODE STYLING

For Stack Overflow, we very clearly have a lot of code to display. Our syntax highlighting colors used completely unbranded colors I'm pretty sure we inherited from the original syntax highlighting library we hacked in. Ultimately, I punted on a heavier redesign of syntax highlighting. Instead, I ended up shifting the existing syntax highlighting toward colors from our design system's values, finding dark mode equivalents that didn't make too big a change too soon.

The Results



Public Questions

19,161,405 questions

24201 votes · 26 answers · 1.5m views

Why is processing a sorted array faster than processing an unsorted array?

Here is a piece of C++ code that shows some very peculiar behavior. For some strange reason, sorting the data miraculously makes the code almost six times faster: #include <algorithm> #include &...

java c++ performance optimization branch-prediction

asked Jun 27 '12 at 13:51 GManNickG 431k • 45 • 444 • 524

20747 votes · 83 answers · 8.5m views

How do I undo the most recent local commits in Git?

I accidentally committed the wrong files to Git, but I haven't pushed the commit to the server yet. How can I undo those commits from the local repository?

git version-control git-commit undo pre-commit

community wiki 84 revs, 55 users 14% Peter Mortensen

16719 votes · 39

How do I delete a Git branch locally and remotely?

I want to delete a branch both locally and remotely. Failed Attempts to Delete Remote Branch $ git branch -d remotes/origin/bugfix error: branch 'remotes/origin/bugfix' not found. $ git branch -d ...

The Overflow Blog

- Podcast: Right Back At Ya
- We're launching an Instagram account

Featured on Meta

- An Update On Creative Commons Licensing
- Community and Moderator guidelines for escalating issues via new response...
- Triage needs to be fixed urgently, and users need to be notified upon...
- Upcoming Feature: New Question Close Experience

Custom Filters

Create a custom filter

Dark mode beta's debut on March 30, 2020

With these refactors in place, we can make larger changes with fewer regressions. We can far more easily consider extending our color palette to include high-contrast accessibility modes.

Building a feature like dark mode is the result of a fundamental shift toward designing systemically at Stack Overflow. I've been pushing adoption of our design system over the last year, using dark mode as an opportunity to rebuild many parts of our products. This is the first of many projects to bring more accessibility to our users.

Not counting the deprecation of Internet Explorer 11, work on dark mode started in earnest in July 2019 with an exploratory pull request.[10] Prior to that, you can see some public discussion[11] of what it took to build a dark mode in April 2019. A proof of concept in the production codebase was hacked together in October 2019. After at least 60 follow-up pull requests, the dark mode beta went live on March 30, 2020.

Stack Overflow Key Takeaways

Reengineering for dark mode provides an opportunity to create the foundation for an accessible design system.

Stack Overflow introduced dark mode in March 2020. Besides identifying the right color set for dark mode, this resulted in an overhaul of their Stacks design system. Enabling dark mode on different interface components presented plenty of challenges. For example, colors within CSS were static earlier and modified using transformations for different states of individual elements (a button with a blue background transforms to dark blue on hover). However, with different modes available, the base color was a variable. CSS classes had to be enhanced to use the `prefers-color-scheme` media feature.

Similarly, there were other challenges in standardizing components across the site. However, the exercise helped lay down the foundation for Stacks components to support high-contrast accessibility modes and other accessibility requirements.

10 https://smashed.by/stackspr
11 https://smashed.by/stacksdiscussion

How Intercom Approached Messenger Accessibility

By Daniel Husar

At Intercom, our mission is to make internet business personal.[1] But in order for an internet business to be personal, it must also be possible for everyone to access. More than one billion people worldwide live with a disability – that's more than 15% of the global population. Without assistive technologies like screen readers, the web is inaccessible or hard for them to use. Think of it this way: it's like entering your neighborhood coffee shop and discovering that the counter is too tall for you to reach – because you're using a wheelchair.

We believe businesses should be able to communicate with everyone on their website, regardless of how their visitors interact with the web. This isn't just a company philosophy; it's also an engineering commitment. To prioritize accessibility in our Messenger,[2] we took a hard look at the technical improvements we needed to make and turned what were often fuzzy requirements into real, meaningful solutions.

What we achieved is making our web Messenger accessible and compliant with the Web Content Accessibility Guidelines 2.0 Level AA.

A Shared Framework for Web Accessibility

The Web Content Accessibility Guidelines (WCAG) are a shared set of technical standards that explain how to make web content accessible to

1 The original version of this article was published in December 2018: https://www.intercom.com/blog/messenger-accessibility/
2 https://smashed.by/messenger

people with disabilities. Its twelve guidelines are organized around four main principles, which provide the foundation for web accessibility:

- Perceivable: Users must be able to perceive the content in some way, using one or more of their senses. For instance, images that convey meaningful information should have alternative text provided.
- Operable: Users must be able to control UI elements. For example, all functionality like buttons and form elements should be accessible using keyboard controls.
- Understandable: The content must be understandable to its users. That means things like the language of the page should be detectable in the code.
- Robust: The content must be developed using well-known and adopted web standards. In other words, your code should be easily parsed and interpreted by different browsers and user agents like screen readers.

Our engineering work started by exploring the WCAG guidelines and then identifying all the areas in our web Messenger that needed improvement. As we quickly learned, turning these four principles into real solutions was simpler on paper than in practice.

Turning Fuzzy Requirements into Real Solutions

The WCAG guidelines are extensive – across the four principles, there are nearly 100 sections – and some areas are quite fuzzy. Requirements like “meaningful sequence” and “focus order” are very broad in scope, especially for applications like ours that get embedded in many different environments.

These fuzzy requirements meant there wasn't always a direct or obvious correlation between the WCAG guidelines and what we needed to build. We encountered issues that didn't have clear answers online, leaving it up to us to come up with the right technical solutions. In the end, we identified three main areas of focus for accessibility in our web Messenger:

- Keyboard navigation
- Screen reader support
- Color contrast

I'll walk you through each of these areas, what we learned, and the solutions we shipped.

Improving Keyboard Navigation

Keyboard navigation is a very important part of making your app accessible. When visually- or auditory-impaired people use web browsers, they often rely on keyboard navigation to tab into fields and then have their screen reader read what action could be performed. Our work on keyboard navigation can be broken down into three main changes:

1. Making elements clickable by keyboard
2. Setting proper focus states
3. Designing intentional focus traps

1. MAKING ELEMENTS CLICKABLE BY KEYBOARD

The Intercom web Messenger is a React app. If we want keyboard navigation to work with the Messenger, every `onClick` handler that is added to an element, except the elements that browsers support natively, also needs `onKeyDown`, which checks if the **Enter** or **Space** keys were pressed and execute the same function as the `onClick` handler.

Let's imagine a scenario where our component looks like this (these examples include React components using JSX. You can learn more about JSX here.[3]):

```
export default function(props) {
   return <div onClick={props.onClick}>Open modal</div>;
};
```

In order to make this component keyboard accessible, we can convert it to a button:

```
export default function(props) {
   return <button onClick={props.onClick}>Open modal</button>;
}
```

This might not always be easy because your component might be quite complex; for instance, you probably don't want to wrap your whole app in a `<button>` element – and buttons have specific styling.

Another approach to make this component accessible is to add `onKeyDown`, `tabIndex` and `role` attributes to it:

```
export default function(props) {
   return <div
   onClick={props.onClick}
   onKeyDown={(e) => (e.keyCode === 13 || e.keyCode === 32) &&
   props.onClick(e)}
   tabIndex="0"
   role="button"
   >Open modal</div>;
};
```

The problem is that adding these three attributes to every clickable element in your app is quite a lot of work, and it's easy for engineers

3 https://smashed.by/jsx

adding new functionality to forget to add these attributes or use the `<button>` element instead.

Building an Automated Solution

To make keyboard accessibility the default condition, we wrote a custom babel plugin[4] that automatically adds `onKeyDown`, `tabIndex` and `role` attributes to all elements where:

1. Browsers do not natively support `tabNavigation`.

2. Browsers do not natively trigger `onClick` handlers when a user hits the **Enter** or **Space** keys.

Since babel transforms JSX into regular JavaScript function calls, it's easy to statically determine components that need to have keyboard events added to them. Our custom babel plugin now handles the majority of our keyboard navigation issues. Here's an example of using babel-plugin-react-add-a11y-props within a React app: https://smashed.by/babelsandbox

For the folks wondering about the performance implications of having an arrow function in the render method, it really is fine, but since your mileage may vary, you should always measure your performance before you optimize.

While this plugin will add keyboard navigation to all elements with `onClick`, if something behaves like a button, the best solution[5] is still to change it to an actual `<button>` element.

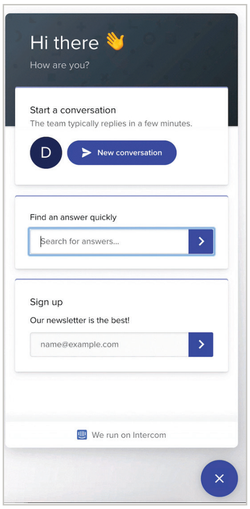
2. SETTING PROPER FOCUS STATES

In order for your app to be navigable by keyboard, you need proper focus states. Focus states help users (and their screen readers) understand where they are in the app and what elements are being selected.

4 https://smashed.by/babelplugin
5 https://smashed.by/ryanflorence

A good rule of thumb: if a user is relying on keyboard navigation, there should be a visual indicator to highlight which element currently has focus. In our case, we've designed our visual indicators to show only if we detect you are using keyboard navigation. That way, we minimize visual noise for our mouse users. You can use the CSS `focus-visible` property to provide a different focus indicator based on the user's input modality, keyboard, or mouse.

Here is our Messenger with the focus indicator turned on:

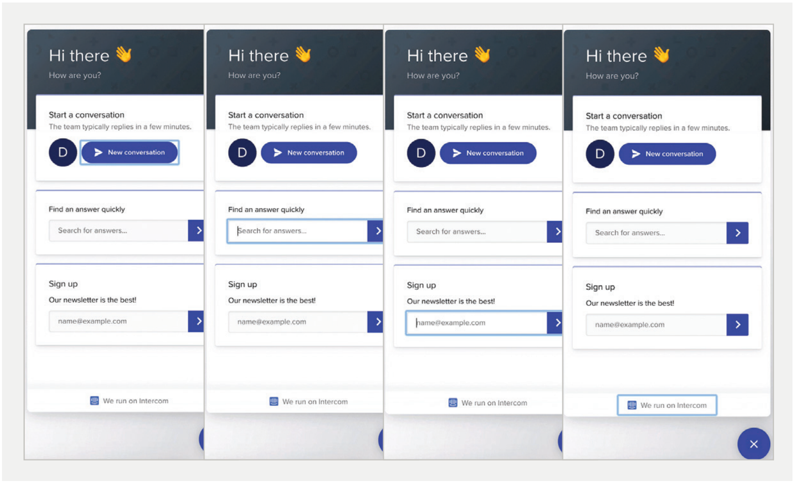


3. DESIGNING INTENTIONAL FOCUS TRAPS

For keyboard navigation to work, you may need to set intentional focus traps. Focus traps refer to times when a user hits the **Tab** key or **Shift** + **Tab** keys, and they're placed in a certain cycle of focusable elements.[6] The most common example where you would want to set up a focus trap is a modal.

In our case, while the Messenger is open, we've set a focus trap so users are not able to tab outside of it. That way, users are able to navigate all of the elements in the Messenger without having to navigate through the entire webpage. Here is our Messenger's focus trap in action:

6 https://smashed.by/dialogmodal



Setting a focus trap is usually a complex task. Focus from the last element should jump to the first one and when going backwards, the focus should jump from the first one to the last one. For that to work, you need to calculate all the focusable elements, set up proper event listeners and have it flexible enough that you can override those rules.

We have created an open-source library[7] to quickly and easily create focus traps. This library provides a high-level API that will handle the focus traps. A simple example is to pass the DOM element in which the focus should be trapped:

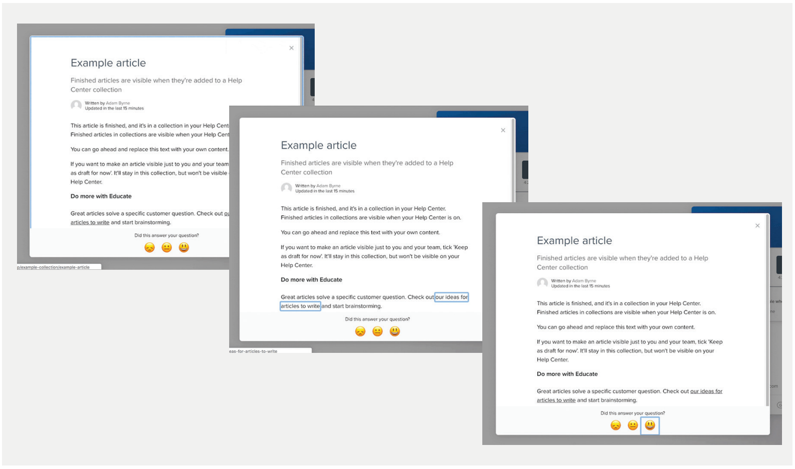
```
const trap = new FocusTrap({
   node: document
});
```

You can check out the focus traps in action here:
https://smashed.by/focustrapsandbox

Just as it's important to set intentional focus traps, you should always provide a way for keyboard users to exit those focus traps.

7 https://smashed.by/focustraplibrary

You can see that in the example below. We've also designed it so that after the modal is closed, the focus returns back to the element that originally opened the modal. Here it is in our Messenger:



Example article

Finished articles are visible when they're added to a Help Center collection

Written by Adam Byrne
Updated in the last 15 minutes

This article is finished, and it's in a collection in your Help Center. Finished articles in collections are visible when your Help Center is on.

You can go ahead and replace this text with your own content.

If you want to make an article visible just to you and your team, tick 'Keep as draft for now'. It'll stay in this collection, but won't be visible on your Help Center.

Do more with Educate

Great articles solve a specific customer question. Check out our ideas for articles to write and start brainstorming.

Did this answer your question?

Supporting Screen Readers

Screen readers are software applications that allow visually-impaired users to read text that is displayed on their computers. Together with keyboard navigation, providing full support to screen readers was crucial to making our Messenger accessible.

Our work on supporting screen readers can be broken down into four main changes:

1. Making text content readable and navigable
2. Defining states and properties with ARIA attributes
3. Adding visually hidden text
4. Removing mouse hover states

1. MAKING TEXT CONTENT READABLE AND NAVIGABLE

The first thing we did was set the language attribute on `html` elements. Screen readers use this attribute to determine the language of the page and read the text in its intended way.

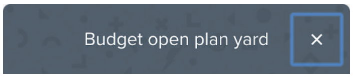
The second thing we did was add semantic markup to our Messenger. Since our Messenger is a single page app that you embed, SEO doesn't apply to it and semantic markup hasn't been a priority. To support screen readers, we updated our code to include semantic elements like headings, paragraphs and labels.

2. DEFINING STATES AND PROPERTIES WITH ARIA ATTRIBUTES

To fully support screen readers in our web Messenger, we used WAI-ARIA attributes. Accessible Rich Internet Applications (ARIA) are a set of attributes that supplement HTML so screen readers can handle common interactions like form hints and error messages, live content updates, and more.

Establishing Non-Text Elements with aria-label

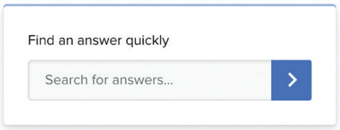
We have added the `aria-label` attribute to all elements that have onClick handlers, but it's not clear to screen readers what the intended functionality is. The most common scenario is when elements have decoration styles without any text. The screen reader will read the `aria-label` when the keyboard is focused on that element. The close button is good example:



We also support an ecosystem of apps[8] that are built on top of our Messenger. Previously it was not possible to make Messenger apps

8 https://smashed.by/intercomapps

accessible. Now we have extended our framework with `aria-label` attributes so every Messenger app can be fully accessible. For instance, the Article Search app[9] is accessible with `aria-label` attributes for the input field and submit button:


Find an answer quickly

Search for answers...

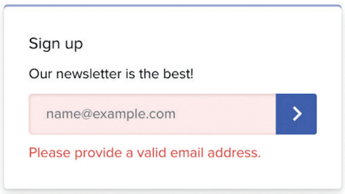
Announcing Dynamic Changes with aria-live

Our Messenger behaves like a single page app. To support dynamic changes to our content without page reload, we added `aria-live` attributes to our app.

Our `aria-live` attributes tell screen readers to watch for changes in selected DOM elements, and any DOM mutation inside of it will be announced. We've wrapped our whole app with this attribute as all changes that are made need to be presented to users. We've also wrapped various parts like our conversation view in an `aria-live` attribute so when a new message is received, the screen reader will announce it first.

Indicating Error States with aria-invalid

All error states need to be properly announced to screen readers. Previously, error states on the input would be represented only with CSS classes. To make those error states visible to screen readers, we've added `aria-invalid` attributes to inputs with errors and `role="alert"` to the error messages. Here is the error state for the Mailchimp app[10] in our Messenger:


Sign up

Our newsletter is the best!

name@example.com

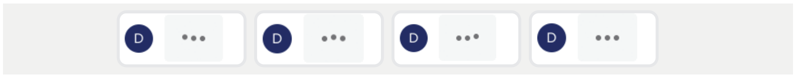
Please provide a valid email address.

9 https://smashed.by/intercomarticle
10 https://smashed.by/mailchimpapp

3. ADDING VISUALLY-HIDDEN TEXT

There are some scenarios where it's only clear visually what components do. Without the visual indicators, these elements are meaningless or confusing. In these cases, we've added visually-hidden text to help screen readers interpret what's happening.

The typing bubble in our Messenger is a good example of this:



While this makes it clear visually that somebody is typing, screen readers have no way of processing or communicating that. We've added hidden text inside the speech bubble for screen readers to reference. You can visually hide text with this CSS snippet:

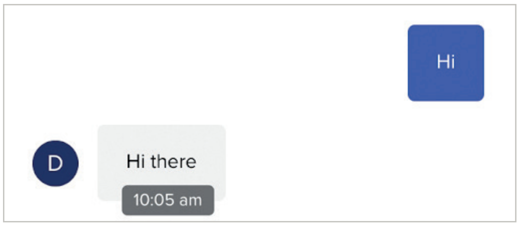
```
.visually-hidden {
   position: absolute !important;
   clip: rect(1px, 1px, 1px, 1px);
}
```

The HTML might look like this:

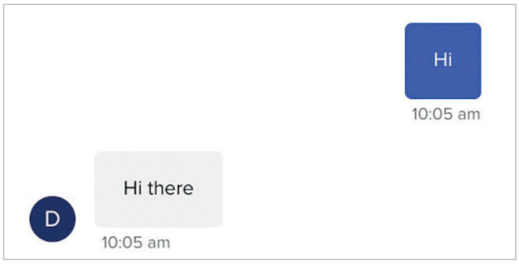
```
<div class="typing-admin-bubble">
   <div class="typing-admin-dot-1"></div>
   <div class="typing-admin-dot-2"></div>
   <div class="typing-admin-dot-3"></div>
   <div class="visually-hidden">Is typing.</div>
</div>
```

4. REMOVING MOUSE HOVER STATES

Any functionality that is available just on mouse hover, such as tooltips, should be accessible to screen readers too. In our case, we display timestamps in conversations when you hover over the specific message:



Since timestamps are very useful information for screen readers, we have opted to show timestamps all the time for screen readers:



Optimizing Color Contrast

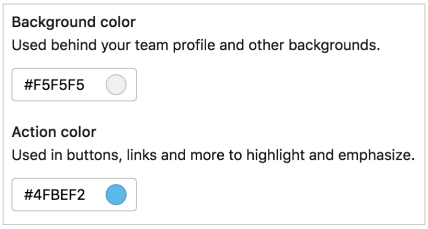
For users with visual impairments like colorblindness, high contrast between colors make it easier to read text content. The recommended contrast ratio for accessible content is 4:5:1 between the text and background colors.

We split our color contrast work into three buckets:

1. Issues caused by customizable colors
2. Issues caused by non-customizable colors
3. Supporting high contrast mode in Windows 10

1. ISSUES CAUSED BY CUSTOMIZABLE COLORS

These are issues caused by customers who customize the Messenger and choose colors that don't match the recommended contrast ratio. For example, teammates can choose the background and action colors of the Messenger:



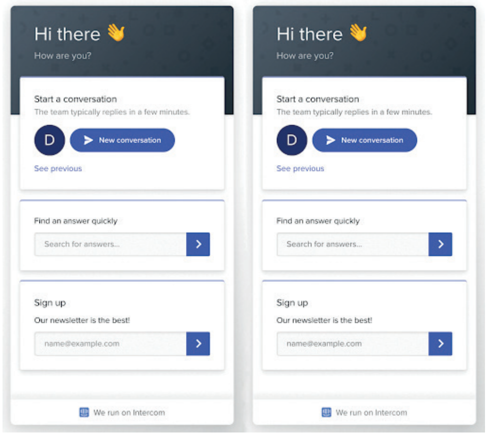
Background color
Used behind your team profile and other backgrounds.
#F5F5F5

Action color
Used in buttons, links and more to highlight and emphasize.
#4FBEF2

To help, we've published documentation[11] on how to choose colors for your Messenger while maintaining accessibility.

2. ISSUES CAUSED BY NON-CUSTOMIZABLE COLORS

These are issues caused by the colors that are hardcoded in our codebase. We have inspected all the hardcoded colors we have in the Messenger. After we identified all the colors that had to change, our designers prepared alternative colors that matched the contrast ratio. You can see the before and after here:

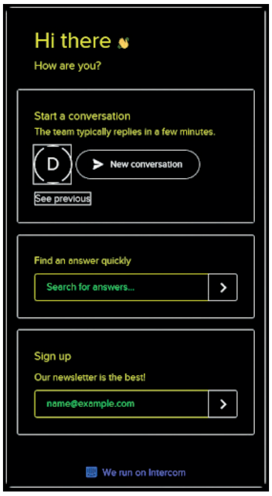


On the left is the Messenger before we made the update. On the right is the Messenger with colors matching the 4:5:1 contrast ratio.

11 https://smashed.by/intercomdocs

3. SUPPORTING HIGH CONTRAST MODE IN WINDOWS 10

We've even added support for high contrast mode in Microsoft Windows 10.[12] High contrast mode is specifically designed for visually-impaired people to consume content more easily.



Ensuring the Future Accessibility of Our Messenger

We offer many customizations in our Messenger. While we want our customers to be able to customize the Messenger to fit their brand, it's not always easy to keep accessibility in mind. That's why we've published a set of guidelines[13] on how to customize the Messenger to be accessible. Guidelines include what colors to pick, which Messenger apps to use, how to send media content and attachments in conversations, and more.

We've also put automated tooling in place to prevent any regressions in the code. We've implemented two main tools: ESLint[14]

12 https://smashed.by/highcontrast
13 https://smashed.by/intercomguidelines
14 https://smashed.by/eslintjsx

and react-a11y. ESLint helps us statically check our codebase for any accessibility issues. React-a11y[15] is a runtime validator that works in our integration tests and will validate accessibility before we ship any changes to production.

Intercom Key Takeaways

Building accessibility into a customizable Messenger so that it could be accessed by anyone who wants to add a personal touch to their internet business.

The Intercom business Messenger appears as a chat widget on websites. To make their Messenger accessible, the team improved keyboard navigation and color contrast, and introduced support for screen readers. A few of the unique issues they addressed were:

- Designing intentional focus traps: When tabbing through elements inside the messenger, the focus should not go outside after the last element. This requires setting up focus traps to reset the focus to the first element.
- Screen reader support for Messenger-specific features: In addition to reading out live text messages, the Messenger can inform screen reader users when another person is typing (audio indicator for "typing..."). Screen readers can also read the timestamp for a message when the user hovers on it.
- Color contrast with customization: Messenger customers can customize it for their businesses. Some colors may make the Messenger inaccessible to the customers of that business. To avoid this, the team has published guidance on choosing colors.

15 https://smashed.by/reacta11y

Making Internet Business Personal and Possible

Every day, thousands of businesses use Intercom to talk to their customers. That's hundreds of thousands of people, or more, who communicate with each other using our Messenger. And while not everyone experiences web content in the same way, using the Messenger should always feel personal and, just as important, be possible.

Making our web Messenger accessible – the engineering work and changes – ended up being a small technical commitment compared to its huge and ongoing impact. At the end of the day, we want the Messenger to be the kind of space online that feels like walking into your neighborhood coffee shop and knowing it's designed to accommodate you.

> Making our web Messenger accessible – the engineering work and changes – ended up being a small technical commitment compared to its huge and ongoing impact.

Shopping Platforms: Accessibility Is More Than a Technical Problem

By Devon Persing

Digital accessibility is more than a technical problem to solve,[1] although many organizations approach it as something that can be fully addressed by development and testing. However, approaching accessibility in a sustainable way requires appreciating the complexity and breadth of disabilities that impact your users, and understanding how accessibility impacts every part of your product or service, as well as your organizational values and goals.

I've been doing accessibility work full-time for about 11 years, after a short career in libraries, so I'm pretty familiar with some of the myths surrounding accessibility work. I've worked as a consultant, both solo and in agencies, as well as in product companies. It's through that product company lens that I'm approaching this article, with the idea that you can improve accessibility programming from within. In this case study, I'm going to help you rethink accessibility work and give you some practical tips for making your products and services more accessible, more easily. To do that, I need to debunk some myths.

Myth #1: Disability is Simple

There's a myth around accessibility work that disability is simple. Or, maybe a better way of saying this is that disability is monolithic. However, "disabled" is not a user type. Disability is often

1 The original version of this case study was published in April 2021: https://smashed.by/shopifya11y

dynamic and situational, and impacts every disabled person's experiences differently.[2]

To start, it's important to know that about 26% of adults in the United States,[3] 22% of adults in Canada,[4] and 15% of people worldwide have a disability that affects their daily lives.[5] (How people measure and count disability varies across nations and cultures, but it's safe to say that approximately a quarter of all people have at least one disability.)

There are a few common types of disabilities that relate to digital spaces, which can be organized into a few categories:

- **Dexterity and mobility**, which impacts how a person physically interacts with devices.
- **Cognitive and neurological**, which impacts how a person takes in, processes, and remembers information and sensory input.
- **Vestibular and motion**, which impacts how people experience visual motion, as well as the physical impacts of motion or perceived motion on the body.
- **Vision**, which impacts how and how much a person can see.
- **Hearing**, which impacts how and how much a person can hear.
- **Speech**, which impacts how or whether a person speaks or communicates verbally.

Using the social model of disability,[6] disability is a mismatch between a person and the environment that has been designed.[7] The negative impacts of disability are caused by systemic barriers, attitudes, and exclusion in society, not a failing of the person with a disability, nor something to be "fixed" or "overcome" in the person.

2 I use the phrase "disabled person" to describe myself, but you may prefer "person with a disability."
3 https://smashed.by/cdc
4 https://smashed.by/cdccanada
5 https://smashed.by/who
6 https://smashed.by/socialmodel
7 https://smashed.by/rethinkingdisability

To make things even more complex, many people experience disabilities that vary day to day. As Brianne Benness says: "In mainstream culture and media, 'disabled' usually refers to people with static and visible disabilities. [...] And so, if I tell somebody that I am disabled, I must explain that not all disabilities are visible and also not all disabilities are static."[8] For example, I have two conditions, fibromyalgia and ADHD, which make my day-to-day very different, depending on how much stress I'm under, whether I've been sleeping, and many other factors.

Assistive Tech Is More than Screen Readers

Often when I talk to designers and developers about assistive technology, they get stuck on the idea of the screen reader experience being *the* accessibility experience we need to work for. But, considering the broad diversity in disabled experiences, people with disabilities use a wide variety of assistive software and hardware tools to connect to technology, and some people with disabilities don't use assistive tech at all.

Here are just a few examples of the *many* types of tech disabled folks use.

TECH FOR DEXTERITY AND MOBILITY DISABILITIES

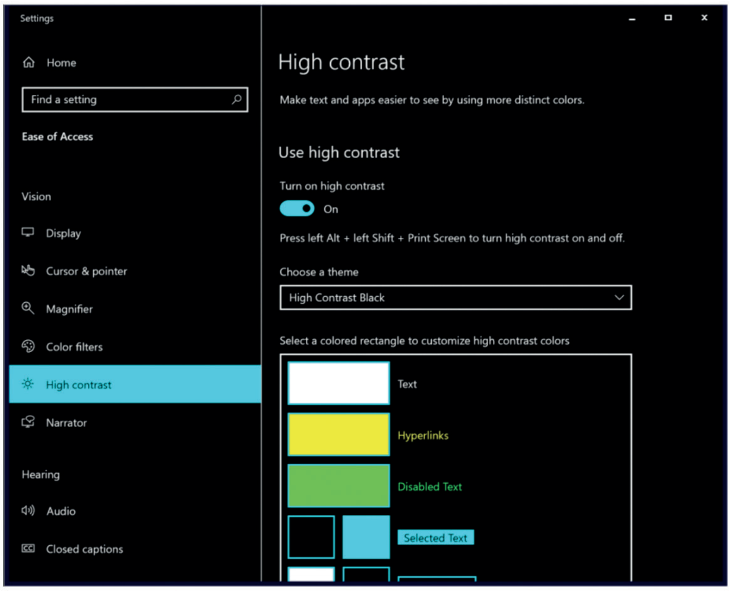
One of the most common categories of assistive tech is for people with dexterity and mobility issues. These might be caused by an injury (even temporary ones), limb difference, paralysis, or chronic pain. (Over 22% of Americans have arthritis, fibromyalgia, or similar chronic pain disorders,[9] so chronic pain is quite common!) These tools might make it easier for a person to use a keyboard and/or

8 https://smashed.by/dynamicdisability
9 https://smashed.by/arthritis

mouse, or might replace those altogether. There are eye-tracking tools that let you interact without touching a device at all, for example. There are similar hardware and software combinations that let users interact through a switch device, by pressing simple buttons or performing small movements with their head or mouth to control the mouse. There are yet other tools that allow people to control their devices with their voice alone, such as Dragon on Windows and Voice Control on Mac devices, as well as highly customizable keyboards designed for use with one hand, or which replace keyboard keys with large paddles.

TECH FOR VISION AND VISUAL SENSORY DISABILITIES

While most folks familiar with accessibility are familiar with screen readers, there is a wide variety of other tools available for people with issues related to vision and visual input.



High contrast

Make text and apps easier to see by using more distinct colors.

Use high contrast

Turn on high contrast: On

Press left Alt + left Shift + Print Screen to turn high contrast on and off.

Choose a theme: High Contrast Black

Select a colored rectangle to customize high contrast colors

Text

Hyperlinks

Disabled Text

Selected Text

Screenshot of high contrast settings in Windows 10, showing the default colors, with a black background and bright colors to indicate links, text, and other elements.

A common vision tool is the high contrast theme in Windows, which allows users to change colors across the operating system and in the browser. This is used by people with vision issues, as well as visual sensory issues. It color-codes different types of content, based on the user's settings and based on how the page is marked up. There are other tools that invert colors to add permanent "dark modes" for content that many people use as well.

For people who aren't on Windows, or who have other preferences, dark mode (or light mode!) at the system level or through browser plug-ins is a legitimate accessibility need. People with photosensitivity or who can have migraines triggered by certain types of contrast may use these settings. Tools and settings to reduce motion or turn off animations are also helpful for people with conditions that are triggered by unnecessary movement or animation. Personally, I get migraines if I look at content that is on a bright white background or has a lot of motion. Slack threads full of animated emojis and gifs are a nightmare, for example, and I'm grateful that I can just turn all that off.

Users with vision issues also often use magnification, and text or content resizing in the browser. This gives users control over what part of the screen they see at a given time, and makes information easier to read.

TECH FOR COGNITIVE AND NEUROLOGICAL DISABILITIES

Options to reduce motion or adjust other sensory inputs can also be extremely helpful for people with cognitive and neurological disabilities. It's critical to prevent triggering seizures. Also, people with ADHD, brain injuries, and other conditions may rely on dark or light mode or motion reduction to prevent sensory overload and headaches. There are also plug-ins that allow users to customize colors related to text, which can help folks with reading disabilities.

For people who have trouble reading on busy web pages, Reader Mode in the Safari and Firefox web browsers allows you to strip out all ads, navigation, sharing buttons – anything that's not an article – while you're reading. It also gives you options for changing fonts and colors. It's designed to make it easier to read without distractions or without complicated layouts, which can be a huge help for folks with reading disabilities and disabilities that impact focus and attention.

Prevent Assistive Tech Barriers

We all need a greater awareness of how people use (and don't use) assistive tech. It's possible to find some accessibility barriers with automated testing, but the vast majority of websites and apps are too complex to rely only on an automated solution. It's important to be able to test behavior as well as check for basic issues.

Pre-Covid, many tech organizations had device labs or other in-house solutions for device testing. These often served as a way for teams to do testing with different types of assistive tech. With many product teams continuing to work remotely, there need to be other options.

One option is to build your own "virtual" assistive tech lab based on the tech that your customers (or potential customers) are probably using. This requires educating teams about how to test with assistive technology effectively, which has its own learning curve but leads to a deeper understanding of how users might actually interact with your product. To be effective, this type of effort requires documentation and clear guidance about how and when to use assistive tech when testing new features and products.

Another option is to work with a vendor that provides virtual machines specifically for accessibility testing. This will give you access to common assistive technologies without the overhead of managing your own virtual machines, but does still have that learning curve.

Focus on the Experience

When I teach workshops about disability and accessibility, I often ask my students to do a matching exercise to map good UX practices to the types of disabilities they might help. My students quickly learn there are no one-to-one relationships between accessibility best practices and individual types of disabilities. The interconnections show us that none of these individual types of disabilities are experienced in a vacuum.

Here's an example of how we might map UX experience to disability categories:

EXAMPLE PRACTICE	DEXTERITY	COGNITIVE	VESTIBULAR	VISION	HEARING	SPEECH
Keyboard support	✓			✓		
Use of color		✓		✓		
Clear labels	✓	✓		✓		
No auto-playing video		✓	✓	✓	✓	
Captions and transcripts		✓		✓	✓	
Text-based commands for virtual assistants like Siri						✓

The upside to this level of complexity is that it forces us to give up the notion that categories of disability are silos. Instead, we can focus on how these different experiences are supported, rather than trying to over-engineer solutions for any one audience or disability type. This actually makes it easier to think about disability as a collection of experiences that can be met with best practices, not a monolith of people or stereotypes.

In this way, we can focus on a few general types of experiences:

- Dexterity and mobility barriers caused by pain or other factors can result in a variety of different ways of touching or interacting with hardware and software. These might even affect how people hold a device, and whether a user might touch a device at all, or rely fully on voice activation or other tools.

- The wide variety of neurodiversity means we need to think broadly about how people think, process, and sense information. No two people think alike, and people with cognitive disabilities typically benefit from the simple language, clear organization, and consistent workflows that help all users.

- Media that relies heavily on visuals, color, or sounds needs to have alternatives for people who can't or prefer to not to take in information in those ways. Many people are visual learners, but considering how people who don't take in information visually strengthens how we design visual or color-based experiences, making our decision-making to use visuals or colors even stronger.

- People with vestibular or mobility issues need to have control over how they interact with motion, or how they move. Thinking about low-motion experiences makes us focus on the real goals we want users to achieve, and how we can use motion to guide those experiences. It forces us to make workflows and transitions that are clear even without motion.

- And people with disabilities that impact speech and people who are nonverbal need non-speech based interfaces for tools like virtual and home assistants.

Include Disabled People in the Process

To better understand the experiences of disabled users, invite disabled people into your usability and inclusivity work. You'll never be able to test for every use case, but engaging in research with participants with disabilities is the best way to ensure the experiences of disabled people are included. This helps designers and developers get better insights into making usable, accessible experiences from the start.

To find participants in this kind of research, there are a few options. One is to survey your existing user base to see if they use assistive tech.[10] You don't even have to call it assistive tech! You can simply ask if folks use a screen reader, switch device, and so on. You can also reach out to organizations in your area that serve disabled people to do your own recruitment, or contract with a company that performs research with disabled users.

Just as critical, however, is considering who is designing, building, and testing your products today. If you are not hiring disabled people to do those jobs, it's a good bet that accessibility is a challenge for your organization. Look into how accessible your organization's hiring process is, and invest in recruiting and supporting disabled employees.

Myth #2: Accessibility is a Technical Problem

To really deal with digital accessibility, we have to go beyond fixing things, and start preventing them. The lack of accessibility, and how to address it, is a cultural problem rooted in ableism. Most of our resources and standards around accessibility are technical and oriented towards testing. And, since so much accessibility work focuses

10 Asking what types of assistive technology a person might use is a way to find participants who fall into the categories we've been discussing. Some people who use assistive tech might not identify as disabled or having a disability, and this also avoids asking people about sensitive medical information.

on fixing things that have already been implemented, developers are often given the responsibility.

MEASURING ACCESSIBILITY

The primary tool we use to measure accessibility is the Web Content Accessibility Guidelines, a technical document created by a working group within the W3C.[11] Not to knock the extremely important work that these folks do, but this approach compounds a testing-oriented culture around accessibility that puts the onus on developers and testers. This results in a lot of accessibility work being done at the end of a project, in a workflow that often starts with auditing sites and apps that are already in the wild, then fixing issues, but not digging into the processes and workflows that caused those problems in the first place.

This has also led to "solutions" like third-party overlays that promise to solve complex accessibility issues with the click of a button, but usually cause more harm than good. (Colleagues in the field have put a useful resource together on overlays if you'd like more information.)[12]

Resources and guidance for designers, writers, researchers, and others are minimal and repetitive. If you're in one of these roles and you've tried to find resources on how to integrate accessibility into your practice, you've probably seen the same advice over and over again: "Use good color contrast!" "Use simple language!" "Test with users!" There is a lot of *why*, and not a lot of *how*. And that's because the how is going to vary from project to project, team to team, and organization to organization.

Instead, try:

- Holistic, continuing education about how disability and technology intersect.

11 https://smashed.by/wcag
12 https://overlayfactsheet.com/

- To include people with disabilities as part of your team and process.
- Continuous improvement of processes and workflows to move beyond technical guidelines to usability.
- To make accessibility part of the current work, not a future goal.

As a place to start, teams can review usability feedback from users with disabilities, acquaint themselves with the assistive tech available on the devices they support, and look at any reported issues for their product. These steps can help teams and team leads think about when they might insert specific steps to avoid accessibility issues in their workflow.

Ideally, a product workflow with accessibility included from the start looks something like this:

1. Users with disabilities are included in the product audience from the start.
2. Accessible experiences are included in design decisions.
3. Prototypes for new work are tested for usability, including with users with disabilities.
4. Built solutions leverage automated testing, and testing with common assistive tech.
5. If issues are reported by users after the product is released, those issues are triaged and addressed by severity and priority along with any other issues.

A lot of accessibility education focuses on developers, designers, and content creators, but doesn't support the people who manage those

UX practitioners. A critical addition was building out training materials for managers to help them better evaluate how literate their teams are in accessibility, and how to better support accessibility work in their processes, rituals, and hiring practices.

Myth #3: Accessibility is Hard

Accessibility work doesn't *have* to be hard. Everything is hard when you don't know enough about it. Think back to when you first started learning your craft, gaining real experiences, learning new tools and standards, and sometimes failing. You have to celebrate small wins! Those wins just don't represent the *end* of improvement.

And your goal doesn't have to aim for expertise. Expertise is hard to teach because it takes a long time. I also don't think it's possible to really teach empathy. Instead, we should focus on ways to make accessibility just another part of every process to create products. **Accessibility work at scale is an exercise in literacy and practice, not expertise or empathy.**

To improve the accessibility of your work, here are some accessibility literacy aims, borrowed from information literacy in library science:

- Learn how to discover resources about accessibility efficiently.
- Evaluate the usefulness and accuracy of resources.
- Understand the context in which those resources were created.
- Create new work using what you have learned.
- Participate in a community of practice to reinforce and scale learning.

ENABLING BEST PRACTICE THROUGH RESEARCH AND ITERATION

In a prior product company, I had the enormous benefit of working alongside a user experience research (UXR) team. We were working on the almost identical problem of scaling accessibility literacy and research literacy in the same organization. This meant that we got to iterate on each other's experiments with tooling, education, and processes, with the aim of creating a consistent, literacy-focused methodology for creating user-focused activities and resources across the organization.

Even if you don't have a strong UXR team, there are other ways to iterate. Many organizations have gone through major changes to address localization and other cultural differences, workflows and tools, and other aspects of their product work. If your organization had a particularly successful campaign to change how people work, study that to get ideas about how you might grow accessibility.

Accessibility work at scale is an exercise in literacy and practice, not expertise or empathy.

SHARE WINS

Another way to grow community is to create an accessibility guild across the organization. It can be a place for teams to share their accessibility wins, to ask questions from internal and external accessibility experts, and generally build a more sustainable community of practice around accessibility. This is a great way to turn accessibility improvements into learning opportunities for other teams, instead of always relying on an accessibility specialist or small accessibility team to do that teaching.

For more formal programming, adding accessibility to project requirements is also a huge step. This allows teams to formally acknowledge their wins at demos, town halls, or whatever other rituals your organization has around your workflows and processes.

LIVE UP TO YOUR CURRENT GOALS

As both an accessibility specialist in organizations and as a consultant, I often found that organizations had values, mission statements, diversity and inclusion programs, or other foundational beliefs that *should* have prioritized accessibility, but did not in practice. This comes down to ableism. If your organization aims to serve "everyone" in a particular demographic, geographic area, or other category of user, you need to consider accessibility, now.

Accessibility should not be a future goal. Start now. Aim to become literate in accessibility, not an expert, and your users and products will benefit exponentially from the experiences you design and consistently improve.

Shopping Platform Key Takeaways

Accessibility is not just a technical problem, but also a cultural one.

Considering accessibility when designing and developing digital products can have a significant impact on the user experience. Designing for disability is not easy. It is important to consider the different types of disabilities when designing digital products, as different disabilities may require different accommodations.

This team integrated accessibility into the design process from the beginning, rather than treating it as an afterthought. This helps ensure that the product is accessible to all users, regardless of their abilities.

Improving Accessibility on YouTube Web

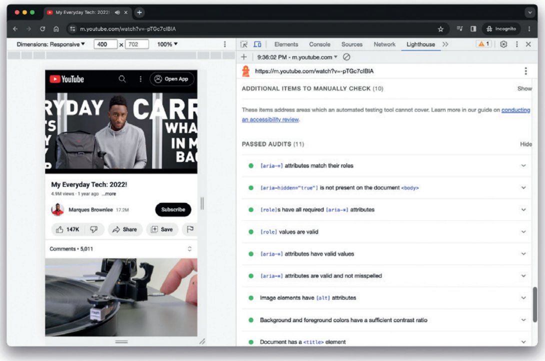
By Addy Osmani & Sriram Krishnan

In this case study, learn how the YouTube team discovered that mobile screen readers often behaved very differently than desktop screen readers, and how they addressed these problems as part of their testing.

Building an Accessible Web

Web accessibility[1] is the practice of ensuring sites are built with inclusion in mind, so that they can be used by users with disabilities. This includes users who are blind, deaf, have mobility impairments, or have cognitive disabilities.

Accessible websites use a variety of techniques to make their content accessible, such as using alternative text for images, ensuring there is enough color contrast, and making sure that keyboards are a viable way to navigate. By making sites accessible, we can ensure that everyone has equal access to information and services online.



9:36:02 PM - m.youtube.com

https://m.youtube.com/watch?v=-pTGc7clBIA

ADDITIONAL ITEMS TO MANUALLY CHECK (10) Show

These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

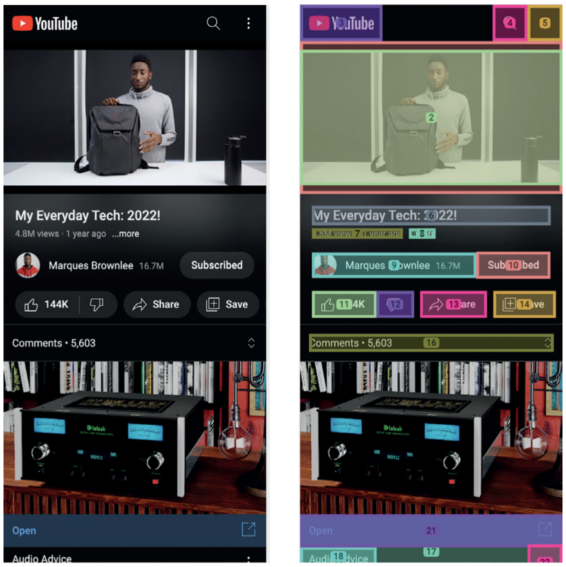
PASSED AUDITS (11) Hide

- [aria-*] attributes match their roles
- [aria-hidden="true"] is not present on the document <body>
- [role]s have all required [aria-*] attributes
- [role] values are valid
- [aria-*] attributes have valid values
- [aria-*] attributes are valid and not misspelled
- Image elements have [alt] attributes
- Background and foreground colors have a sufficient contrast ratio
- Document has a <title> element

1 https://smashed.by/learnaccessibility

Developers can use Lighthouse in Chrome DevTools for auditing a number of accessibility opportunities. Of course, testing a real screen reader environment would be even better.

YouTube web provides accessibility features[2] so users with visual disabilities can experience YouTube with screen readers. Support for reading titles, accessing player controls, and settings is available on both YouTube desktop web and mobile web properties.



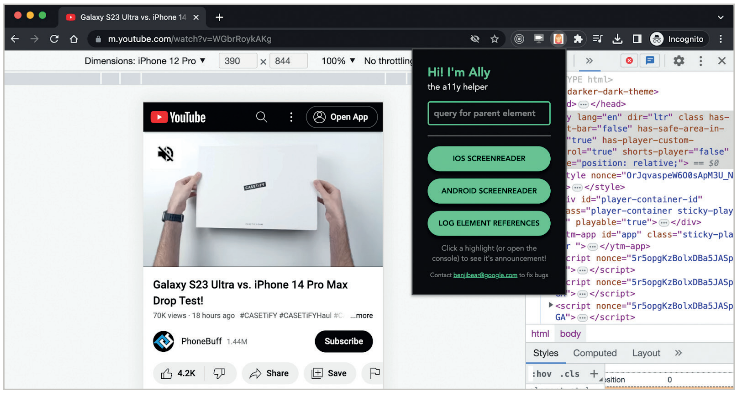
A YouTube mobile page and how it looks with current visualization tools

Issues on Mobile Web

The YouTube team had followed accessibility best practices, such as providing alt text for images[3] and using accessible color contrast combinations[4] on web pages. However, in early 2020, the YouTube team discovered a number of common issues related to screen readers on mobile web.

2 https://smashed.by/youtubescreenreaders
3 https://smashed.by/alttext
4 https://smashed.by/contrast

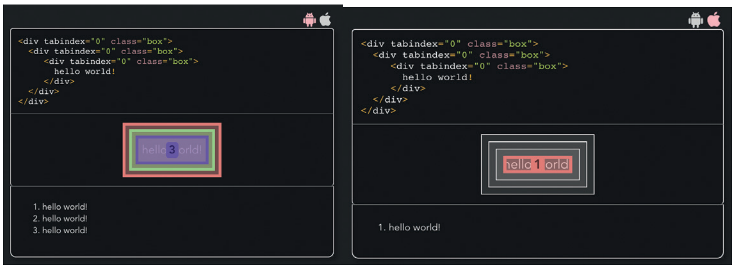
They soon realized that testing with desktop screen readers was insufficient because the screen reader behavior is different on desktop web and on various mobile devices.


Galaxy S23 Ultra vs. iPhone 14
m.youtube.com/watch?v=WGbrRoykAKg
Incognito
Dimensions: iPhone 12 Pro
390 × 844
100%
No throttling
YouTube
Open App
Galaxy S23 Ultra vs. iPhone 14 Pro Max Drop Test!
70K views · 18 hours ago #CASETiFY #CASETiFYHaul ...more
PhoneBuff 1.44M
Subscribe
4.2K
Share
Save
Hi! I'm Ally
the a11y helper
query for parent element
IOS SCREENREADER
ANDROID SCREENREADER
LOG ELEMENT REFERENCES
Click a highlight (or open the console) to see it's announcement!
Contact benjibear@google.com to fix bugs
html body
Styles Computed Layout
:hov .cls

Testing for A11Y on YouTube

The first issue they discovered was that Android and iOS mobile screen readers behaved differently for the same piece of HTML.

For example, one screen reader would allow the user to swipe to two DOM nodes; but for another screen reader, the user could swipe to three. To make things even more complicated, when both screen readers had the same elements getting focus, they would almost always have different announcements.



```
<div tabindex="0" class="box">
  <div tabindex="0" class="box">
    <div tabindex="0" class="box">
      hello world!
    </div>
  </div>
</div>
```

1. hello world!
2. hello world!
3. hello world!

```
<div tabindex="0" class="box">
  <div tabindex="0" class="box">
    <div tabindex="0" class="box">
      hello world!
    </div>
  </div>
</div>
```

1. hello world!

An example of the same DOM structure focusing very differently on Android and iOS screen readers

Common Roadblocks to Testing Screen Readers for Mobile Web

To prevent issues in production, they had to be caught when testing. However, there were several roadblocks to simulating mobile screen reader behavior to find and fix accessibility issues.

- **No current standard for mobile web screen readers**
  Existing standards are based on older device specifications and are often outdated. For this reason, the team made recommendations based on intuition, which wasn't natural for screen readers. As a result, the same DOM may have different behaviors between screen readers.

- **Mobile screen readers are unavailable in the development environment**
  Unlike desktop, where you have access to all your screen readers, you are required to port the code to mobile devices for testing.

- **Getting development code on real devices is tricky**
  For large companies like YouTube, the release process is locked for only trusted devices and getting development code on real devices can be slow. Emulators, like Chrome DevTools, can speed up testing and help catch 90% of the issues in the development environment. However, to catch all issues, consumer apps such as YouTube must be tested on untrusted devices.

- **Testing for regression after every change is difficult**
  It was really easy to cause a regression with one line change because screen reader behavior is invisible. This made it difficult for the team to verify the existing behavior and prevent regressions.

The Solution

Since the challenge was in the testing process, the team conducted research on how screen readers behaved with common

HTML patterns and documented where behavior differed between screen readers.

That knowledge was then transferred to a JavaScript library to predict what gets focused and what gets announced for each screen reader. This was integrated into our testing framework, a Chrome extension, internal education materials, and a pre-submit check to help the team speed up their workflow.

Finally, to make accessibility issues as visible to engineers as traditional UX bugs, tests are purposely very colorful, to highlight what gets focus on the page.

Takeaways

The lessons learned from YouTube's accessibility improvements can be applied to other web development projects. Here are some key best practices that were instrumental in enhancing YouTube's web accessibility, with a particular focus on manual testing.

UNDERSTAND THE LIMITATIONS OF AUTOMATED TESTING

Automated testing can catch many accessibility issues, but not all. For example, while automated tools can detect if image alternative text exists, they can't verify if the text is accurate and properly assigned. Similarly, automated tools can identify keyboard-focusable elements, but they can't determine if the focus order makes logical sense or if the focus indicator is visible. Learn more about the limitations of automated testing[5] on web.dev.

> Automated testing can catch many accessibility issues, but not all.

5 https://smashed.by/testmanual

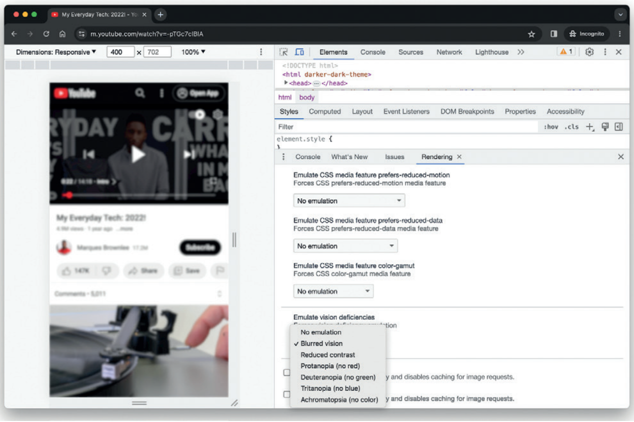
PERFORM KEYBOARD CHECKS: KEYBOARD FUNCTIONALITY IS CRUCIAL FOR ACCESSIBILITY

Test your website using only your keyboard to ensure all functionality is accessible without a mouse. Check if the tabbing order is logical and intuitive, if the keyboard focus indicator is always visible, and if focus is managed correctly when interacting with different elements. Note that keyboard checks may not always map to gestures such as swipe on mobile devices. For example, tabbing through a page will be very different from swipes on a platform like iOS.

Learn more about managing focus[6] in the Learn Accessibility course on web.dev.

CONDUCT VISUAL CHECKS

Visual checks can help identify color contrast issues that automated tools might miss, such as text on top of a gradient or image. They can also help ensure that elements that look like headings, lists, and other structural elements are coded as such, and that navigation links and form inputs are consistent throughout the website or app.



Emulate CSS media feature prefers-reduced-motion
Forces CSS prefers-reduced-motion media feature
No emulation

Emulate CSS media feature prefers-reduced-data
Forces CSS prefers-reduced-data media feature
No emulation

Emulate CSS media feature color-gamut
Forces CSS color-gamut media feature
No emulation

Emulate vision deficiencies
No emulation
✓ Blurred vision
Reduced contrast
Protanopia (no red)
Deuteranopia (no green)
Tritanopia (no blue)
Achromatopsia (no color)

Developers can also emulate vision deficiencies[7] in Chrome DevTools (such as low-contrast and blurred vision) and test features like `prefers-reduced-motion`.

6 https://smashed.by/learningfocus
7 https://smashed.by/visionemulator

PERFORM CONTENT CHECKS

Content checks can help ensure that page titles, headings, and form labels are clear and descriptive, that image alternatives are concise, accurate, and useful, and that color is not the only means of conveying meaning or information. They can also help identify issues with link text, language changes within a page, and the use of plain language.

TEST ON DIFFERENT DEVICES

Screen reader behavior can vary between desktop and mobile devices, and even between different mobile devices. Always test your website on a variety of devices to ensure it is accessible to all users.

ADDRESS SCREEN READER FOCUS FLOW AND ANNOUNCEMENTS

It's crucial to ensure that screen readers are focusing on the right elements and providing appropriate announcements. This can be particularly challenging to test, as behavior can vary between different screen readers and devices.

Note that teams should ideally align on standards for what screen readers should do for certain patterns, rather than simply going off of a gut feeling of what might be intuitive. This can otherwise lead to an endless cycle of bugs being filed where one person makes a fix that a different tester may later consider a bug. This is applicable for many mobile web experiences where there are fewer aligned on standards.

By following these best practices, you can make your web projects more accessible and inclusive for all users. Remember, manual testing is a vital part of the process, and while it may be more complex and time-consuming than automated testing, it can catch a higher percentage of issues.

By following these best practices, you can make your web projects more accessible and inclusive for all users.

Accessibility on YouTube in the Future

Maintaining accessibility for YouTube's mobile website is no longer seen as a painful problem for engineers. Now YouTube has tools for instant feedback, debugging, and programmatic testing. Potential issues get addressed beforehand, with pre-submit tests preventing regressions for new releases.

YouTube Key Takeaways

Use of automation and manual testing can help to improve the accessibility of web applications.

In early 2020 the YouTube team discovered a number of common issues related to screen readers on their mobile web platform. They soon realized that testing with desktop screen readers was insufficient because the screen reader behavior is different on desktop web and on various mobile devices.

They researched these differences for screen readers on different mobile devices and addressed accessibility by developing a JavaScript library to predict what gets focused and what gets announced for each screen reader. This library was integrated into their testing framework, a Chrome extension, internal education materials, and a pre-submit check. As a result, they were able to identify and fix accessibility issues more quickly and easily.

This has significantly decreased efforts in fixing accessibility issues and other Google properties, such as Maps and Merchant Center, have adopted these same tools for testing.

USER-FACING IMPACT

Google sets high accessibility standards for its products, which include ensuring experiences work well with different screen readers and assistive technology. YouTube's work to improve web accessibility helped it better adhere to more advanced levels of these standards.

We believe the changes outlined in this case study enable screen reader users to have an even more complete, seamless experience browsing YouTube on the web than in previous years and look forward to hearing any additional feedback from users.

While YouTube's accessibility isn't perfect yet, the improvements here already represent a more inclusive experience for users. YouTube, of course, continues to have opportunities to improve accessibility, and the team is excited to iterate on the improvements mentioned in this study.

097
1
2
3
1
2
3
DEVELOPER EXPERIENCE

DEVELOPER EXPERIENCE

Introduction

When starting a new project, engineering teams aim to create a project environment that developers feel good about and that motivates them to do better. Developer experience[1] (DX) is a term used to indicate how developers think and feel about their activities within their working environments. It assumes that an improvement of the developer experience positively impacts characteristics like sustained team and project performance.

DX is the sum of experiences resulting from different activities that developers perform. It includes perceptions about the following:

1. Development infrastructure consisting of tools, languages, processes, etc.

2. Feelings about work, such as respect, attachment, belonging, work-life balance, etc.

3. Value of their own contributions due to alignment of goals, feedback, etc.

Knowledge of these factors can help design a development experience so that the platform and ecosystem are more attractive to developers. Research[2] indicates that "happy software developers solve problems better." Unhappy developers[3] can result in low productivity and low code quality, thus affecting the software product being developed. Several factors[4] affect the overall developer experience. Many of them depend on individual needs, personality, and interactions with others. This section focuses on aspects related to development infrastructure and how engineering tools and processes contribute to DX.

1 https://smashed.by/dx
2 https://smashed.by/happydevs
3 https://smashed.by/unhappydevs
4 https://smashed.by/satisfaction

DX and Development Infrastructure

Development infrastructure can comprise technologies, applications, software, and content. A few examples of development tools are:

- A wide range of applications that include IDEs (VS Code), bundlers (webpack), source control software (GitHub), etc.
- Languages (TypeScript, GraphQL), frameworks (React or Next.js), and API providers (Stripe).
- Testing and performance tools (Selenium, Enzyme, Lighthouse).
- Hardware (machines or virtually hosted environments).
- Documentation, blogs, forums, and a community knowledge base.
- Team collaboration and communication tools (Slack, Teams, Zoom).

User experience measures how easy or pleasing it is to use an application. A good developer experience ensures that developers will enjoy working with development tools whose user base consists exclusively of developers. Development tools, practices, and platforms should be designed as aids that allow developers to achieve their maximum potential in terms of productivity and code quality. This becomes impossible with poorly designed tools or frameworks where developers end up spending time on Stack Overflow, Reddit, or other forums before they can get started.

A positive developer experience can boost team morale and improve overall project execution and quality. It contributes to the following aspects of a project:

1. **Velocity**: The team spends more time on activities that add value to the project. This leads to a faster implementation path.

2. **Quality**: Tools that reduce development, debugging, and testing times help teams catch more bugs and improve the quality of products.

3. **Onboarding**: New team members have a shorter learning curve, and they can get started independently without any hand-holding required.

Developer experience can mean different things for different tools. To ensure that a tool provides a good developer experience, we must understand: what do developers want from engineering infrastructure? A simple answer is: developers just want to write efficient code that is easy to test, maintain, and deploy. Let's break this down further in terms of software requirements.

Developers want:

- Languages and tech stacks they are comfortable with or find easy to learn.

- A robust, obstacle-free, easy-to-use development environment for writing code, version control, and management of builds.

- Easy-to-set-up testing infrastructure and data suitable for the application.

- Accessible support for times when they do get stuck.

Using the list above, we can define some of the desirable characteristics in a development platform to ensure a pleasing end-to-end developer experience.

1. Function
2. Stability
3. Adoption time and learning curve
4. Coding experience
5. Documentation clarity
6. Intuitiveness

Let's examine each of the above characteristics in detail and discuss how they contribute to the overall developer experience.

DX Components

FUNCTION

Probably the most fundamental characteristic of DX, function ensures that the software works and performs the set of activities that developers expect it to perform. If it does not work, there is no experience to talk about. A few examples of development tools with their core functions are:

1. An IDE should allow you to edit code, manage projects and files, debug, build, etc.

2. A BI tool should allow you to connect to databases or upload text/CSV/Excel files.

3. A language should support different constructs such as arrays, functions, events, etc.

4. An image editor should support commonly used file formats like JPEG and PNG.

These are features that developers automatically assume the tool or platform will support. All other features mean little without function. The function thus creates the first impression when it comes to developer experience.

STABILITY

Tools used by developers should be performant and reliable. Developers would find it challenging to trust frameworks requiring frequent patches. Minor issues can be dismissed as platform quirks but would deteriorate the DX over time. The instability could hinder developer performance and developer experience because:

1. Addressing issues with tools will distract developers from their core tasks and affect the focus required for efficient cognitive function.

2. They will lose trust in the tool and may not quickly identify valid issues in their code that could also be caused by a malfunctioning tool.

3. Upgrades that are not backward-compatible can break the build.

4. Slow functionality can slow down developers and lead to frustration: for instance, having to wait too long to build code after every small change.

A stable development ecosystem automatically creates an obstacle-free environment for developers where they can focus on their work.

ADOPTION TIME AND LEARNING CURVE

Developers who have already been assigned to a project may not have a lot of time to learn a new tool or language. A tool that follows many of the standard practices will be easier to adopt or learn. A shorter learning curve is beneficial for both the team and the project.

Some features that can result in a shorter adoption time and learning curve are:

1. Tools that use established standards for UI interactions, such as menus and keyboard shortcuts, are easy to adopt for first-time developer users.

2. Frameworks and libraries should be easy to install and get started.

3. In cases where registration is required, the registration process should be simple, requiring a minimum number of fields.

4. Compatibility issues with other hardware or software should be clearly highlighted with the corrective action the developer can take.

If developers adopt and learn a language or tool quickly, it will provide the feel-good factor and ensure better developer engagement.

CODING EXPERIENCE

Coding is what developers do, primarily. An excellent coding experience is thus crucial to ensure a great developer experience. Code editors that enable developers to write correct code quickly provide the best experience. Some features that contribute to the coding experience are:

1. WYSIWYG editors allow you to see the changes as you make them.

2. IntelliSense[5] features help developers focus on implementing logic without worrying about syntax.

5 https://smashed.by/intellisense

3. Integration with source control lets you check out, commit, or merge in the same window.

4. Social development environments like CodePen[6] let you run your code and share it with a larger user base or embed it in tutorials and blogs.

VS Code[7] is probably the best example of a tool that provides a fantastic coding experience and is constantly improving. It supports hundreds of languages and features such as syntax highlighting, bracket-matching, and auto-indentation and is highly customizable.

INTUITIVENESS

A product designed to be intuitive does not need an instruction manual or documentation to support it. This quality is inherent to a good user experience and developer experience. Developers deserve a simple, easy-to-use interface as much as any other user. Some areas that can be designed to be intuitive are:

- APIs should be named clearly with the parameters they need and what they return.
- Developer-facing UI should be intuitive, helpful, and easily configurable.
- Errors raised should be understandable and actionable and explain how developers can fix or avoid the error.
- APIs should be uniform. For example, if a token is expected as a parameter in all function calls, it should be consistently placed at the beginning or end of the parameter list.

6 https://codepen.io/
7 https://smashed.by/vscode

DOCUMENTATION CLARITY

Most developers rely on documentation, forums, discussions, blogs, and the community to address specific needs and use cases. Mostly they rely on a search engine to find what they are looking for. The following factors affect documentation clarity:

1. Documentation should be indexed and easy to search.
2. Issues like broken links and infinite loops should be addressed quickly. Page A may refer to Page B for a specific topic, B refers to A, and the developer's question remains unanswered.
3. Availability of use cases and getting started docs to complement the primary documentation are usually helpful.
4. Mandatory or legal documentation like terms of service or privacy policies for APIs should be reviewed for clarity and accuracy as they are known to cause DX issues.[8]

Ideal DX is where the tool is intuitive enough, so developers do not need docs to guide them. However, even for advanced users and sophisticated use cases, there will always be situations where an easily discoverable explanation can go a long way in enhancing the experience.

While these characteristics are critical to good DX, you cannot isolate and measure them. The best DX is where you never notice the tools or the technology because they just work as expected, giving you a smooth experience. Let us now look at some teams that have done this and done it well.

8 https://smashed.by/dxissues

Apideck: How to Build a Great Developer Experience

By Nick Lloyd

Nowadays,[1] developer experience (DX) is essential for software products. Akin to user experience (UX), DX instead focuses on improving the developer's journey, reducing hang-ups, and encouraging success throughout their programming experience. As software-as-a-service (SAAS) rises in importance, quality DX is no longer a choice – it is imperative in order to remain competitive. Below, we'll define what it takes to build a great developer experience. We'll consider why having quality DX is important and highlight some stellar examples of great DX in practice.

What Does DX Mean?

Developer experience (abbreviated as DX, DevX, or Dev EX) is like user experience but for software developers. To have excellent DX means that a software tool is functional, usable, well-designed, self-serviceable, and easy to navigate. This general gauge of quality can be applied to any developer-facing utility, whether it's APIs (application programming interfaces), SKKs (Software development Kits), CLIs (Command line interfaces), cloud infrastructure, or other software-as-a-service.

WHY IS QUALITY DEVELOPER EXPERIENCE IMPORTANT?

Developers are customers now. Many companies have found success in defering to the developer's judgment when selecting new tools, as opposed to relying on top-down direction. In this environment, competition is increasingly driven by the usability of the

1 The original version of this case study was published at Apideck: https://smashed.by/apideck

integration. In other words, usability matters for developers just as much as it does for users.

Furthermore, there are now more APIs than ever. Even the number of APIs within a single sector has flourished – for example, we've tracked over 25 APIs in the CRM space alone.[2] The hockey-stick growth of the API economy means that software providers must fight to survive, and battles are being won by the tools with the best developer experience, such as Twilio, Stripe, and Shopify. Improving usability directly correlates to more users and more sales.

In recent years, the bar has risen dramatically. It can also be hard to cold-call developers, meaning SaaS must be discoverable and self-service. With the advent of spec-driven development,[3] the standard for developer support materials has grown to include sleek documentation, shared libraries, testing sandboxes, and more open experiences curated to developer tastes.

In essence, APIs must supply more than a static reference – the market now expects intuitive, self-service experiences. In this new paradigm, things are more sophisticated, and to remain relevant, quality DX is no longer optional.

Tips for Building a Great Developer Experience

Now that we understand what DX is and why it has soared in importance, let's consider some specific ways API providers can build a great developer experience.

MAKE IT SELF-SERVICE

Make the entire process as self-service as possible. The API should require zero human support intervention – from onboarding to API

2 https://smashed.by/crmapis
3 https://smashed.by/openapi

key generation, integration, and ongoing maintenance. Instead, design the service to be easily discoverable and understandable by users. Increasing self-service capabilities means providing public documentation with a playground to test requests and responses quickly. Avoiding a paywall with a free tier can encourage early use.

HAVE GREAT FUNCTIONALITY

Developers integrate APIs to avoid reinventing the wheel, so the functionality had better be worth it. Beyond simply performing as expected, a quality DX supports engineers with non-reproducible, innovative functionality. As a baseline, the functionality must be stable and predictable. The API must also function as documented, as inconsistencies between docs and production can be a big deterrent. Human-readable error messages are also critical for diagnosing failed requests.

MEET SLAS AND BE TRANSPARENT

Quality DX also means high uptime and reliability. API.expert[4] does a good job showcasing the performance standards in our sector – most APIs hover around three-nines (99.9%) availability or higher, with a 200–500 ms median latency. APIs should specify a service-level agreement (SLA) with consumers, and strive to meet these benchmarks. It's also vital to monitor the service and maintain transparent status reports; this could be accomplished with a real-time status page or a decoupled uptime endpoint. When errors occur, notify the community and correct them immediately, as did Stripe in a recent outage.[5]

USE MODERN DESIGN STANDARDS

A quality DX also considers modern programming trends. For example, a SOAP web API serving XML feels antiquated. Nowadays, REST design and JSON data formatting are more standardly used. When appropriate, GraphQL can also increase usability signifi-

4 https://www.api.expert/
5 https://smashed.by/stripeoutage

cantly by combining disparate requests. Or, to cater to event-driven workflows, asynchronous webhooks[6] may be worth considering. APIs are also expected to open-source OpenAPI specifications to increase transparency and interoperability. Regardless, staying on top of API design trends[7] will help keep your service sleek, relevant, and developer-friendly.

MEET DEVELOPERS WHERE THEY ARE

Part of this developer-friendly focus is meeting engineers wherever they are – in skill-level, programming language, and preferred development tool. APIs can boost developer experience by providing easy integration for whatever programming language the engineer is most comfortable with. Accomplish this by offering code samples and libraries for popular languages, such as Python, Ruby, PHP C++, or Go, and by generating SDKs for mobile platforms like iOS, Android, or Windows. Other ideas include:

- a CLI to test requests from the command line
- a virtual testing environment
- a Run in Postman button
- a Run in Insomnia button

The point is to accommodate developer skill sets and appeal to the workflows and tools they already use.

INCLUDE ACCOUNT DASHBOARDS

APIs are not usable with reference material alone. In addition to documentation, a complete developer portal requires account management features. An API dashboard should provide ways to generate API keys, handle billing, upgrade rate limits, and transparently view

6 https://smashed.by/webhook
7 https://smashed.by/apitrends

usage. An advanced dashboard that logs all API calls and supplies metadata such as specific client usage could benefit developers as they refine their applications and perform error diagnostics.

ENFORCE DATA PRIVACY AND SECURITY COMPLIANCE

Conforming to regulations like GDPR or CCPA is now mandatory for most businesses. Without the proper data compliance and security clearance, the use of an API could stop dead in its tracks – a total buzz-kill for potential developer consumers. To understand what policies to enforce, API developers can easily compare security and compliance ratings between cloud services, companies, and subprocessors using a tool like ComplianceRank.[8] From a security perspective, APIs that only support HTTP Basic Authentication or don't use mTLS are prone to abuse. Instead, advanced API security structures adopt OAuth and JWT to share authorization scopes.

INVEST IN EXCELLENT DEVELOPER RELATIONS

A final piece of advice to instill great DX is to invest in community development. Being part of a community can help boost confidence and increase overall engagement around software. Developer advocates can host virtual seminars to engage with developers, or organize the community around shared forums, message boards, or chat, like Discord or Slack. API evangelists can elevate users by offering pragmatic advice and creating training materials. All in all, developer relations provides a path for additional support, greatly supporting the developer journey and improving DX.

Being part of a community can help boost confidence and increase overall engagement around software.

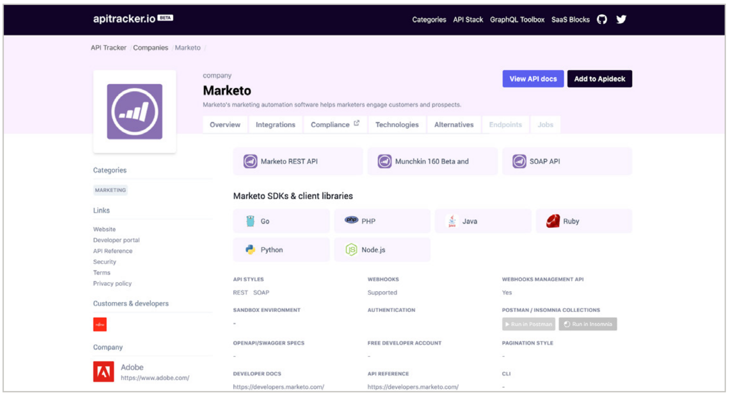
8 https://compliancerank.com/

Five Examples of High-Quality Developer Experience

The best way to demonstrate high-quality developer experience is to see it in practice. Here are five examples of great DX in the API industry.

1. API TRACKER[9]

To compare API developer experiences on the market, look no further than API Tracker. API Tracker, a curated directory of over 1,200 APIs, is an excellent way to discover APIs and see how they compare to others. API Tracker scores each API's developer experience by showcasing supported SDKs and client libraries, API styles, compliance statuses, the presence of a free account, and other pertinent integration information.



company
Marketo
Marketo's marketing automation software helps marketers engage customers and prospects.

Marketo REST API | Munchkin 160 Beta and | SOAP API

Categories
MARKETING

Links
Website
Developer portal
API Reference
Security
Terms
Privacy policy

Customers & developers

Company
Adobe
https://www.adobe.com/

Marketo SDKs & client libraries
Go | PHP | Java | Ruby | Python | Node.js

API STYLES	WEBHOOKS	WEBHOOKS MANAGEMENT API
REST SOAP	Supported	Yes
SANDBOX ENVIRONMENT	AUTHENTICATION	POSTMAN / INSOMNIA COLLECTIONS
-		
OPENAPI/SWAGGER SPECS	FREE DEVELOPER ACCOUNT	PAGINATION STYLE
-	-	-
DEVELOPER DOCS	API REFERENCE	CLI
https://developers.marketo.com/	https://developers.marketo.com/	-

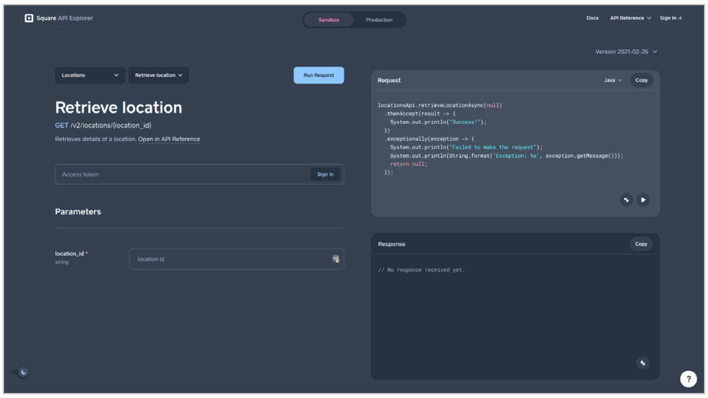
Marketo's page on the API Tracker app.

2. SQUARE API EXPLORER[10]

Square, the popular payment-processing platform, offers a truly innovative way to explore its API catalog. In addition to the typical three-columned API documentation most developers have come to anticipate, Square also created a reactive API explorer. Using drop-downs, developers can select the API, the method, and preferred language to

9 https://apitracker.io
10 https://smashed.by/square

quickly generate responses and run requests in any language. Users can insert an API key to run in sandbox or production modes. This simple tool represents a distinctive way to navigate a web API.



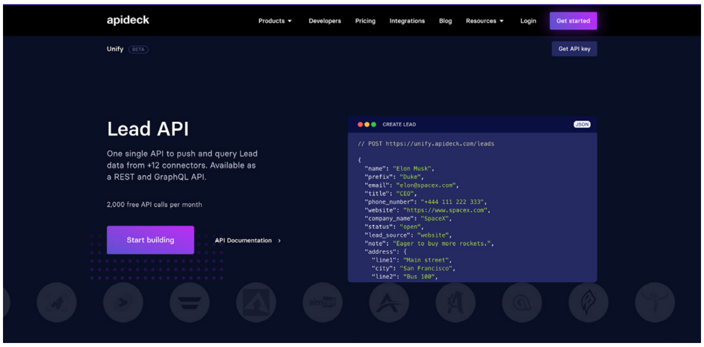
```
locationsApi.retrieveLocationAsync(null)
  .thenAccept(result -> {
    System.out.println("Success!");
  })
  .exceptionally(exception -> {
    System.out.println("Failed to make the request");
    System.out.println(String.format("Exception: %s", exception.getMessage()));
    return null;
  });
```

```
// No response received yet.
```

Using the Square API Explorer

3. UNIFIED APIs[11]

API aggregation is often necessary for industry-specific integrators – but consuming many APIs at once can be tiresome. Unified APIs decrease this burden, offering an easy way to integrate multiple APIs of the same vertical simultaneously. By reducing development overhead, Unified APIs can significantly improve developer experience. For example, the Lead API[12] offers one Unified API to push and query lead data from over 12 connectors.



Lead API

One single API to push and query Lead data from +12 connectors. Available as a REST and GraphQL API.

2,000 free API calls per month

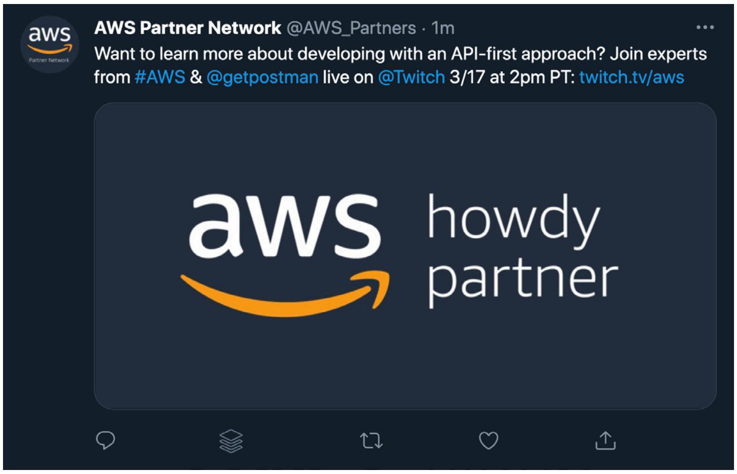
```
// POST https://unify.apideck.com/leads

{
  "name": "Elon Musk",
  "prefix": "Duke",
  "email": "elon@spacex.com",
  "title": "CEO",
  "phone_number": "+444 111 222 333",
  "website": "https://www.spacex.com",
  "company_name": "SpaceX",
  "status": "open",
  "lead_source": "website",
  "note": "Eager to buy more rockets.",
  "address": {
    "line1": "Main street",
    "city": "San Francisco",
    "line2": "Bus 100",
```

API aggregation using Lead API

11 https://smashed.by/unifiedapi
12 https://smashed.by/leadapi

4. AWS DEVELOPER COMMUNITY[13]

When it comes to building developer communities, Amazon Web Services (AWS) is unrivaled. For one, the AWS Heroes[14] program recognizes specific AWS engineers, turning customers into stars in a blog and podcast format. Many developer support options exist, including developer advocates, user groups, a dedicated AWS subreddit, and live video training. AWS also supports language-specific forums for Java, JavaScript, .NET, PHP, Python, and Ruby. As you can see, AWS isn't just talking about "meeting developers where they are" – they're actually doing it.


AWS Partner Network @AWS_Partners · 1m
Want to learn more about developing with an API-first approach? Join experts from #AWS & @getpostman live on @Twitch 3/17 at 2pm PT: twitch.tv/aws

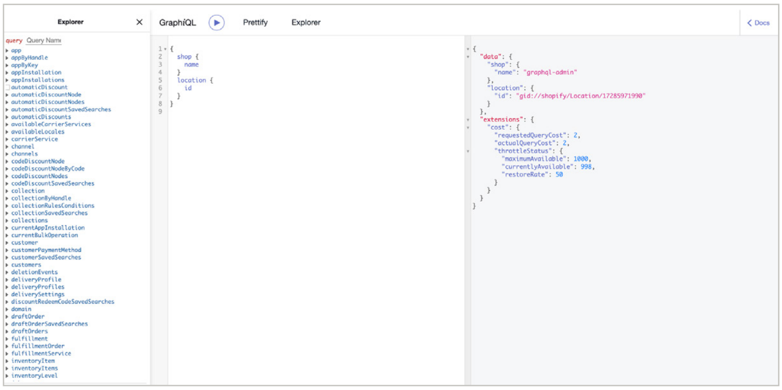
AWS Partner Network on social media

5. SHOPIFY GRAPHIQL EXPLORER[15]

Shopify is another API-first company known for its excellent developer community-building and high-quality DX. To increase usability for its Shopify Admin API, Shopify provides a GraphiQL explorer. Using the explorer on the left side, users can select what data to return, and the playground will generate a request and response in

13 https://smashed.by/awscommunity
14 https://smashed.by/awsheroes
15 https://smashed.by/graphiql

the following columns – this makes constructing GraphQL queries much easier. You can view a demo here,[16] and download this package[17] to build queries and mutations.



```
app
appByHandle
appByKey
appInstallation
appInstallations
automaticDiscount
automaticDiscountNode
automaticDiscountNodes
automaticDiscountSavedSearches
automaticDiscounts
availableCarrierServices
availableLocales
carrierService
channel
channels
codeDiscountNode
codeDiscountNodeByCode
codeDiscountNodes
codeDiscountSavedSearches
collection
collectionByHandle
collectionRulesConditions
collectionSavedSearches
collections
currentAppInstallation
currentBulkOperation
customer
customerPaymentMethod
customerSavedSearches
customers
deletionEvents
deliveryProfile
deliveryProfiles
deliverySettings
discountRedeemCodeSavedSearches
domain
draftOrder
draftOrderSavedSearches
draftOrders
fulfillment
fulfillmentOrder
fulfillmentService
inventoryItem
inventoryItems
inventoryLevel
```

```
{
  shop {
    name
  }
  location {
    id
  }
}
```

```
{
  "data": {
    "shop": {
      "name": "graphql-admin"
    },
    "location": {
      "id": "gid://shopify/Location/17285971990"
    }
  },
  "extensions": {
    "cost": {
      "requestedQueryCost": 2,
      "actualQueryCost": 2,
      "throttleStatus": {
        "maximumAvailable": 1000,
        "currentlyAvailable": 998,
        "restoreRate": 50
      }
    }
  }
}
```

Shopify's GraphiQL explorer

DX Shapes the Future of Software Development

Nowhere was quality DX needed most than within the API economy where, for years, integration was notoriously painful. Yet in recent times this has all changed. The standards have risen. Now, great DX is no longer optional – instead, it's core to the API-as-a-product movement.

As seen in the above examples, interactive demonstrations help programmers learn and get started quickly. Developers have also come to expect great functionality, high uptime, and modern design. A developer dashboard is also necessary for self-service capabilities, while developer relations can extend a helping hand to the struggling implementers. Lastly, meeting standard compliances is needed to avoid a broken experience entirely.

16 https://smashed.by/admingraphiql
17 https://smashed.by/graphiqlinstaller

Optimizing DX is an art. For powerful software tools, balancing complexity and usability is challenging. Simultaneously, developer love is hard to generate. Though not mentioned above, it probably goes without saying that APIs should avoid empty sales dialogue, as it could do more harm than good. Instead, show the power of the utility and get out of the way.

With more and more products targeting the developer mindshare, the market is highly competitive. The tools that prosper will surely be the ones that invest in developer experience and reimagine it in unforeseen ways. Hopefully, then, only the software with great DX will rule the future.

Apideck Key Takeaways

Implementing robust development tools and fostering a feedback-centric culture significantly enhances the developer experience.

- Solid onboarding processes and comprehensive documentation smooth the learning curve for new developers.
- Automated testing and continuous integration/continuous deployment (CI/CD) pipelines allow developers to catch and correct errors sooner.
- A culture that values feedback and iteration leads to continual improvement of processes and tools.
- Providing developers with the tools that allow them to easily reproduce and diagnose problems minimizes frustration.
- Regularly soliciting and acting on feedback from developers helps to identify and rectify areas of friction.

Deploying New Tech for Facebook.com

By Ashley Watkins & Royi Hagigi

Facebook.com launched in 2004 as a simple server-rendered PHP website.[1] Over time, we've added layer upon layer of new technology to deliver more interactive features. Each of these new features and technologies incrementally slowed the site down and made it harder to maintain. This made it harder to introduce new experiences. Features like dark mode and saving your place in news feed had no straightforward technical implementation. We needed to take a step back to rethink our architecture.

When we thought about how we would build a new web app – one designed for today's browsers, with the features people expect from Facebook – we realized that our existing tech stack wasn't able to support the app-like feel and performance we needed. A complete re-write is extremely rare, but in this case, since so much has changed on the web over the course of the past decade, we knew it was the only way we'd be able to achieve our goals for performance and sustainable future growth. Today, we're sharing the lessons we've learned while rearchitecting Facebook.com, using React (a declarative JavaScript library for building user interfaces) and Relay[2] (a GraphQL client for React).

Getting Started

We knew we wanted Facebook.com to start up fast, respond fast, and provide a highly interactive experience. Although a server-driven

1 The original version of this article was published in May 2020: https://smashed.by/fbtechstack
2 https://relay.dev/

app could deliver a fast startup time, we weren't convinced we could make it as interactive and delightful as a client-driven app. However, we believed we could build a client-driven app with a competitively fast startup time.

But starting from the ground-up with a client-first app brought a new set of problems. We needed to rebuild the tech stack quickly while also addressing speed and other user experience issues – and we needed to do it in such a way that it would be sustainable for years to come.

Throughout the process, we anchored our work around two technical mantras:

1. **As little as possible, as early as possible**. We should deliver only the resources we need, and we should strive to have them arrive right before we need them.

2. **Engineering experience in service of user experience**. The end goal of our development is all about the people using our website. As we think about the UX challenges on our site, we can adapt the experience to guide engineers to do the right thing by default.

We applied these same principles to improve four main elements of the site: CSS, JavaScript, data, and navigation.

Rethinking CSS to Unlock New Capabilities

First, we reduced the CSS on the homepage by 80% by changing how we write and build our styles. On the new site, the CSS we write is different from what gets sent to the browser.

While we write familiar CSS-like JavaScript in the same files as our components, a build tool splits these styles into separate, optimized bundles. As a result, the new site ships less CSS, supports dark mode and dynamic font sizes for accessibility, and has improved image rendering performance – all while making it easier for engineers to work with.

GENERATING ATOMIC CSS TO REDUCE HOMEPAGE CSS BY 80%

On our old site, we were loading more than 400 KB of compressed CSS (2 MB uncompressed) when loading the homepage, but only 10% of that was actually used for the initial render. We didn't start out with that much CSS; it just grew over time and rarely decreased. This happened in part because every new feature meant adding new CSS.

We addressed this by generating atomic CSS at build time. Atomic CSS has a logarithmic growth curve because it's proportional to the number of unique style declarations rather than to the number of styles and features we write. This lets us combine the generated atomic CSS from across our site into a single, small, shared style sheet. As a result, the new homepage downloads less than 20% of the CSS the old site downloaded.

COLOCATING STYLES TO REDUCE UNUSED CSS AND MAKE IT EASIER TO MAINTAIN

Another reason our CSS grew over time was that it was difficult to identify whether various CSS rules were still in use. Atomic CSS helps mitigate the performance impact of this, but unique styles still add unnecessary bytes, and the unused CSS in our source code adds engineering overhead. Now, we colocate our styles with our components so they can be deleted in tandem, and only split them into separate bundles at build time.

We also addressed another issue we were facing: CSS precedence depends on ordering, which is especially difficult to manage when using automated packaging that can change over time. It was previously possible for changes in one file to break the styles in another without the author realizing it. Instead, we now author styles using a familiar syntax inspired by React Native[3] styling APIs: we guarantee that the styles are applied in a stable order, and we don't support CSS descendant selectors.

CHANGING FONT SIZES FOR BETTER ACCESSIBILITY

We've taken advantage of our offline build step to make accessibility updates as well. On many websites today, people enlarge text by using their browser's zoom function. This can accidentally trigger a tablet or mobile layout or increase the size of things they didn't need to enlarge, such as images.

By using rems,[4] we can respect user-specified defaults and are able to provide controls for customizing font size without requiring changes to the style sheet. Designs, however, are usually created using CSS pixel values. Manually converting to rems adds engineering overhead and the potential for bugs, so we have our build tool do this conversion for us.

SAMPLE BUILD-TIME HANDLING

```
const styles = stylex.create({
 emphasis: {
  fontWeight: 'bold',
 },
 text: {
  fontSize: '16px',
  fontWeight: 'normal',
 },
);
```

3 https://reactnative.dev/
4 https://smashed.by/rems

```
function MyComponent(props) {
   return <span className={styles('text', props.isEmphasized
   && 'emphasis')} />;
} }
```

Example of source code

```
.c0 { font-weight: bold; }
.c1 { font-weight: normal; }
.c2 { font-size: 0.9rem; }
```

Example of generated css

```
function MyComponent(props) {
   return <span className={(props.isEmphasized ? 'c0 ' :
   'c1 ') + 'c2 '} />;
}
```

Example of generated JavaScript

CSS VARIABLES FOR THEMING (DARK MODE)

On the old site, we used to attempt to apply themes by adding a class name to the body element and then using that class name to override existing styles with rules that had a higher specificity. This approach has issues, and it no longer works with our new atomic CSS-in-JavaScript approach, so we have switched to CSS variables[5] for theming.

CSS variables are defined under a class, and when that class is applied to a DOM element, its values are applied to the styles within its DOM subtree. This lets us combine the themes into a single style sheet, meaning toggling different themes doesn't require reloading the page, different pages can have different themes without downloading additional CSS, and different products can use different themes side-by-side on the same page.

5 https://smashed.by/cssvariables

```
.light-theme {
 --card-bg: #eee;
}
.dark-theme {
 --card-bg: #111;
}
.card {
 background-color: var(--card-bg);
}
```

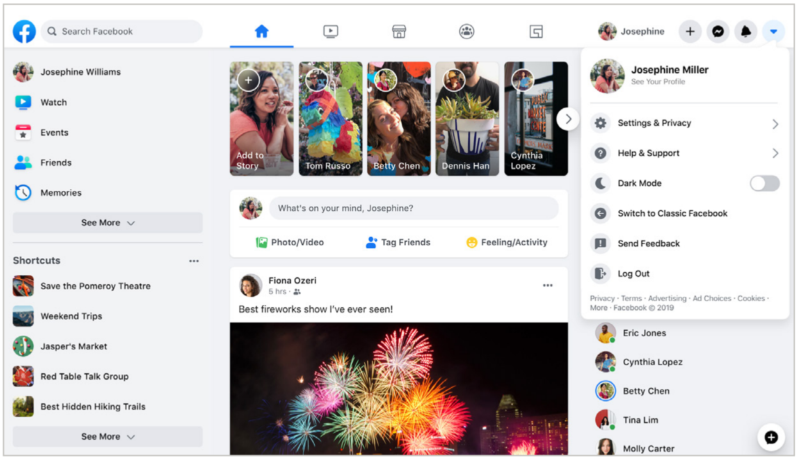
This made the performance impact of a theme proportional to the size of the color palette rather than to the size or complexity of the component library. A single atomic CSS bundle also includes the dark mode implementation.

SVGS IN JAVASCRIPT FOR FAST, SINGLE-RENDER PERFORMANCE

To prevent flickering as icons come in after the rest of the content, we inline SVGs into the HTML using React rather than passing SVG files to `<img>` tags. Because these SVGs are now effectively JavaScript, they can be bundled and delivered together with their surrounding components for a clean one-pass render. We've found that the upside of loading these at the same time as the JavaScript is greater than the cost of SVG painting performance. By inlining, there's no flickering of icons that pop in afterward.

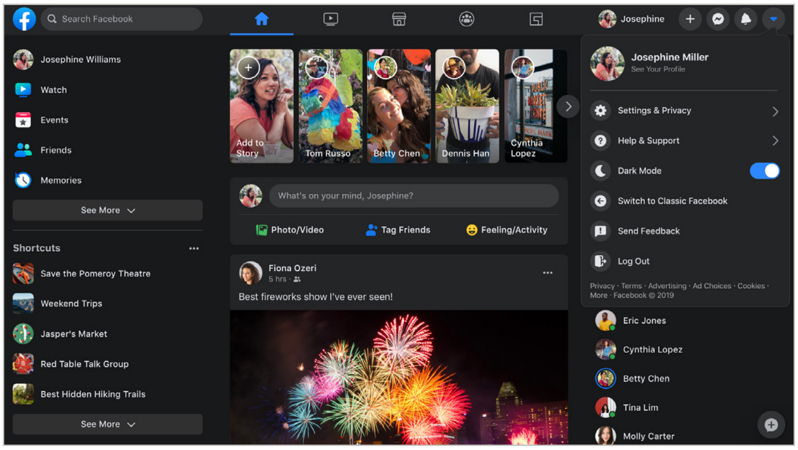
```
function MyIcon(props) {
 return (
  <svg {...props} className={styles({/*...*/})}>
   <path d="M17.5 ... 25.479Z" />
  </svg>
 );
}
```

Additionally, these icons can change colors smoothly at runtime without requiring further downloads. We're able to style the icon according to its props and use our CSS variables to theme certain types of icons, especially ones that are monochrome.



Fiona Ozeri
5 hrs
Best fireworks show I've ever seen!

SVG icons without dark mode



Fiona Ozeri
5 hrs
Best fireworks show I've ever seen!

SVG icons in dark mode

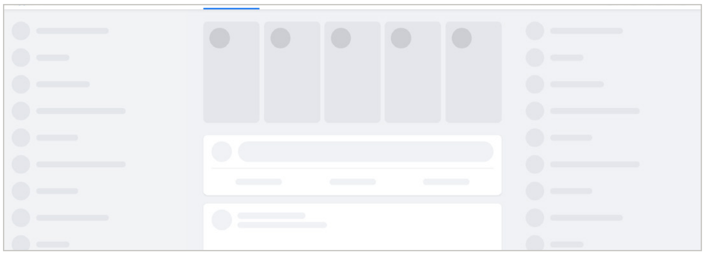
Code-Splitting JavaScript for Faster Performance

Code size is one of the biggest concerns with a JavaScript-based single-page app because it has a large influence on page load performance. We knew that if we wanted a client-side React app for Facebook.com, we'd need to solve for this. We introduced several new APIs that work in line with our "as little as possible, as early as possible" mantra.

INCREMENTAL CODE DOWNLOAD TO DELIVER JUST WHAT WE NEED, WHEN WE NEED IT

When someone is waiting for a page to load, our goal is to give immediate feedback by rendering UI "skeletons" of what the page is going to look like. This skeleton needs minimal resources, but we can't render it early if our code is packaged in a single bundle, so we need to code-split into bundles based on the order in which the page should be displayed. However, if we do this natively (that is, by using dynamic imports[6] that are fetched during render), we could hurt performance instead of helping it. This is the basis of our code-splitting design of JavaScript loading tiers: we split the JavaScript needed for the initial load into three tiers, using a declarative, statically analyzable API.

Tier 1 is the basic layout needed to display the first paint for the above-the-fold content, including UI skeletons for initial loading states.



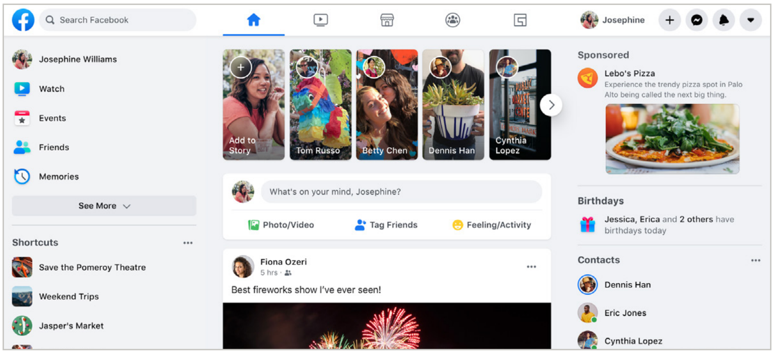
The page after tier 1 code loads and renders

6 https://smashed.by/dynamicimports

```
import ModuleA from 'ModuleA';
```

Tier 1 uses regular import syntax.

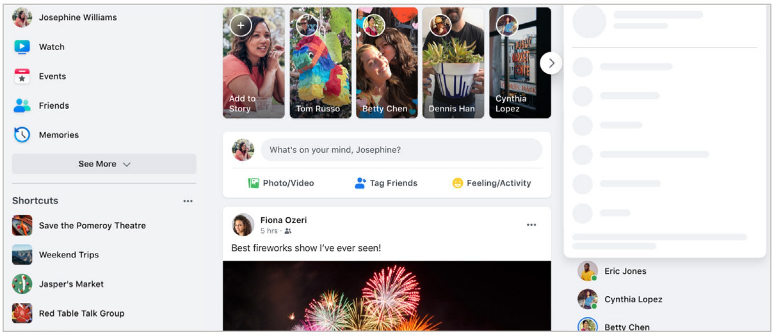
Tier 2 includes all the JavaScript needed to fully render all above-the-fold content. After tier 2, nothing on the screen should still be visually changing as a result of code loading.



The page after tier 2 code loads and renders.

```
importForDisplay ModuleBDeferred from 'ModuleB';
```

Once an `importForDisplay` is encountered, it and its dependencies are moved into tier 2. This returns a promise-based wrapper to access the module once it's loaded.



Tier 2 needs to be fully interactive. If someone clicks on a menu after tier 2 code loads and renders, they get immediate feedback about the interaction, even if the contents of the menu are not ready to render.

Tier 3 includes everything that is only needed after display that doesn't affect the current pixels on the screen, including logging code and subscriptions for live-updating data.

```
importForAfterDisplay ModuleCDeferred from 'ModuleC';
// ...
function onClick(e) {
 ModuleCDeferred.onReady(ModuleC => {
  ModuleC.log('Click happened! ', e);
 });
}
```

Once an `importForAfterDisplay` is encountered, it and its dependencies are moved into tier 3. This returns a promise-based wrapper to access the module once it's loaded.

A 500 KB JavaScript page can become 50 KB in tier 1, 150 KB in tier 2, and 300 KB in tier 3. Splitting our code this way enables us to improve time to first paint and time to visual completion by reducing the amount of code that needs to be downloaded to hit each milestone. Because tier 3 doesn't affect the pixels on the screen, it isn't really a render, and the final paint finishes earlier. Most significantly, the loading screen is able to render much earlier.

DELIVERING EXPERIMENT-DRIVEN DEPENDENCIES ONLY WHEN THEY'RE NEEDED

We often need to render two variations of the same UI; for example, in an A/B test. The simplest way to do this is to download both versions for all people, but this means we often download code that is never executed. A slightly better approach is to use dynamic imports on render, but this can be slow.

Instead, in keeping with our "as little as possible, as early as possible" mantra, we built a declarative API that alerts us to these decisions early and encodes them in our dependency graph. As the page is loading, the server is able to check the experiment and send down only the required version of the code.

```
const Composer = importCond('NewComposerExperiment', {
 true: 'NewComposer',
 false: 'OldComposer',
});
```

This works well when the conditions we split on are static across page loads for that person, such as A/B tests, locales, or device classes.

DELIVERING DATA-DRIVEN DEPENDENCIES ONLY WHEN THEY'RE NEEDED

What about code branches that are not static across page loads? For example, sending down all the rendering code for all the different types and combinations of components for news feed posts would considerably bloat the page's JavaScript size.

These dependencies are decided at runtime, based on which data is returned from the back end. This allows us to use a new feature of Relay[7] to express which rendering code is needed, depending on what type of data is returned. If the post has a special attachment, such as a photo, we describe that we need the `PhotoComponent` in order to render that photo.

```
... on Post {
 ... on PhotoPost {
  @module('PhotoComponent.js')
  Photo_data
 }
 ... on VideoPost {
  @module('VideoComponent.js')
  Video_data
 }
}
```

7 https://smashed.by/relay

We express the dependencies needed to render each post type as part of the query.

Even better, the `PhotoComponent` itself describes exactly which data on the photo attachment type it needs as a fragment, which means we can even split out the query logic.

USING JAVASCRIPT BUDGETS TO PREVENT CODE CREEP

Tiers and conditional dependencies help us deliver just the code necessary for each phase, but we also need to make sure the size of each tier stays under control over time. To manage this, we've introduced per-product JavaScript budgets.

We set budgets based on performance goals, technical constraints, and product considerations. We allocated page-level budgets and subdivide the page based on product boundaries and team boundaries. Shared infra is added to a carefully curated list and given its own budget. Shared infra counts against all pages' budgets, but modules within it are free for product teams to use. We also have budgets for code that's deferred, conditionally loaded, or loaded on interaction.

We've created additional tooling for each step of the process:

- A dependency graph tool makes it easier to understand where bytes are coming from and identify opportunities to decrease code size.
- Size monitoring on merge requests displays size regressions/ improvements and triggers customizable alerts.
- Interactive graphs show historical size and how things have changed between revisions.
- Dashboards help us understand the current state of sizes in relation to budgets.

Modernizing Data-Fetching to Get It as Early as Possible

As part of this rebuild, we modernized our data-fetching infra on the web. While some features of the old site used Relay and GraphQL for data-fetching, most fetched data ad hoc as part of their server-side PHP rendering. With the new site, we were able to standardize with our mobile apps and ensure that all data-fetching goes through GraphQL. Since Relay and GraphQL already handle the "as little as possible" work for us, we just needed to make some changes to support getting the data we needed as early as possible.

PRELOADING DATA ON THE INITIAL SERVER REQUEST TO IMPROVE STARTUP

Many web apps need to wait until all their JavaScript is downloaded and executed before fetching data from the server. With Relay, we know statically what data the page needs. This means that as soon as our server receives the request for a page, it can immediately start preparing the necessary data and download it in parallel with the required code. We stream this data with the page as it becomes available so the client can avoid additional round trips and render the final page content sooner.

STREAMING DATA FOR FEWER ROUND TRIPS AND BETTER INTERACTIVITY

On the initial load of Facebook.com, some content may initially be hidden or rendered outside of the viewport. For example, most screens fit one or two news feed posts, but we don't know in advance how many will fit. Additionally, it's very likely the user will scroll, and it would take time to fetch each story individually in a serial round trip. On the other hand, the more stories we fetch in one query, the slower that query gets, which leads to longer query times and a longer visually complete time for even the first story.

To solve this, we use an internal GraphQL extension, @stream, to stream the feed connection to the client both for initial load and subsequent pagination on scroll. This allows us to send each feed story as soon it's ready, one by one, with just a single query operation.

```
fragment HomepageData on User {
 newsFeed(first: 10) {
  edges @stream
 }
 ...AdditionalData
}
```

DEFERRING DATA THAT'S NOT NEEDED RIGHT AWAY

Different parts of certain queries take longer to compute than others. For example, when viewing a profile, it's relatively quick to fetch a person's name and profile photo, but it takes a bit longer to fetch the contents of their timeline.

To fetch both types of data with a single query, we use @defer, which enables different sections of the response to be streamed as soon as they're ready. This lets us render the bulk of the UI with initial data as quickly as possible, and render loading states for the rest. With React Suspense,[8] this is even easier, as we can explicitly craft our loading states to ensure a smooth, top-down page load experience.

```
fragment ProfileData on User {
 name
  profile_picture { ... }
  ...AdditionalData @defer
}
```

8 https://smashed.by/suspense

Route Maps and Definitions for Faster Navigation

Fast navigation is an important feature of single-page applications. When navigating to a new route, we need to fetch various code and data from the server to render the destination page. To reduce the number of network round trips required when loading a new page, the client needs to know which resources will be needed for each route ahead of time. We call this a route map and each entry a route definition.

GETTING ROUTE DEFINITIONS AS EARLY AS POSSIBLE

For Facebook, this route map is too large to send all at once. Instead, we dynamically add route definitions to the route map during the session, as new links are rendered. The route map and the router live at the very top of the application, allowing the combination of current application and router state to drive app-level state decisions, such as the behavior of the top navigation bar or chat tabs based on the current route.

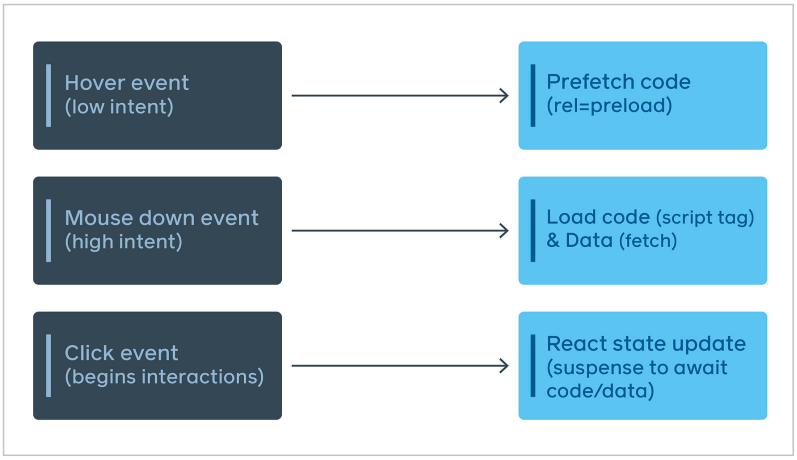
PREFETCHING RESOURCES AS EARLY AS POSSIBLE

It's common for client-side applications to wait until a page is being rendered by React to download the code and data needed for that page. Often this is done using `React.lazy` or a similar primitive. Since this can make page navigation slow, we instead start our first request for some of the necessary resources even before a link is clicked.

To provide a more fluid experience than just showing a blank screen when navigating, we use React Suspense transitions[9] to continue rendering the previous route until the next route is either fully

9 https://smashed.by/transitions

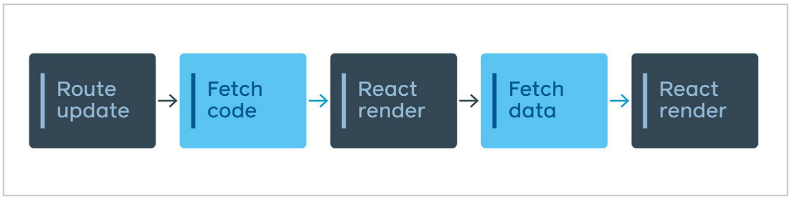
rendered or suspends into a “good” loading state with UI skeletons for the next page. This is much less jarring, and it mimics standard browser behavior.


Hover event (low intent)
Prefetch code (rel=preload)
Mouse down event (high intent)
Load code (script tag) & Data (fetch)
Click event (begins interactions)
React state update (suspense to await code/data)

We kick off fetches early, preloading on hover or focus, and fetching on mousedown. This example is specific to desktop, but other heuristics can be used for touch devices.

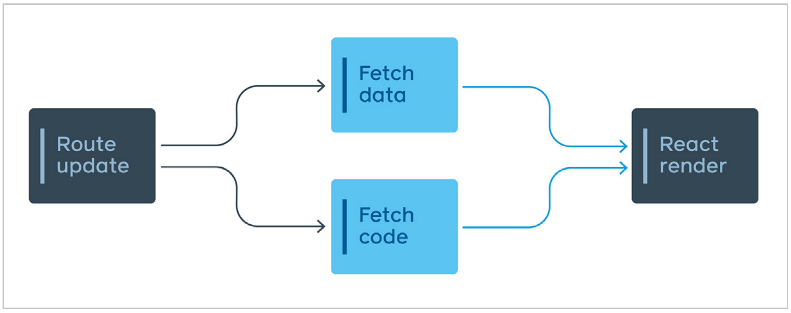
PARALLELIZING CODE AND DATA DOWNLOAD

We do a lot of lazy loading of code on the new site, but if we lazy load the code for a route and the data-fetching code for that route lives inside of that code, we end up with a serial load.



A “traditional” React/Relay app with lazy-loaded routes results in two round trips.

To solve this problem, we came up with EntryPoints, which are files that wrap a code-split point and transform inputs into queries. These files are very small and are downloaded in advance for any reachable code-split point.



Code and data are fetched in parallel, allowing us to download these in a single network round trip.

The GraphQL query is still colocated with the view, but the Entry-Point encapsulates when that query is needed and how to transform the inputs into the correct variables. The app uses these EntryPoints to automatically decide when to fetch the resources, making sure the right thing happens by default. This has the added benefit of creating a single JavaScript function that contains all the data-fetching needs for any given point in the app, which can be used for the server preloading discussed earlier.

Engineering experience improvements and user experience improvements must go hand-in-hand, and performance and accessibility cannot be viewed as a tax on shipping features.

Many of the changes we've discussed here are not specific to Facebook. These concepts and patterns can be applied to any client-side app using any framework or library. By standardizing our tech stack, we have been able to rethink how we introduce functionality that people want in a performant, sustainable way – even as we operate at engineering and product scale.

Engineering experience improvements and user experience improvements must go hand in hand, and performance and accessibility cannot be viewed as a tax on shipping features. With great APIs, tools, and automation, we can help engineers move faster and ship better, more performant code at the same time. The work done to improve performance for the new Facebook.com was extensive and we expect to share more on this work soon.

Facebook Key Takeaways

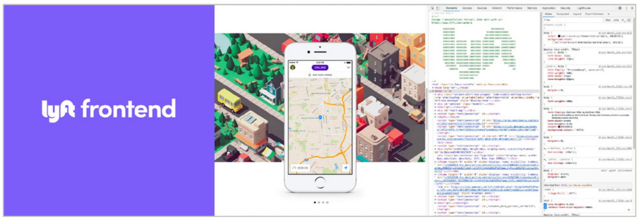
Gradual rollouts and extensive testing can make deploying new technology more manageable and less risky.

- Gradual rollouts help in managing risk when deploying new technology.
- Rigorous testing allows for catching and addressing issues before they affect end users.
- Clear communication across the team is vital during the transition process to keep everyone informed and coordinated.
- Collecting and analyzing metrics allows for data-driven decisions and iterations.
- Ensuring backward compatibility aids in smooth transitions, allowing parts of the application to fall back to the old technology if necessary.

Front End at Lyft: An Overview

By Fernando Augusto López Plascencia

At Lyft we have the enormous privilege of working with a product that impacts millions of people all around the US and Canada.[1] Most people are familiar with Lyft through our apps, which are the main way our users give and request rides, rent a vehicle, or take rides on bikes and scooters. There's also the infrastructure that allows that to happen, mainly in the form of thousands of back-end microservices. But did you know that there is a sizable portion of web front-end development as well?



While it might come as a surprise, front-end development is an integral part of what makes Lyft possible. Most of our front-end microservices are built on a TypeScript-heavy stack; these services, for example:

- Power lyft.com and affiliate websites
- Provide services to our corporate partners, allowing them to manage their fleets through the Lyft Network (such as our partnership with Hertz, which allows you to rent a car to drive with Lyft[2])
- Allow the management of internal resources through easy-to-use interfaces that add reliability to our operations

1 The original version of this article was published in March 2021: https://smashed.by/lyftfrontend
2 https://smashed.by/expressdrive

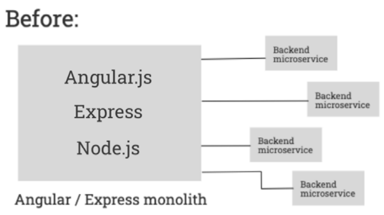
- Empower our scientists to understand and improve how Lyft runs through data tooling
- Show dashboards through custom UX/UI to support activities such as indicator monitoring, searches and decision making

All of the above need to be delivered with a high degree of quality. Front-end software development is a complex solution space, where a number of factors have to be taken into consideration, such as scalability, resiliency, consistent user experiences, heterogeneous client capabilities, and real-time experiences, with code that runs on both server and browser environments. Accomplishing these goals will provide our riders and drivers with the best transportation experiences.

Tooling and Infrastructure

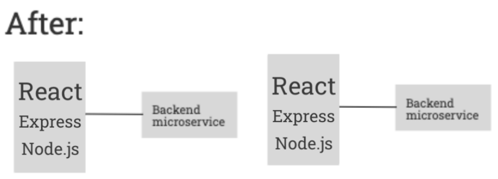
To make things happen reliably, quickly, and at the same time provide an awesome developer experience, we invested in infrastructure tooling early on.

When I joined Lyft, in mid-2016, we were no more than 25 front-end engineers. Our front-end infrastructure was simpler too, but it was already evident that it needed to become more scalable. Thus, we started migrating from an Angular-based monolithic approach for all front-end internal tools to a truly distributed network of React-on-Node services to host those tools.



Before: a front-end monolith connected to many back-end services

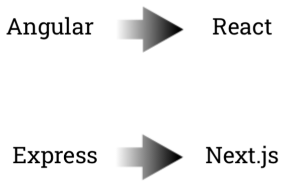
The transition from having dozens to hundreds of microservices was made possible by the use of shared, centralized libraries, such as our build system package, called Frontend Build. This allowed us to create a new service from scratch on top of Express, with preconfigured defaults for webpack, and connected to the Lyft microservice mesh through Envoy.[3] This also helped us better support recommended technologies and standards, such as Redux for state management and Jest for testing.



After: While the ratio is not necessarily 1:1, front-end services today are much more distributed.

Another shared piece of code, in this case for styling, was a set of classes that followed the atomic CSS[4] philosophy, allowing everyone to create "Lyft-looking" interfaces by composing a set of well-documented classes.

By this time, an internal portal to share React components was also created, using Lerna[5] and Storybook.[6] Each component becomes its own package in an internal namespace, which means each component is trivially easy to include in other front end projects across Lyft.



An excessively simplified view of how the front-end platform at Lyft has evolved

3 https://www.envoyproxy.io/
4 https://smashed.by/atomiccss
5 https://lerna.js.org
6 https://storybook.js.org

Frontend Build kept evolving as the front-end environment did. For example, when the time was right, we stopped endorsing Redux as a default, in favor of a React Context-heavy approach.[7] Still, developers and teams truly owned the ever-increasing number of services they created. This meant we were able to provide the best experience for each particular case, whether it was about data exploration, data visualization, or support resolution for our drivers and passengers through specific flows.

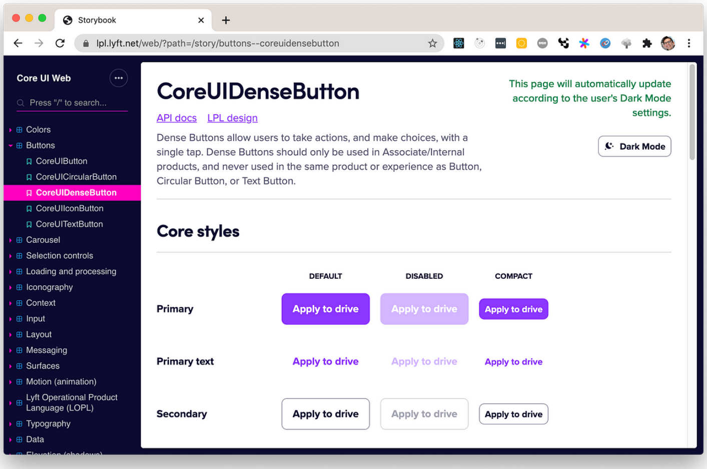
Just four years later we were almost 100 front-end developers, distributed in six offices across two continents. The distributed approach was right for Lyft, but also introduced some challenges. For example, having teams working independently on their own services meant that each service could diverge greatly from others. Due to this, we were at a point where upgrading Frontend Build was a complex task and, in some cases, a real showstopper – and not doing it could expose us to a number of issues, including potential security risks. It was time for a change, one that would still support a wide variety of use cases, streamlining upgrades, yet not leaving the developer out of the driver's seat.

The solution came in 2020 with `@lyft/service`, a new infrastructure platform based on Next.js. Adopting Next.js means the infrastructure is, in many cases, automatically upgraded, but the developers and teams still get to retain control over configuration: most Lyft-specific integrations have migrated to a plugin system, and external packages are still supported. (For more on how this migration happened, read "Changing Lanes: How Lyft is Migrating 100+ Frontend Microservices to Next.js."[8])

In the meantime, our friends in the design organization, who were facing similar challenges, created the Lyft product language (LPL) design system, which now brings visual consistency to all our exter-

7 https://smashed.by/reactcontext
8 https://smashed.by/changinglanes

nal mobile and web apps. Paired with this, in 2019 the first version of the LPL was created for front end as a library called CoreUI. This has now replaced our old, atomic CSS classes. Being a port, CoreUI makes the implementation of interfaces much easier, since it's based in the same language in which the designs were made. Also, CoreUI uses styled components for embedding CSS, which provides for much better modularity and smaller bundle sizes. Examples of what can be found in CoreUI are: typography, colors, inputs, buttons, dropdowns, icons, and layout components.


Storybook

lpl.lyft.net/web/?path=/story/buttons--coreuidensebutton

Core UI Web

Press "/" to search...

- Colors
- Buttons
  - CoreUIButton
  - CoreUICircularButton
  - CoreUIDenseButton
  - CoreUIIconButton
  - CoreUITextButton
- Carousel
- Selection controls
- Loading and processing
- Iconography
- Context
- Input
- Layout
- Messaging
- Surfaces
- Motion (animation)
- Lyft Operational Product Language (LOPL)
- Typography
- Data

CoreUIDenseButton

API docs LPL design

This page will automatically update according to the user's Dark Mode settings.

Dark Mode

Dense Buttons allow users to take actions, and make choices, with a single tap. Dense Buttons should only be used in Associate/Internal products, and never used in the same product or experience as Button, Circular Button, or Text Button.

Core styles

	DEFAULT	DISABLED	COMPACT
Primary	Apply to drive	Apply to drive	Apply to drive
Primary text	Apply to drive	Apply to drive	Apply to drive
Secondary	Apply to drive	Apply to drive	Apply to drive

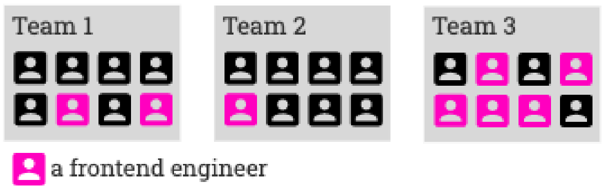
CoreUI is also hosted internally using Storyboard.

Today, almost a hundred different front-end-specific microservices exist in the Lyft ecosystem, owned by a variety of teams.

Organization and Communication

Engineering at Lyft is vertically integrated. Instead of having teams organized by work function (such as a front-end team, a back-end team, and so on), every team instead owns a portion of the Lyft product. For example, my team is called Pricing: we make ourselves responsible for providing accurate pricing for all rides, and the algorithms around that. This is a multidisciplinary effort, of which I, as a front-end developer, am also a part.

With this organization, a challenge is to effectively be in touch with other front-end developers. Collaboration needs to happen just to make things work: for example, every pull request has to be reviewed by another front-end developer. In my case, as the sole front-end developer in my team, I require to build relationships across teams just to make my work happen.



Front end members are distributed across a number of teams.

This is why, as a distributed organization, front-end at Lyft puts special emphasis on community and communication. Here are just some of the ways we achieve this:

- We have a number of organization-wide events, such as a monthly all-hands meetings, where everybody – from the most senior members to the most junior ones – are encouraged to talk about what they care about.
- We have a shared calendar where everybody that does front end can check out what events are happening.
- There's a #frontend Slack channel for general questions, plus a number of specialized channels for inquiries about particular topics. Slack is a great way to get help fast, request peer reviews, look at tech issues others have encountered in the past, and so on.

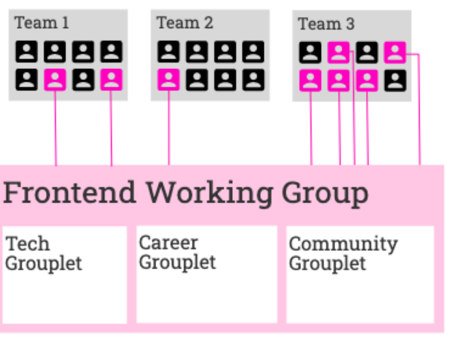
- We hold informal events as well, such as a monthly meetup called "Front End and Friends," with the sole purpose of connecting with each other.

- The front-end organization also started its own mentoring program, which greatly helped reduce siloing and allowed individuals to support each other technically and professionally. This program has recently grown to include all of engineering.

- An internal portal exists to document all of the generally accepted front-end standards at Lyft. Another one lists all of the front-end services and how they align with these standards.

- At the center of it all is our Front End Working Group, a voluntarily run organism that coordinates work to standardize and advance our technologies and internal standards, to advocate for the career and growth of front-end engineers, and to ensure we have a real community – a particularly tough challenge during the Covid pandemic.

The Front End Working Group has executive sponsors and representatives from across all of Lyft's business lines. It consists of a central group and three grouplets (tech, career and community), with each grouplet working independently on a number of work streams. The tech grouplet works on topics such as accessibility, state management, performance, and so on. The career grouplet concerns itself with topics such as: upward mobility, and education for calibrations.



All front-end members are linked through the Front End Working Group.

Finally, the community grouplet is responsible for initiatives and events that foster a sense of connectedness. Any front-end engineer at Lyft can be a part of the Front End Working Group.

Apart from the above, we have a dedicated front-end infrastructure team, which exists as part of the general infrastructure vertical and whose sole mission is to support and improve the experience of the front-end developer. The members of this team are responsible for the creation and maintenance of Frontend Build, `@lyft/service` and all of the other pieces of software around it, along with documentation, help systems and developer assistance. Having a dedicated team for this important work has been a key for having a quickly

Lyft Key Takeaways

Cohesive tooling and clear architectural guidance can drastically improve the developer experience in large codebases.

Insights from Lyft's case study are:

- A single end-to-end type system improves the developer experience by reducing inconsistencies and potential bugs.
- Clear architectural patterns make it easier for developers to understand how to contribute to the codebase.
- Coordinated tooling reduces friction and speeds up development.
- Performance should always be a consideration. Lyft used lazy loading and server-side rendering to ensure its client-side JavaScript remained fast.
- The usage of monorepos can simplify dependency management and code sharing across teams.

iterative improvement process on front-end tooling and the software development processes themselves in all of the organization.

We All Make Front-End Possible

Front-end work at Lyft is important, diverse, and exciting. While our organization has become robust during the years, we are still very flexible and anybody that contributes can have the chance to create real impact. For a couple of examples:

- A couple of front-end engineers proposed recently the internal adoption of XState as a state machine engine; this is now recognized as an optional part of our state management stack.
- A group of engineers is tackling low-hanging web performance fruit in our tech stack.
- Another group is leading an internal initiative to rebuild end-to-end testing using the Cypress framework.
- Another group of engineers is driving GraphQL adoption across the organization by building tooling and internal support.

For a couple of public examples of our work, you can check out:

- The Lyft website, with info on all things Lyft
- Our ride web page, where you can book a ride now, no app needed
- Our open-source projects with front-end components, such as Clutch, our extensible platform infrastructure manager; also, projects formerly owned by Lyft and donated to the open-source community, such as Amundsen, a data discovery and metadata engine, and Flyte, a machine learning and data processing platform.

Thanks to Andrew Hao, Jamie Cohen, Alex Ung, and Sheng Hong Tan for their collaboration in this article.

Migrating Notion's Marketing Site to Next.js

By Cory Etzkorn

One of the most challenging aspects of being an engineer[1] is knowing when to introduce abstractions and when to keep things simple. How many times have you invested time in building a scalable system upfront, planning for growth that never comes to fruition? At Notion, we're certainly no stranger to this age-old engineering dilemma.

At the end of 2020, however, we decided the time had come to invest in scalable systems that would help propel our marketing efforts into the future. Our content team had begun frequently publishing blog posts,[2] guides,[3] and spinning up various new landing pages. We were creating a large amount of content to support our rapidly growing user base, but at the same time were concerned that performance issues were preventing us from reaching our full audience.

We rebuilt our entire marketing site from scratch, choosing to go with a statically-generated architecture over our former purely client-rendered approach. Two months and 109 React components

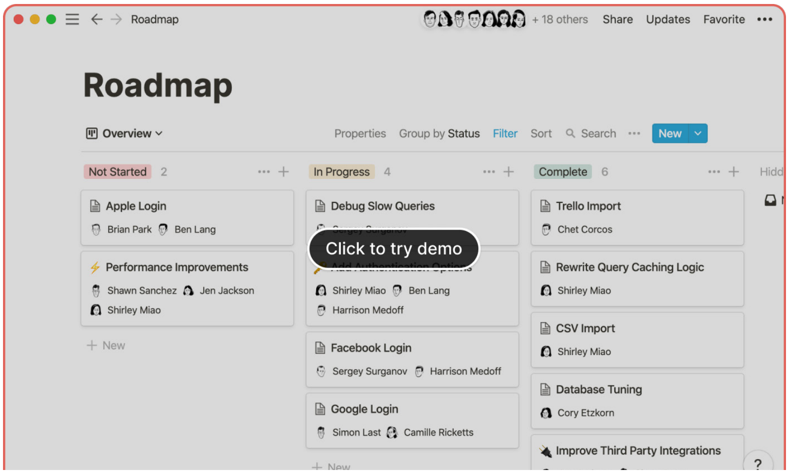
1 The original version of this case study was published in August 2021: https://smashed.by/migratingnotion
2 https://smashed.by/notionblog
3 https://smashed.by/notionguides

later, we’ve now fully migrated to our framework of choice, Next.js, and couldn’t be happier with our decision. Here’s how we got there.

Where We Started

In the early days of Notion, we chose to build our marketing site as an extension of our core app. We leveraged existing components and infrastructure so we could ship big with a small team. This approach served us well for about two years, but over time the productivity benefits we enjoyed were overshadowed by the many technical and user-facing problems associated with this approach.

The core benefit of our original client-rendered approach was developer experience. Our app and marketing site shared one large folder of React components. If we needed a popup menu on the marketing site, it was likely an existing component from the app could be reused. Shared code also enabled some interesting user experiences, like the ability to embed the full app in the marketing site as a live demo.



Sharing client app and marketing code enabled an embedded live product demo.

Eventually, the developer experience degraded. We felt like we were inheriting complexity from the app when implementing even small things on the marketing site. For example, we might import a button component from the app that looked like this:

```
<Button
  variant="marketingPrimary"
  onClick={() => soSomething()}
  mobileFeedback={() => doSomething()}
  allowTextSelection={() => doSomething()}
  onDoubleClick={() => doSomething()}
  onTouchStart={() => doSomething()}
  onTouchEnd={() => doSomething()}
  onTouchCancel={() => doSomething()}
  onContextMenu={() => doSomething()}
>
```

When all we really needed for marketing purposes was a button with a few props like this:

```
<Button
  variant="primary"
  onClick={() => doSomething()}
>
```

It got to a point where we felt our entire codebase would be easier to maintain by splitting the app and marketing site apart. Marketing teams need to be nimble and our existing setup was holding us back.

On top of the engineering problems, our implementation was causing a slew of user-facing problems. To name just a few:

- **JS bundle size**: On initial load of the marketing site, visitors were forced to download a 9.1 MB *app.js* file that contained code for the entire app. Very little of this code was marketing-related.

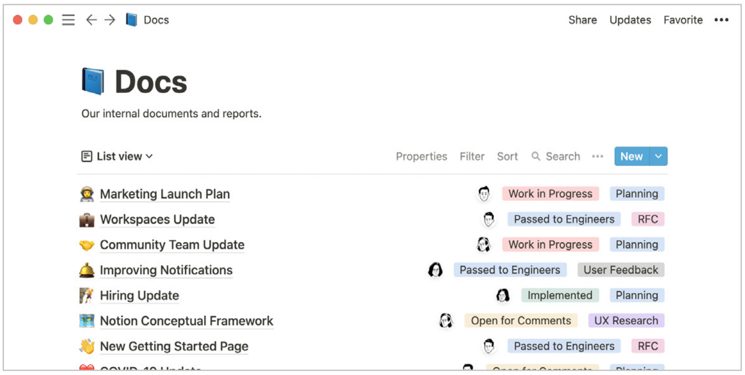
- **SEO**: Since pages were only client-rendered, crawling by search engines was dubious at best. Google has gotten better at crawling client-side JS, but nothing beats a static or server-rendered page.

- **Content management**: Without a build system, requests had to be made to our content management system's (contentful) API on every client-side visit. This resulted in millions of unneeded API calls and loading spinners on otherwise straightforward marketing pages.
- **Performance**: For the reasons above, our Google Lighthouse score for marketing pages hovered around 50/100.

The combination of these engineering and user experience issues made it clear that we needed a new, more scalable approach.

EXPLORING SOLUTIONS

Like all big decisions at Notion, our decision to integrate a static site generator began with an RFC (a "request for comment," where we ask the broader team for feedback) in our docs database.

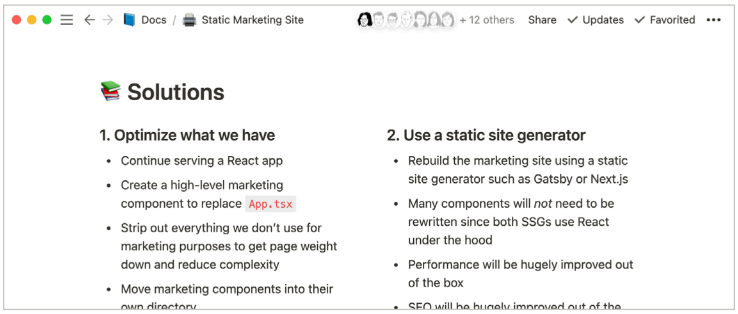


Docs

Our internal documents and reports.

Marketing Launch Plan	Work in Progress	Planning
Workspaces Update	Passed to Engineers	RFC
Community Team Update	Work in Progress	Planning
Improving Notifications	Passed to Engineers	User Feedback
Hiring Update	Implemented	Planning
Notion Conceptual Framework	Open for Comments	UX Research
New Getting Started Page	Passed to Engineers	RFC

Our company-wide docs database



Solutions

1. Optimize what we have
- Continue serving a React app
- Create a high-level marketing component to replace `App.tsx`
- Strip out everything we don't use for marketing purposes to get page weight down and reduce complexity
- Move marketing components into their own directory

2. Use a static site generator
- Rebuild the marketing site using a static site generator such as Gatsby or Next.js
- Many components will *not* need to be rewritten since both SSGs use React under the hood
- Performance will be hugely improved out of the box
- SEO will be hugely improved out of the

Part of the static site RFC

After documenting the problems we were experiencing, we were faced with two divergent paths forward.

1. OPTIMIZE EXISTING CLIENT-SIDE CODEBASE

In this scenario, we'd build on top of what we already had instead of forging an entirely new path. This work would include:

- Using code splitting to reduce marketing JS bundle size.
- Implementing better asset caching to reduce page weight.
- Creating a distinction between marketing and app components to reduce inherited complexity and risk of marketing bugs affecting the app.
- Sticking to client-side rendering in hopes performance improvements alone would improve SEO.

2. MIGRATE TO A STATIC SITE GENERATOR

This approach would mean starting from scratch so we could leverage the full benefits of a static site. The work would include:

- Rebuilding the marketing site using a JS-based static site generator such as Gatsby or Next.js.
- Migrating approximately 109 React components.
- Configuring a new build and deployment process.
- Rethinking our editorial workflows.
- Routing requests to *notion.so* between a separate client and marketing router.

No matter which path we chose, there'd be work ahead. Weighing the pros and cons, we felt the extra investment required to go fully static would pay long-term dividends for both user experience and developer productivity. It would let us be nimble.

Choosing a Static Site Generator

We kicked off the next phase of the RFC process by writing down our desires for our new static site. Seeing our needs written down helped illuminate the problems we hoped to solve. We didn't want to pick the shiniest new tool off the shelf unless it truly aligned with our goals.

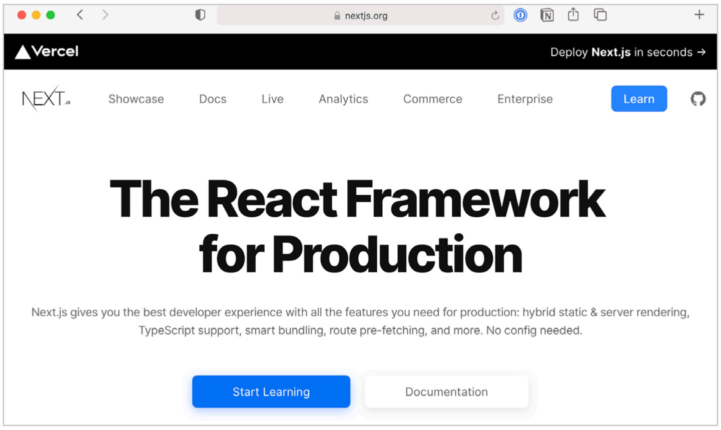
OUR STATIC SITE WISHLIST

- **React-based**: Our apps are powered by React.[4] Our marketing site should use the same technology.
- **TypeScript support**: Our entire codebase has benefited greatly from static typing. This power should be extended to the marketing site. We also have bits of code and logic that still need to be shared between the app and marketing site.
- **Contentful integration**: Our content lives here and needs to integrate seamlessly.
- **Localization**: A large percentage of Notion users reside outside of the US. Our marketing site needs to be fully localized to create a better experience for those customers.
- **Full css support**: We need the ability to use pseudo-selectors and modern techniques that cannot be expressed via inline styles.
- **Publishing workflow**: Our content creators need a way to preview their work before publishing.
- **Future-proof**: This is a big investment and we need to be confident the framework we choose is here to stay.

These parameters naturally narrowed down to just a few contenders.

4 https://smashed.by/datamodel

WHY WE PICKED NEXT.JS



The React Framework for Production

Next.js gives you the best developer experience with all the features you need for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more. No config needed.

Next.js powers our new static marketing site.

After a thorough review and proof-of-concept built in Next.js, we realized there was a lot to love:

- The framework is lightweight and declarative in nature. It handles the important things: routing, code splitting, static generation, localization, and image optimization. After that it gets out of the way.
- Full TypeScript support.
- The docs[5] and code examples[6] are excellent and left us feeling supported in the migration.
- Server-side rendering[7] wasn't something we initially needed, but we were excited to have it in our toolbox for future use.
- Internationalized routing[8] comes out of the box. A huge time saver.
- The image component[9] can integrate with Cloudflare to cache assets and improve performance.

5 https://smashed.by/nextjsdocs
6 https://smashed.by/codeexamples
7 https://smashed.by/serversiderendering
8 https://smashed.by/internationalizedrouting
9 https://smashed.by/imagecomponent

In nearly every arena, Next.js aligned with our technical goals.

Building the Static Site

Migrating our marketing site to an entirely new framework was a gigantic undertaking. It required the contributions of three engineers off and on over the course of two months. Together we migrated and refactored:

- 200k+ lines of code
- 109 React components
- 23 static pages
- 129 dynamically generated pages
- 2 locales

The migration process was smooth – which mostly involved copy/pasting code or modifying functionality to follow Next.js best practices. But there were a few areas that required extra attention and consideration.

VERSION CONTROL

Our entire codebase lives in a single monorepo. The web app, desktop apps, mobile apps – everything. We briefly considered starting a new repo specifically for the marketing site. The main benefit being that the marketing team could deploy autonomously to a static hosting platform like Vercel.[10]

Our intention was to split up the app and the marketing site, but we found separate repos unnecessary. We did end up fulfilling our dream of having a separate set of components just for marketing, but

10 https://vercel.com/

we still needed access to some shared resources. Things like analytics events, APIs, and helper methods needed to be kept in sync. Thus, we chose to stick with our monorepo and sort out deployment ourselves.

ROUTING

Adding Next.js to our stack meant we needed a way for our main client router to be aware of our new statically generated routes. Our client app and marketing site live on the same domain, making things a bit more complicated. Luckily, the solution ended up being rather straightforward.

We set up a reverse proxy that works like this:

- A request comes into notion.so.
- The API server inspects the request and parses out the path and user agent.
- We check the request's path against an allow-list of known marketing subpaths.
- If the path and user agent qualify as a marketing route, we forward the request to our marketing service.
- If the path and user agent qualify as an app route, the API server handles the request directly.

This approach allows us to maintain custom routing in the client app while also taking advantage of dynamically-generated routes in Next.js for the marketing site.

HOSTING AND DEPLOYMENT

To take advantage of some of the best Next.js features, our marketing site lives in its own Docker container and is deployed to AWS

ECS. Having a full server environment enables features like preview mode,[11] internationalized routing, and SSR.

Logging in production ended up being more challenging than expected. We created a custom server entry point[12] specifically for error handling and performance monitoring.

CSS

One of the biggest pain points of our previous marketing codebase was how we handled styles. The app primarily uses React style props. Styles are generally returned from functions:

```
class Button extends React.Component {
  // isHovered stored in component state up here
  render() {
    return (
      <button
        onMouseEnter={setIsHovered(true)}
        onMouseLeave={setIsHovered(false)}
        style={getButtonStyle(isHovered)}>
        Log in
      </button>
    )
  }
  private getButtonStyle(isHovered: boolean): cssProperties {
    return {
      height: 45,
      background: isHovered ? this.theme.buttonHoverColor :
this.theme.buttonColor,
      fontSize: 16,
      fontWeight: bold,
    }
  }
}
```

This approach works great for a complex app like Notion where a large portion of styles need to be computed at runtime, but it's less suitable for a marketing site with a more traditional publishing use case.

11 https://smashed.by/previewmode
12 https://smashed.by/preloadhack

Our marketing codebase now uses Styled JSX[13] instead of inline styles. It looks like this:

```
const Button: FunctionComponent = () => {
  const theme = useTheme()
  return (
    <>
      <button>Log in</button>
      <style jsx>{`
        button {
          height: 45px;
          background: ${theme.buttonColor};
          font-size: 16px;
          font-weight: bold;
        }
        button:hover {
          background: ${theme.buttonHoverColor};
        }
      `}</style>
    </>
  )
}
```

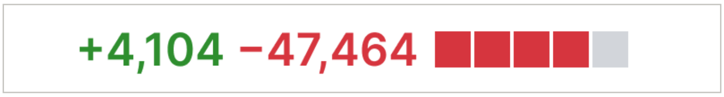
We were overwhelmed by the number of great CSS-in-JS options available and it was hard to choose just one. We've been extremely happy with Styled JSX for a few reasons:

- We get to write full, real CSS! Things like pseudo-selectors and media queries "just work."
- Styles are component-scoped by default, eliminating nasty specificity bugs.
- It works out-of-the-box in Next.js. No additional packages are needed.
- We can still interpolate values from JS as needed.

Our new approach to CSS has cut the development time of landing pages in half and has allowed us to style more expressively.

13 https://smashed.by/styledjsx

Making the Switch


+4,104 −47,464 🟥🟥🟥🟥⬜

One of our final PR diffs removing a bunch of old code

The most challenging aspect of this project was the sheer scale of it. We not only had to rebuild large portions of our site, but also maintain and update the existing site at the same time.

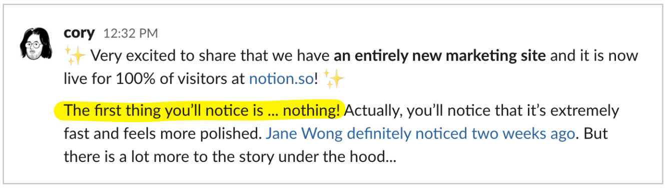
To make this possible, there were a few technical considerations:

- The new static site was developed in the same repo. Any external dependencies stayed in sync throughout the build.
- We hid the new routing logic behind an experiment, allowing us to turn on the new site in our dev environment and keep it off in production.
- When it came time to launch, we were able to ramp up traffic to the experiment over a period of two weeks to ensure everything worked as expected.

This approach made the transition completely seamless and resulted in zero downtime.

The Results

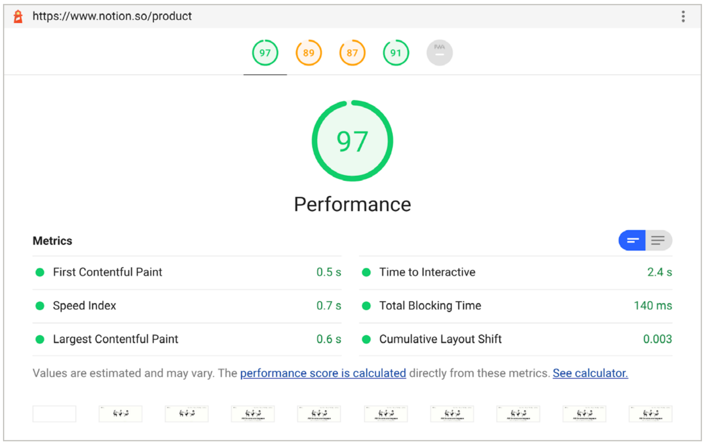
After two months of intense migration work, it was beyond exciting to share this message internally with the team on launch day:


cory 12:32 PM
✨ Very excited to share that we have **an entirely new marketing site** and it is now live for 100% of visitors at notion.so! ✨

The first thing you'll notice is ... nothing! Actually, you'll notice that it's extremely fast and feels more polished. Jane Wong definitely noticed two weeks ago. But there is a lot more to the story under the hood...

It was a funny message to share because, to the naked eye, nothing on our site had really changed. We intentionally chose to limit the scope of this project to migration and performance improvements. Adding a layer of design polish would have complicated the process and made it more challenging to measure results.

So what were the results? A whole bunch of incredible quantitative and qualitative improvements.



Metrics

● First Contentful Paint	0.5 s	● Time to Interactive	2.4 s
● Speed Index	0.7 s	● Total Blocking Time	140 ms
● Largest Contentful Paint	0.6 s	● Cumulative Layout Shift	0.003

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

Our new Google Lighthouse score for notion.so/product

Performance: We now have one of the best-performing marketing sites in the entire industry. Our previous Google Lighthouse performance score hovered around 50/100 for most pages. Our new score is 97/100. We plan to improve it even further.

User experience: There is no longer a single loading spinner on the entire marketing site. Everything is pre-rendered, cached by a CDN, and delivered instantly. Performance is a feature!

Page weight: The size of our initial required JavaScript is down 93% from 9.1 MB to 847 KB. Similar improvements are seen across the entire site. The total file size of https://notion.so/product is down 75% from 12.5 MB to 3.1 MB.

SEO: Google can now fully crawl and index our marketing pages.

Developer productivity: We can now make sweeping changes to the marketing codebase without worrying they'll cause trouble in the app. We can write full, modern CSS. Best of all, we can safely query any piece of content from our CMS, knowing that the majority of data fetching will happen at build time instead of at request.

With our new, rock-solid foundation, we're excited for the marketing team's shipping cadence to accelerate significantly. We have big things planned – and we could use a little help making them a reality.

Notion Key Takeaways

Carefully planned and executed migrations can result in significant developer experience improvements.

Takeaways from Notion's migration case study:

- Migrating to a modern framework can provide benefits in performance, developer productivity, and user experience.
- The benefits of incremental adoption allow you to slowly transition over to a new framework, reducing the risk of big-bang migrations.
- Careful planning and testing are essential to avoid disruptions during migration.
- Clear communication across the organization ensures everyone understands the process and can assist where necessary.
- Post-migration, it's important to monitor the new system closely to catch and fix any unforeseen issues promptly.

Bloomberg: 10 Insights to Adopting TypeScript at Scale

By Rob Palmer

A few years ago, Bloomberg Engineering decided to adopt TypeScript as a first-class supported language.[1] This case study shares some of the insights and lessons we learned during this journey. The headline is that we found TypeScript to be a strong net positive! Please keep that in mind when reading about some of the surprising corners we explored. As engineers, we are naturally attracted to seeing, solving, and sharing problems, even when we're having a good time.

Background

Bloomberg already had a colossal investment in JavaScript before TypeScript even existed – more than 50 million lines of JS code. Our main product is the Bloomberg Terminal, which contains more than 10,000 apps. The variety of apps is huge, ranging from the display of intensive real-time financial data and news, to interactive trading solutions, and many forms of messaging. Back in 2005, the company started migrating those apps from Fortran and C/C++ to server-side JavaScript, with client-side JavaScript arriving around 2012. Today, we have more than 2,000 software engineers at the company writing JavaScript.

Transitioning this scale of codebase from plain JavaScript to TypeScript is a big deal. So we worked hard to ensure there was a thoughtful process that would keep us aligned with standards and

1 The original version of this case study was published November 2020: https://smashed.by/typescriptatscale

preserve our existing capabilities to evolve and deploy our code quickly and safely.

The Bloomberg Terminal

If you've ever been part of a technology migration in a large company, you may be used to heavy-handed project management being used to force progress from reluctant teams who would rather be working on new features. We found that adopting TypeScript was something altogether different. Engineers were self-starting conversions and championing the process! When we launched the beta version of our TypeScript platform support, more than 200 projects opted into TypeScript in the first year alone. Zero projects went back.

What Makes This Usage of TypeScript Special?

In addition to scale, something that makes this integration of TypeScript unique is that we have our own JavaScript runtime environment. This means that, in addition to well-known JavaScript host environments, such as browsers and Node, we also embed the V8 engine and Chromium directly to create our own JavaScript platform. The upside of this situation is that we can offer a simple developer experience in which TypeScript is supported directly by our platform and package ecosystem. Ryan Dahl's Deno pursues similar ideas by putting TypeScript compilation into the runtime, whereas we keep it in tooling that is versioned independently of the runtime. An interesting consequence is that we get to explore what it's like to exercise the TypeScript compiler in a standalone JS environment that spans both client and server and that does not use Node-specific conventions (for example, there is no *node_modules* directory).

Our platform supports an internal ecosystem of packages that uses a common tooling and publishing system. This allows us to encourage and enforce best practices, such as defaulting to TypeScript's `strict` mode, as well as ensuring global invariants. For example, we guarantee that all published types are modular rather than global. It also means that engineers can focus on writing code rather than needing to figure out how to make TypeScript play nicely with a bundler or test framework. DevTools and error stacks use sourcemaps correctly. Tests can be written in TypeScript, and code coverage is accurately expressed in terms of the original TypeScript code. It just works.

> Our platform supports an internal ecosystem of packages that uses a common tooling and publishing system. This allows us to encourage and enforce best practices

We aim for regular TypeScript files to be the single source of truth for our APIs, as opposed to maintaining handwritten declaration files. This means we have a lot of code leaning heavily on the Type-Script compiler's automatic generation of `.d.ts` declaration files from TypeScript source code. So when declaration emit is not ideal, we notice it, as you will see.

Principles

Let's outline three key principles we're striving for.

1. **Scalability**: Development speed should be kept high as more packages adopt TypeScript. Time spent installing, compiling, and checking code should be minimized.

2. **Ecosystem coherence**: Packages should work together. Upgrading dependencies should be pain-free.

3. **Standards alignment**: We want to stick with standards, such as ECMAScript, and be ready for where they might go next.

The discoveries that surprised us usually came down to cases where we weren't sure if we could preserve these principles.

Ten Learning Points

1. TYPESCRIPT CAN BE JAVASCRIPT + TYPES

Over the years, the TypeScript team has actively pursued the adoption of and alignment with standard ECMAScript syntax and runtime semantics. This leaves TypeScript to concentrate on providing a layer of type syntax and type-checking semantics on top of JavaScript. The responsibilities are clearly separated: TypeScript = JavaScript + Types!

This is a wonderful model. It means that the compiler output is human-readable JavaScript, just like the programmer wrote. This makes debugging production code easy even if you don't have the original source code. It means you do not need to worry that choosing TypeScript might cut you off from future ECMAScript[2] features.

2 https://tc39.es/

It leaves the door open to runtimes, and maybe even future JavaScript engines, that can ignore the type syntax and therefore "run" TypeScript natively. A simpler developer experience is in sight!

Along the way, TypeScript was extended with a small number of features that don't quite fit this model: enum, namespace, parameter properties, and experimental decorators all have semantics that require them to be expanded into runtime code that, in all likelihood, will never be directly supported by JavaScript engines.

Standards Alignment

This is not a big deal. The TypeScript design goals[3] articulate the need to avoid introducing more runtime features in the future. One member of the TypeScript team, Orta Therox, created a meme-slide to emphasize this recognition.

Friendship ended with expression features
TS
TC 39
Now TC-39 is my best friend
enum MyThing {
namespace {

Our toolchain addresses this set of undesirable features by preventing their use. This ensures that our growing TypeScript codebase is truly JS + Types.

2. KEEPING UP WITH THE COMPILER IS WORTHWHILE

TypeScript evolves rapidly. New versions of the language introduce new type-level features, add support for JavaScript features,

3 https://smashed.by/designgoals

and improve performance and stability, as well as improve the type-checker to find more type errors. So there's a lot of enticement to use new versions!

While TypeScript strives to preserve compatibility, these type-checking improvements represent breaking changes to the build process as new errors are identified in codebases that previously appeared error-free. Upgrading TypeScript therefore requires some intervention to get these benefits.

There is another form of compatibility to consider, which is inter-project compatibility. As both JavaScript and TypeScript syntaxes evolve, declaration files need to contain new syntax.

If a library upgrades TypeScript and starts producing modern declaration files with new syntax, application projects using that library will fail to compile if their version of TypeScript does not understand that syntax. An example of new declaration syntax is the emit of getter/setter accessors in TypeScript 3.7. These are not understood by TypeScript 3.5 or earlier. This means that having an ecosystem of projects using different compiler versions is not ideal.

Ecosystem Coherence

At Bloomberg, codebases are spread across various Git repositories that use common tooling. Despite not having a monorepo, we do have a centralized registry of TypeScript projects. This allowed us to create a continuous integration (CI) job to "build the world" and verify the build-time and runtime effects of compiler upgrades on every TypeScript project.

This global checking is very powerful. We use this to assess beta and candidate releases of TypeScript to discover issues ahead of general release. Having a diverse corpus of real-world code means we also find edge cases. We use this system to guide fix-ups to projects ahead of the compiler upgrade, so that the upgrade itself is flawless. So far, this strategy has worked well and we have been able to keep the entire codebase on the latest version of TypeScript. This means we have not needed to employ mitigations such as down-levelling `.d.ts` files.

3. CONSISTENT *TSCONFIG* SETTINGS ARE WORTHWHILE

Much of the flexibility provided by *tsconfig.json* is to allow you to adapt TypeScript to your runtime platform. In an environment where all projects are targeting the same evergreen runtime, it turns out to be a hazard for each project to configure this separately.

Ecosystem Coherence

Therefore, we made our toolchain responsible for generating *tsconfig.json* at build time with "ideal" settings. For example, `strict` mode is enabled by default to increase type-safety. `isolatedModules` is enforced to ensure our code can be compiled quickly by simple transpilers that operate on a single file at a time.

A further benefit of treating *tsconfig.json* as a generated file, rather than as a source file, is that it permits higher-level tooling to flexibly link together multi-project workspaces by taking responsibility for defining options such as references and paths.

There is some tension here, as a minority of projects wanted the ability to make customizations such as switching to looser modes to reduce the migration burden. Initially we tried to cater to these requests and gave access to a small number of options. Later, we found that this resulted in interpackage conflicts, when declaration files built using one set of options were consumed by a package using different options. Here's one example.

It's possible to create a conditional type that is directed by the value of `strictNullChecks`.

```
type A = unknown extends {} ? string : number;
```

If `strictNullChecks` are enabled, then `A` is a number. If `strictNullChecks` are disabled, then `A` is a string. This code breaks if the package exporting this type is not using the same strictness settings as the package importing it.

This is a simplified example of a real-life issue we faced. As a result, we chose to deprecate the flexibility on strictness modes in favor of having consistent configurations for all projects.

4. HOW YOU SPECIFY THE LOCATION OF DEPENDENCIES MATTERS

We needed the ability to explicitly declare the location of our dependencies to TypeScript. This is because our ES module system does not rely on the Node file system convention of finding dependencies by walking up a series of directories named *node_modules*.

We needed the ability to declare a mapping of bare specifiers ("lodash," for example) to directory locations on disk (*c:\dependencies\lodash*). This is similar to the problem that import maps attempt to solve for the web. At first, we tried using the paths option in *tsconfig.json*.

```
// tsconfig.json
"paths": {
  "lodash": [ "../../dependencies/lodash" ]
}
```

This worked great for nearly all use cases. However, we discovered this degraded the quality of generated declaration files. The TypeScript compiler necessarily injects synthetic import statements into declaration files to allow for composite types – where types can depend on types from other modules. When the synthetic imports reference types in dependencies, we found the paths approach injected a relative path (`import(“../../dependencies/lodash”)`) rather than preserving the bare specifier (`import “lodash”`). For our system, the relative location of external package typings is an implementation detail that may change, so this was unacceptable.

Ecosystem Coherence

The solution we found was to use ambient modules:

```
// ambient-modules.d.ts
declare module "lodash" {
  export * from "../../dependencies/lodash";
  export default from "../../dependencies/lodash";
}
```

Ambient modules are special. TypeScript’s declaration emit preserves references to them rather than converting them to a relative path.

5. DE-DUPLICATING TYPES CAN BE IMPORTANT

App performance is critical, so we try to minimize the volume of JS that apps load at runtime. Our platform ensures that only one version of a package is used at runtime. This de-duplication of versions means that a given package cannot freeze or pin their dependencies. Consequently, this means packages must preserve compatibility over time.

We wanted to provide the same “exactly-one” guarantee for types to ensure that, for a given compilation of a project, the type check would only ever consider one single version of a package’s dependencies. In addition to compile-time efficiency, the motivation was to ensure the type-checked world better reflects the runtime world. We specifically wanted to avoid staleness issues and “nominality hell,” in which two incompatible versions of nominal types are imported via a diamond pattern. This is a hazard that will likely grow as ecosystem adoption of nominal types increases.

Scalability and Ecosystem Coherence

We wrote a deterministic resolver that selects exactly one version of each dependency to type against based on the declared version constraints of the package being built.

This means the graph of type dependencies is dynamically assembled – it is not frozen. While this unpinned dependency approach provides benefits and avoids some hazards, we later learned that it can introduce a different hazard due to subtle behavior in the TypeScript compiler. (See item 9 on this list to learn more.)

These trade-offs and choices are not specific to our platform. They apply equally to anyone publishing to DefinitelyTyped/npm, and are determined by the aggregate effect of each package's version constraints expressed in *package.json* `dependencies`.

6. IMPLICIT TYPE DEPENDENCIES SHOULD BE AVOIDED

It's easy to introduce global types in TypeScript. It's even easier to depend on global types. If left unchecked, this means it is possible for hidden coupling to occur between distant packages. The TypeScript handbook calls this out as being "somewhat dangerous."

Scalability and Ecosystem Coherence

```
// A declaration that injects global types
declare global {
  interface String {
    fancyFormat(opts?: StringFormatOptions): string;
  }
}

// Somewhere in a file far, far away...
String.fancyFormat();  // no error!
```

The solution to this is well known: prefer explicit dependencies over global state. TypeScript has provided support for ECMAScript `import` and `export` statements for a long time, which achieve this goal.

The only remaining need was to prevent accidental creation of global types. Thankfully, it is possible to statically detect each of the cases where TypeScript permits the introduction of global types. So, we were able to update our toolchain to detect and error in the cases where these are used. This means we can safely rely on the fact that importing a package's types comes without side effects.

7. DECLARATION FILES HAVE THREE EXPORT MODES

Not all declaration files are equal. A declaration file operates in one of three modes,[4] depending on the content, specifically the use of import and export keywords.

1. **Global**: A declaration file with no usage of import or export will be considered to be global. Top-level declarations are globally exported.

2. **Module**: A declaration file with at least one export declaration will be considered to be a module. Only the export declarations are exported and no globals are defined.

3. **Implicit exports**: A declaration file that has no export declarations, but does use import will trigger defined yet undocumented behavior. This means that top-level declarations are treated as named export declarations and no globals are defined.

We do not use the first mode. Our toolchain prevents global declaration files (see section 6). This means all declaration files use ES module syntax.

4 https://smashed.by/modes

Scalability, Ecosystem Coherence, and Standards Alignment
Perhaps surprisingly, we found the slightly spooky third mode to be useful. By adding just a single-line self-import to the top of ambient declaration files, you can prevent them from polluting the global namespace: `import {} from "./<my-own-name>";`. This one-liner made it trivial to convert third-party declarations, such as *lib.dom.d.ts*, to be modular and avoided the need to maintain a more complex fork.

The TypeScript team do not seem to love the third mode, so consider avoiding it where possible.

8. ENCAPSULATION OF PACKAGES CAN BE VIOLATED

As explained earlier (in item 5, "De-Duplicating Types Can Be Important"), our use of unpinned dependencies means it is important for our packages to preserve not only runtime compatibility, but also type compatibility over time. That's a challenge, so to make this preservation of compatibility practical, we have to really understand which types are exposed and must be constrained in this way. A first step is to explicitly differentiate between public and private modules.

Node recently gained this capability in the form of the package.json exports field.[5] This defines an encapsulation boundary by explicitly listing the files that are accessible from outside the package.

Today, TypeScript is not aware of package exports and so does not have the concept of which files within a dependency are considered public or not. This becomes a problem during declaration generation, when TypeScript synthesizes import statements to transitive types in the emitted `.d.ts` file. It is not acceptable for our `.d.ts` files to reference private files in other packages. Here's an example of it going wrong:

5 https://smashed.by/exportsfield

```
// index.ts
import boxMaker from "another-package"
export const box = boxMaker();
```

The above source can lead to tsc emitting the following undesirable declaration:

```
// index.d.ts
export const box : import("another-package/private").Box
```

This is bad because *"another-package/private"* is not part of that package's compatibility promise, so might be moved or renamed without a SemVer major bump. TypeScript today has no way of knowing it generated a fragile import.

Ecosystem Coherence

We mitigate this problem using two steps:

1. Our toolchain informs the TypeScript resolver of the intentionally public bare specifier paths that point to dependencies (e.g. "`lodash/public1`", "`lodash/public2`"). We ensure TypeScript knows about the full set of legitimate dependency entry points by silently adding type-only import statements to the bottom of the TypeScript files just before they flow into the compiler.

   ```
   // user's source code
   // injected by toolchain to assist declaration emit
   import type * as __fake_name_1 from "lodash/public1";
   import type * as __fake_name_2 from "lodash/public2";
   ```

   When generating references to inferred transitive types, TypeScript's declaration emit will prefer to use these existing namespace identifiers rather than synthesizing imports to private files.

2. Our toolchain generates errors if TypeScript generates a path to a file in a dependency that we know is private. This is analogous to the existing TypeScript errors emitted when TypeScript realizes that it is generating a potentially hazardous path to a dependency.

   ```
   error TS2742: The inferred type of '...' cannot be
   named without a reference to '...'.
   This is likely not portable. A type annotation is
   necessary.
   ```

   This informs the user to work around the issue, by explicitly annotating their exports. Or, in some cases, they need to update the dependency to publicize internal types by directly exporting them from a public package entry point.

We look forward to TypeScript gaining first-class support for entry points so that workarounds like this are unnecessary.

9. GENERATED DECLARATIONS CAN INLINE TYPES FROM DEPENDENCIES

Packages need to export *.d.ts* declarations so that users can consume them. We choose to use the TypeScript `declaration` option to generate *.d.ts* files from the original *.ts* files. While it's possible to manually write and maintain *.d.ts* sibling files alongside regular code, this is less preferable because it is a hazard to keep them synchronized.

TypeScript's declaration emit works well most of the time. One issue we found was that sometimes TypeScript will inline types from a dependency into the generated types (#37151). This means the type definition is relocated and potentially duplicated, rather than being referenced via an import statement. With structural typing, the compiler is not compelled to ensure types are referenced from one definition site – duplication of these types can be OK.

We have seen extreme cases where duplication has inflated file sizes, such as a declaration file growing from 7 KB to 700 KB. That's quite a lot of redundant code to download and parse.

Scalability

Inlining of types within a package is not an ecosystem problem, because it is not externally visible. It becomes problematic when types are inlined across package boundaries, because it couples those two specific versions together. In our unpinned package system, packages can evolve independently. This means there is a risk of type incompatibility and, in particular, a risk of type staleness.

Ecosystem Coherence

Through experimentation, we discovered potential techniques to prevent inlining of type declarations, such as:

- Prefer `interface` instead of `type` (interfaces are not inlined):
    - If an `interface` needed by a declaration is not exported, tsc will refuse to inline the type and will generate a clear error[6] `(e.g. TS4023: Exported variable has or is using name from external module but cannot be named.)`.
    - If a type needed by a generated declaration is not exported, tsc will silently inline the type.[7]
    - Nicholas Jamieson wrote an article on preferring interface over types,[8] including an ESlint rule.
- Make types nominal (nominal types such as `enum` and `class` with private members are not inlined).

6 https://smashed.by/inliningdeclarations
7 https://smashed.by/silentinlining
8 https://smashed.by/declarations

- Add type annotations to exports:
    - With no annotation we see inlining.[9]
    - With an explicit type annotation we force referencing behavior.[10]

The inlining behavior does not seem to be strictly specified. It is a side effect of the way declaration files are constructed. So the above methods may not work in future. We hope this is something that can be formalized in TypeScript. Until then we shall rely on user education to mitigate this risk.

10. GENERATED DECLARATIONS CAN CONTAIN NON-ESSENTIAL DEPENDENCIES

Consumers of TypeScript declaration files typically only care about the public type API of a package. TypeScript declaration emit generates exactly one declaration file for every TypeScript file in a project. Some of this content can be irrelevant to users and can expose private implementation details. This behavior can be surprising to newcomers to TypeScript, who expect the typings to be a representation of the public API like the handwritten typings found on DefinitelyTyped.

One example of this is generated declarations including typings for functions used only for internal testing.[11]

Scalability

Since our package system knows all the public package entry points, our tooling can crawl the graph of reachable types to identify all the types that do not need to be made public. This is dead type elimination (DTE) or, more precisely, tree-shaking. We wrote a tool to do this; it performs minimal work by only eliminating code from declaration files. It does not rewrite or relocate code – it is not a bundler. This

9 https://smashed.by/noannotation
10 https://smashed.by/explicitannotation
11 https://smashed.by/declarations

means the published declarations are an unchanged subset of the TypeScript-generated declarations.

Reducing the volume of published types has several advantages:

- It decreases the coupling to other packages (some packages do not re-export types from their dependencies).
- It aids encapsulation by preventing fully private types from leaking.
- It decreases the count and size of the published declaration files that need to be downloaded and unpacked by users.
- It decreases the volume of code the TypeScript compiler has to parse when type-checking.

The "shaking" can have a dramatic effect. We've seen several packages where more than 90% of the files and more than 90% of the lines of types can be dropped.

Some Options Have Sharp Edges

We found a few surprises in the semantics of some of the *tsconfig.json* options.

MANDATED `baseUrl` IN *TSCONFIG.JSON*

In TypeScript 4.0. if you want to use project references or paths, you are required to also specify a `baseUrl`. This has the side effect of causing all bare specifier imports to resolve relative to your project's root directory.

```
// package-a/main.ts
import "sibling"    // Will auto-complete and type-check if
`package-a/sibling.js` exists
```

The hazard is that if you want to introduce any form of `paths`, it carries the additional implication that import `"sibling"` will be undesirably interpreted by TypeScript as an import of `<project-root>/sibling.js` from inside your source directory.

Standards Alignment
To work around this, we used an unspeakable `baseUrl`. Using a null character prevents the undesirable bare auto-completions. We don't recommend you try this at home. We reported this on the TypeScript issue tracker and were thrilled to see that Andrew has solved this for TypeScript 4.1, which will enable us to say goodbye to the null character!

JSON MODULES IMPLY SYNTHETIC DEFAULT IMPORTS

If you want to use `resolveJsonModules`", you are required to also enable `useSyntheticDefaultImports` in order for TypeScript to see the JSON module as a default import. Using default imports is likely to become the way that Node and the web handle JSON modules in future.

Enabling `useSyntheticDefaultImports` has the unfortunate consequence of artificially allowing default imports from regular ES modules that do not have a default export! This is a hazard that you will only find out about when you come to run the code and it quickly falls over.

Standards Alignment
Ideally, there should be a way to import JSON modules that does not involve globally enabling synthetic defaults.

The Great Parts

It's worth calling out some of the particularly good things we've seen from TypeScript along the way from a tooling perspective.

Incremental builds have been essential. API support for incremental builds was a huge boost to us in TypeScript 3.6, allowing custom toolchains to have fast rebuilds. After we reported a performance issue when combining incremental with `noEmitOnError`, Sheetal Kamat made them even faster in TypeScript 4.0.

`isolatedModules` was vital to ensure we can perform fast stand-alone (one in, one out) transpilation. The TypeScript team fixed a bunch of issues to improve this option including:

- allowing `emitDeclaration` with `isolatedModules`
- allowing `noEmitOnError` with `isolatedModules`

Bloomberg Key Takeaways

The adoption of TypeScript can improve code quality and developer productivity, but the migration process should be well thought out.

- TypeScript provides self-documentation, catches errors at compile time, and improves developer tooling, all of which enhance the developer experience.
- Migration to TypeScript should be gradual and incremental to manage risk and limit disruption.
- Training and resources should be provided to help developers get up to speed with TypeScript.
- Build systems and continuous integration processes need to be updated to support TypeScript.
- A balance must be found in defining TypeScript configuration and rules to gain the benefits without overly burdening developers.

- clarifying that types must be exported explicitly with `isolatedModules`

Project references are the key to providing a seamless IDE experience. We leverage them greatly to make multipackage workspace-based development as slick as single-project development. Thanks to Sheetal, they are now even better and support file-less "solution-style" tsconfigs.

Type-only imports have been super useful. We use them everywhere to safely distinguish runtime imports from compile-time imports. They are essential for certain patterns using isolatedModules and allowed us to use `importsNotUsedAsValues: error` for maximum safety. Thanks to Andrew Branch for delivering this!

`useDefineForClassFields` was important for ensuring our emitted ESNext code does not get rewritten, preserving the JS + Types nature of the language. It means we can natively use class fields. Thanks to Nathan Shively-Sanders for providing this and making the migration process as smooth as possible. Feature delivery in TypeScript has been very fortuitous. Each time we realized we needed a feature, we frequently discovered it was already being delivered in the next version.

Conclusion

The end result is that TypeScript is now a first-class language for our application platform. Integrating TypeScript with yet another runtime shows that the language and compiler seem to be just as flexible as JavaScript – they can both be used pretty much anywhere. While we had to learn a lot along the way, nothing was insurmountable. When we needed support, we were pleasantly surprised at the responses from both the community and the TypeScript team themselves. A clear benefit of using shared open source technology is that when you have a problem, more often than not you find you are not alone. And when you find answers, you get the joy of sharing them.

Interview

Rob Palmer

JavaScript Infrastructure & Tooling Lead

Author of “**Bloomberg - 10 insights from Adopting TypeScript at Scale**”

What excited you or your team the most about the work in the case study?

The real motivator was the ability to connect our existing application platform and software engineers to the same technologies enjoyed by the wider open source software community – specifically TypeScript and the ecosystem of tooling and knowledge that has grown up around it. Bloomberg has always invested heavily in JavaScript and the result was a thriving set of internal JS-related technologies.

Previous approaches resulted in an inventive, yet insular, style that did not lend itself to the reuse of industry-standard technology. In this initiative, we threw away that playbook and aggressively pursued standards compliance and industry alignment. This was appreciated by our internal community of more than 2,000 JavaScript developers and helped energize the project team. It’s fun to find and fix bugs in widely used projects because you get the added bonus of knowing your contributions will have an impact beyond your own organization.

Were you surprised by the impact your work had on the overall user experience, business, team, or other metrics?

We were impressed by the level of engagement of software engineers around the company. Mostly TypeScript sells itself: its awesome IDE navigation, completions, and error checking are almost entirely a pure win. However, converting existing JavaScript code is still work that product teams need to fold into their schedules, while still continuing to deliver. The best antidote for that is grassroots developer enthusiasm – and that's exactly what we saw. Early adopters in particular not only converted their own codebases, but also promoted the benefits to others in turn. So it triggered a viral effect of adoption that scaled up the number of people who can support each other through the transition.

If you had a similar project/problem today, do you think your process/tooling/decisions would be exactly the same? Or, to put it differently, looking back now, what would you have done differently if you had a chance to make adjustments?

The JavaScript world never sleeps! We were very fortunate to begin this project at a time when TypeScript was already mature. This meant we could leverage relatively new TypeScript features like project references and incremental builds to make our large multi-project set-ups work well. We were also boosted by a historical internal decision to use the AMD module format many years ago. This made migrating to modern ES modules easy, because it was primarily just a mechanical exchange of one syntax for another.

The industry is currently undergoing a revolution in tooling triggered by the rise of browser-loadable native ES Modules as a replacement for the widely used CommonJS format. This enables leaps forward to be made in the speed of development and the transparency of debugging, so it's easy to anticipate that there will be further iterations in this space. While there's no need to always be leaping on the latest shiny things, the theory is that we best prepare ourselves for whatever wins out long-term by sticking to standards.

What do you think was the one critical decision that made the outcome successful? What brought you to this decision, and how did you or your entire team make it?

We had to make some hard technical decisions. Looking back, I believe some were critical choices. For example, ensuring the build was fully deterministic based on the source code rather than permitting it to be affected by environmental factors that can change over time. In retrospect, the most important choice was non-technical. It was to release a minimum viable product (MVP) as early as possible, first to one courageous guinea-pig customer, and then to incrementally release it to a defined set of early adopters with functionality based on their feedback.

This created a virtuous cycle and unearthed requirements and problems we could not have predicted ahead of time. This structure meant that, even if we had made mistakes, there was an inherent mechanism by which problems would be identified and fixed. It helped make success inevitable.

What came next after the case study was published?

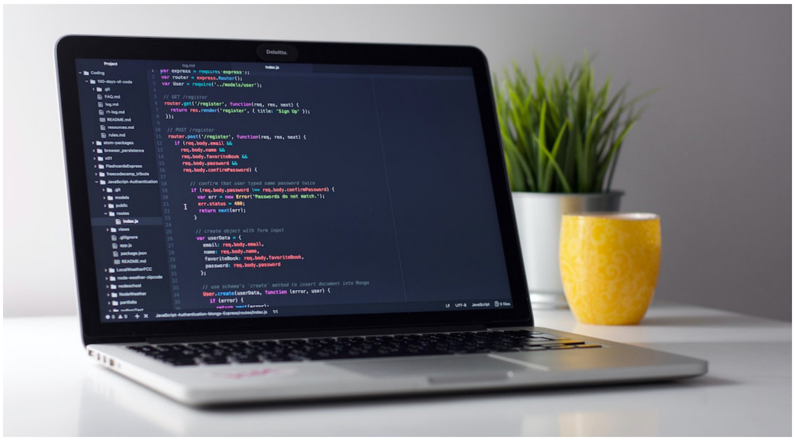
After the general availability release, we held an internal "Get Typed!" event to promote TypeScript within Bloomberg. This was heavily advertised, and we tried to make it both fun and educational. The theme was "Back To The Future," a 1980s time-travel film, to convey the sense of returning to a world of static typing that we had temporarily left behind during the transition from C++ to JavaScript many years ago.

One team created a highly produced comedy infomercial to convey the life-changing properties of static types. It managed to attract a very large audience and led to a healthy bump in the number of projects adopting TypeScript.

Zoover: Using Monorepos Is Not That Bad

By Medhat Dawoud

The decision either to use or not to use monorepos[1] has been very difficult for a lot of reasons, especially when you do some quick research and read this article titled “Monorepos: Please don’t” and then read this one titled “Monorepo: please do!” There are a lot of good reasons here and there. In this case study I’ll tell a quick story about me and my team at Zoover and why we got into the decision of adopting a monorepo in our projects, what tools we used, and what the impact was, as well as workarounds for a common issue.



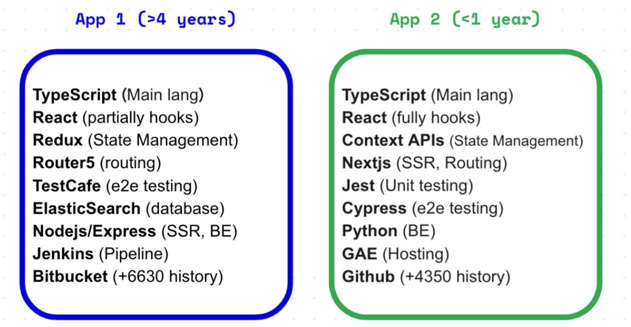
A Brief About the Problem

Let me tell you first about the situation and what pushed us to our decision in the first place.

1 The original version of this case study was published September 2021: https://smashed.by/monoreposcasestudy

I have been working for a company called Zoover since August 2019, working in the travel industry, and shortly after that building our booking system to turn into an online travel agency (OTA). COVID-19 hit hard and governments stopped travel, and – obviously – we were affected badly by that.

A few months later we were acquired by another company called Vakanties, which means we (the development team) now had two brands to support, and a lot of services and libraries were to be shared. However, the tech stacks were not really matching, and this was a problem.



App 1 (>4 years)	App 2 (<1 year)
TypeScript (Main lang)	**TypeScript** (Main lang)
React (partially hooks)	**React** (fully hooks)
Redux (State Management)	**Context APIs** (State Management)
Router5 (routing)	**Nextjs** (SSR, Routing)
TestCafe (e2e testing)	**Jest** (Unit testing)
ElasticSearch (database)	**Cypress** (e2e testing)
Nodejs/Express (SSR, BE)	**Python** (BE)
Jenkins (Pipeline)	**GAE** (Hosting)
Bitbucket (+6630 history)	**Github** (+4350 history)

As you can see, there's some matching in the tech stack but a lot of things are not easily reusable between the two websites. And since we were now a single team developing and supporting two different websites, we needed to minimize the time to create new features or fix bugs for both brands.

The core functionality of the OTA is the process of booking. On the two websites this was almost identical, and we need to share that part, at least in the beginning.

And a lot of challenges started to shine:

- Two big repositories with two large `git history` logs
- Different APIs within different pages and different third-party libraries

- Different development stacks (state, routing, and so on)
- How to manage dependencies for shared parts
- Deployment: how/when/resources
- Deciding which one is going to be merged with the other
- And much more…

The goal or key metric for having this as a successful step is to have more story points burned each sprint, and for it to be less effort for developers to create or fix and change both brands at the same time, and share services and common parts so we can use the same third-party libraries in both brands.

First Thoughts About Solutions

I conducted research with people who had had a similar problem, asking how they tried to fix it, and I found a significant amount of people talking about different ways to overcome that problem.

1. USING MONOREPOS

First, what are monorepos?
Simply, a monorepo is one large repo that contains multiple apps in a folder structure instead of a multiple repositories structure; not only apps, but also libraries, documentation, tests, build files, back end, and front end, and so on.

Who is using monorepos?
All FAANG companies and more (including X/Twitter, and Microsoft) might make it a good solution for us too! if it works for all those giant companies, most probably monorepos could work for us as well.

However, during my research, I hit some drawbacks of using monorepos that intimidated me from the beginning of the project – some very common issues that people who are against monorepos have described:

- **Git slowdown**: Codebase growth might affect the performance of simple git commands like git status.
- **Broken master**: As all apps are under the same repo then a single mistake can affect all other teams' work.
- **No autonomy**: All teams have to use the same tech stacks or, at least, have limited options to change the tooling because of the shared stuff.
- **Long build time**: Building the whole app will take too long when compared with building only an updated service or library.

Given these findings, I decided to park the idea of monorepo for now.

2. USING POLYREPOS AND DEPLOYING SHARED CODE INTO PACKAGES (NPM/GPR)

Another solution was to use multiple repositories (polyrepos) and build a small proof of concept to see how smooth it could be in sharing code, assets, and APIs between the two brands.

For the shared code, we could package it as a shared library and publish it on a private registry on NPM or GPR (GitHub Package Registry). Versioning each published library can support the two brands at the same time.

I found some very good features and also some drawbacks.

Positive:

- **Strong team ownership**: Each team owns a specific part of a separate repo, which might be useful for splitting responsibilities.
- **Fast build time**: Because the repos are separate, the build takes a short time.

- **Isolate master break**: If someone broke master, that would affect a single app or service, not others.
- **Creating multiple versions of each library**: Versioning could be helpful.

Negative:

- **Duplicate work**: Some code cannot be shared, so copy/paste.
- **Access to different repos**: Every team member should have access to each app/lib.
- **Dependency hell**: Diamond dependency problem.
- **The overhead of publishing dependencies**: On NPM or GPR.
- **Newcomers heavy setup**: Onboarding process will take longer.
- **Coding style/architecture silos**: Each team will have a different code standard and cannot enforce it for all codebases easily.

So I had to park using polyrepos too. There was one more solution to check.

3. USING GIT SUBMODULES

The third solution to fix our problems was to use Git submodules. This is basically cloning a repository into a directory inside another repository and running some git commands to make that subdirectory a submodule from the parent Git repo. You can read more here.[2] After researching the expected results and making a quick proof of concept, here is a list of issues I found:

- **No big efforts in setup**: Almost no change in the current two repositories, which could be used as they were without merging.

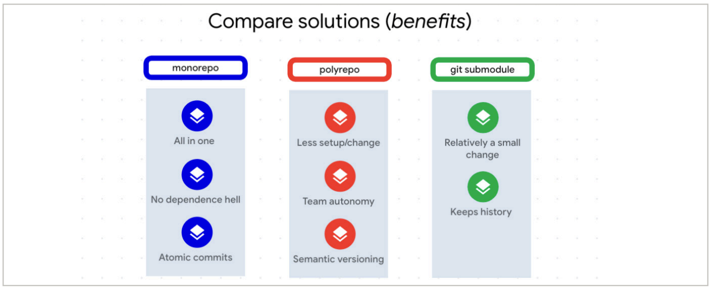
2 https://smashed.by/submodules

- **Steep learning curve**: Learning new Git commands that might be a bit more difficult is a challenge for developers who are used to the normal Git commands.

- **Switching branches**: It is a well-known issue in using Git submodules, when you switch the branch in the parent repo you have to run a command to switch it as well in the submodule, which makes it error-prone.

- **Complex to understand**: The techniques of working with submodules are a bit complex and hard to understand or explain.

Even for this solution, you have to compromise, and it was not an easy decision to adopt, especially after discussing it with the team. I then reconsidered the three solutions and compared their benefits, as all of them have their own problems.



The benefits of monorepos vs. polyrepos vs. submodules

From the comparison, we found that the monorepo solution was the one that fit our needs most – and we could work on finding ways to avoid its problems.

Raise the Monorepo Solution "Again"

This time, I needed to make sure that we were picking the right tooling and that the above-mentioned problems were as far from our team as possible; we are a relatively small team and our app is less scalable compared to the apps in the FAANG companies that have these problems.

In my journey to find a good monorepo tool, I found a lot of solutions from big companies, including: Pants by Twitter; Bazel by Google; Buck by Facebook; Rush by Microsoft; and others made especially for applications of our size like Nx, Bit, and Lerna.

NX

I started with Nx.[3] I had read very good reviews about the tool and its abilities, and I was very enthusiastic to try it out. Here are the good things about using Nx:

- **It can be used to manage projects with different stacks**: Given that we needed to refactor one app to use some new stacks in the other one, this feature might be very good for us.

- **Directed acyclic graph (DAG)**: This is a tool that comes with Nx to draw a graph out of your application dependencies and show you who is going to be affected by your changes.

- **Support is top-notch**: I needed some support and they jumped with me in a one-to-one call immediately, I'll talk about that later.

Some bad things (at this point) that I found:

- **Depending on angular CLI releases of TypeScript**: Means that when we need to update the TypeScript version in our project, we also need to update the version of Nx, which will wait for the updated version of angular CLI every six months (too long).

- **Not ready to react**: I faced many issues on the first try and although jumping on a call helped me to work around it, I have

3 https://nx.dev/

a feeling that more of these issues are coming in the future and we'll need a lot of support.

BIT

Then I decided to give Bit[4] a try. Here's what I thought:

- **Very easy to set up**: It was kind of two steps to be in the game and start sharing the code you want with your team.
- **Very organized to use and host code**: This is a great way to try to preview the code running that could make us get rid of Storybook.
- **Very expensive**: A very important factor is the cost. It costs $200 per month which is a bit too much for our needs now.

LERNA

Finally, switching to the great combination: Lerna[5] + Yarn workspaces. It is future-proof, used broadly with our stack, and I'd even used it in a React library side project, so I had some good experience with it – but no experience of usage with big projects. Here is a list of good signals about using Lerna:

- **Easy setup**: Very easy and fast to set up with a few commands
- **Guaranteed**: Proven for our case and our stack
- **Free**: We got a lot of features without paying a cent

TURBOREPO

While we faced this issue and during my research for solutions in 2020, Turborepo had not yet been released. If you don't know Turborepo, it's a new build system introduced by Jared Palmer and

4 https://bit.dev/
5 https://lerna.js.org/

acquired by Vercel[6] in December 2021. I think it is a brilliant solution that could have been a great option to pick if it had been available. Here is my take:

- **Easy setup**: It just works, plug and go, no major changes, and super clear docs.

- **Trusted creators**: It was created by learning from other solutions and by great creators powered by Vercel, which is powering Next.js[7], Turbopack[8] as a successor to webpack, and more. We have a history with the creators and the quality of software they support and the ecosystem they make around it.

- **Content awareness hashing and incremental builds**: Some of the features that caught my eye are content awareness hashing and incremental builds. Any shared content or libraries would not be rebuilt in another module if they had not changed from the last build within any module in the monorepo. Also useful is skipping the last built stuff so as not to rebuild everything every round in an incremental way.

- Based on my research I didn't find many problems in Turborepo. The only thing that people comment most about it is that it might not be clearly ready for production. However, any major problems are getting fixed super fast.

Challenges We Faced When Using a Monorepo

Everything comes with its taxes, and having a monorepo is the same: some challenges are well known in the community, and some others are specific to our case.

6 https://vercel.com/
7 https://nextjs.org/
8 https://smashed.by/turborepo

- **New tools/commands learning curve**: Whatever tool we picked has some difference from what the team used to use.
- **How to merge the two repos and retain git history**: That is a big hassle. I wrote my research and solution in an article here[9] that you can check out: exciting solution!
- **Cost of building monorepo setup**: To build it in the first place, we needed to decide on the service (GitHub, Bitbucket, and so on) and then set up scripts for running and linking projects to each other. That might be a one-time effort but it takes time.
- **What do we need to share?** Types, services, components…
- **Global types**: Any file **.d.ts* has only the scope of the project and is not shared with other apps.
- **Jest doesn't support ES modules out of the box**: That is something that I'd never faced before, and error messages were not really helping much.
- **Dependency versions**: Decide what to stop hoisting and what to hoist and the versions should be aligned in all apps.
- **React-DOM errors**: Multiple versions give a stupid hooks error, which is not easy to detect.
- **Bundle size**: Tree shaking in webpack is not straightforward.
- **Theming**: Having two brands using the same components was a new thing to the team and we needed a solution for theming.
- **Deployments**: A completely new method of deployment (sophisticated pipelines).

9 https://smashed.by/persistingcommit

- **Broken master**: It's true – any teammate can break the master branch and ruin your day.

Well, all this was difficult, but we could manage it finally with some steps and other tooling we decided to adopt.

Benefits We Gained from a Monorepo

Let's talk first about the benefits we gained by having the monorepo setup for the two brands we have and making the whole front-end projects share a lot of code.

- **Single source of truth**: Yes, a monorepo has all the code in one place, and dependencies for shared code are clear – finally.

- **Automatic linking apps and packages**: Very easy and with one command

- **Atomic commits**: This is a very important feature in working with a monorepo. Imagine that you need to change something in a shared library. If you are not using a monorepo, you will commit it to push and let someone review it and then deploy it. After it is deployed, you start making the change in the two brands, and you'd never know if the change in the library was fulfilling the needs of the two brands. So to make any modifications you need to redo the same process again and again. But using atomic commits means that changes in the shared library and for all consumers will be done in one go in the same commit. All succeed or all fail, which is very handy and saves a lot of time.

- **Diamond dependency problem fix**: No dependencies require two different versions.

- **Codebase modernization**: We can now enforce code quality across all codebases.
- **Faster feedback loops**: As we have an atomic commit, any change in any part will give us short and fast feedback about if it works or not in all clients.
- **No need for permissions**: To use different apps you need no different permissions, and you get all code at once.
- **Enforcing a workflow for the whole team**: That helps in avoiding deployment issues
- **Easier to cross-build apps**: We can now build multiple apps at once with a single command or a single commit.
- **Much easier to set up a newcomer's environment**: Only one permission, one setup. It's not complex at all.

We have managed to work around the common issues for monorepos in general:

- Git slowdown: Not the case yet, but you can use Mercurial
- Broken master: Use Git hooks (`pre-push`, `pre-commit`)
- Long build time: Split build (GitHub Actions, Lerna)
- Codebase complexity: Write more documentation and comments
- Bundle sizing: Use absolute paths and chunking
- Still enhancing every time something new appears

After all those trials and given the results we had, it is clear that using monorepos is not that bad!

Are Monorepos for You?

Some of you might be very enthusiastic now about using a monorepo in your next project – but are monorepos for you? The answer is that they're not the best for all projects or for all teams. As always, it depends.

Here is a list of situations in which using a monorepo would not be wise:

- **If you don't have a lot of shared code**: The most crucial gain of using a monorepo is sharing code easily and having atomic commits or atomic deployments for projects asynchronously. If you don't have or need that, please don't.

- **If you have some private projects/code parts**: In a monorepo, anyone can access any project's code. Some bigger companies have their own ways and tools to limit that, but most tools don't, so be careful with private projects you don't want to share in a monorepo.

- **If you don't suffer from dependency hell with polyrepos**: Using polyrepos can come with great benefits. If you have microservices, for example, in different repositories, and you have no problems with that, don't hassle yourself with monorepos, which would not bring a big benefit.

- **If you or your team are not ready for it**: It is important to have everyone on the team capable of using the tools. Otherwise it will turn to bad practices and jeopardize the whole project for nothing but the bad decision of adopting a not useful enough tool.

- **If you are going to have millions of lines of code later, think twice**: most of the common issues like slow git commands or long time builds are very common. Think twice before adopting a tool, and make sure you can live with the issues that come with monorepos.

Zoover Key Takeaways

While there are challenges with using monorepos, they can enhance developer productivity when used correctly.

- Monorepos can simplify dependency management and code sharing across teams.
- With monorepos, tooling must be efficient to handle larger codebases and maintain developer productivity.
- Enforcing coding standards and practices is easier within a monorepo.
- Monorepos can enhance collaboration across teams as changes can be seen and understood across the entire codebase.
- The decision to use monorepos should be balanced against the scale and requirements of the organization.

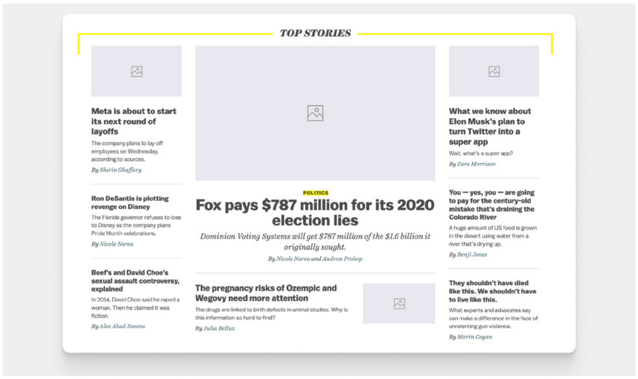
Rebuilding a Featured News Section with Modern CSS: Vox News

By Ahmad Shadeed

Looking at a layout at first glance might imply that it's easy and straightforward to build.[1] The moment you start building the initial layout, you will face challenges that you didn't think about in your initial look at the design.

In this case study, I will rethink how to build the featured news section on Vox.com and try to see if modern CSS will be helpful or not. For example, do we need to use container queries? Or fluid sizing? That's the goal of this case study. It's a journey as I think aloud about building a layout that seems simple.

Analyzing the Section



TOP STORIES

Meta is about to start its next round of layoffs
The company plans to lay off employees on Wednesday, according to sources.
By Shirin Ghaffary

Ron DeSantis is plotting revenge on Disney
The Florida governor refuses to lose to Disney as the company plans Pride Month celebrations.
By Nicole Narea

Beef's and David Choe's sexual assault controversy, explained
In 2014, David Choe said he raped a woman. Then he claimed it was fiction.
By Alex Abad-Santos

POLITICS
Fox pays $787 million for its 2020 election lies
Dominion Voting Systems will get $787 million of the $1.6 billion it originally sought.
By Nicole Narea and Andrew Prokop

The pregnancy risks of Ozempic and Wegovy need more attention
The drugs are linked to birth defects in animal studies. Why is this information so hard to find?
By Julia Belluz

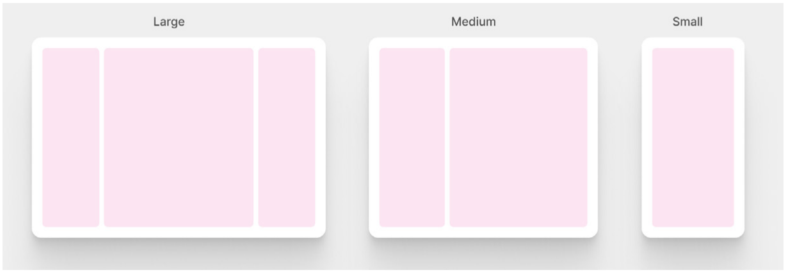
What we know about Elon Musk's plan to turn Twitter into a super app
Wait, what's a super app?
By Sara Morrison

You — yes, you — are going to pay for the century-old mistake that's draining the Colorado River
A huge amount of US food is grown in the desert using water from a river that's drying up.
By Benji Jones

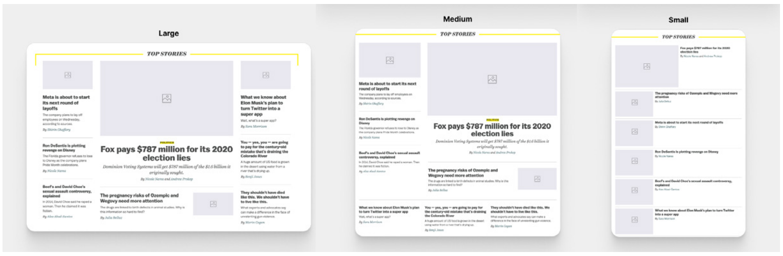
They shouldn't have died like this. We shouldn't have to live like this.
What experts and advocates say can make a difference in the face of unrelenting gun violence.
By Marin Cogan

In the largest viewport, we have a 3-column layout. Two of the columns take 25% of the width, and the middle one takes 50%. Here is a visual that shows them.

1 The original version of this article was published April 2023: https://smashed.by/voxnewscasestudy

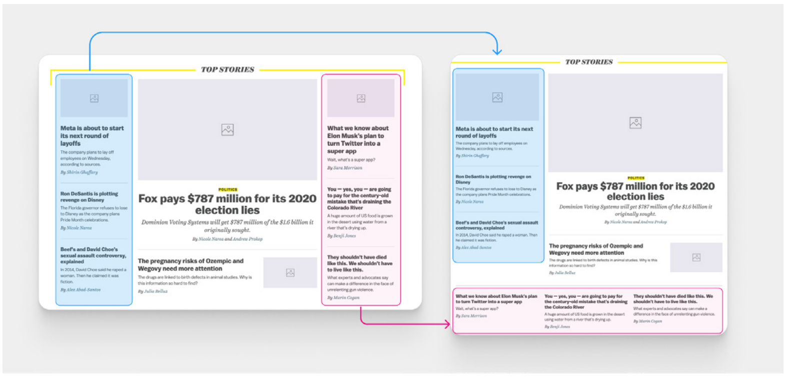


Now that we have an idea about the columns, let's take a look at the components within them.



It might look a bit confusing to spot the differences, but I will walk you through each change so we can have an idea about what's changing on each viewport size.

Changes From Large to Medium


TOP STORIES
Meta is about to start its next round of layoffs
Ron DeSantis is plotting revenge on Disney
Beef's and David Choe's sexual assault controversy, explained
POLITICS
Fox pays $787 million for its 2020 election lies
The pregnancy risks of Ozempic and Wegovy need more attention
What we know about Elon Musk's plan to turn Twitter into a super app
You — yes, you — are going to pay for the century-old mistake that's draining the Colorado River
They shouldn't have died like this. We shouldn't have to live like this.

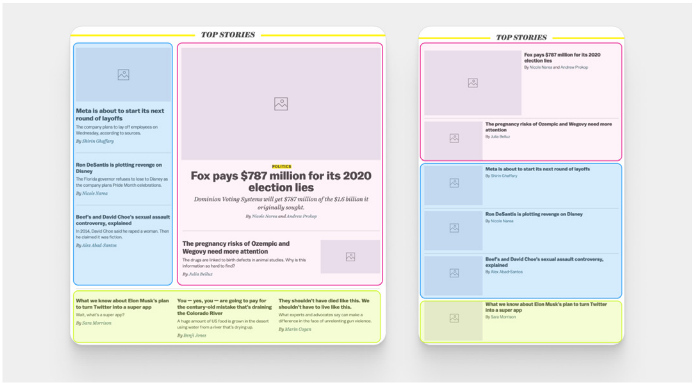
- Featured section: almost the same, but with a different font size that changes based on the viewport width.

- Blue section: the font size of each card title got smaller.

- Pink section:

    - The first article's thumb is hidden.
    - Layout is changed from one column to three columns.
    - Adding a separator at the top of the section.

Changes from Medium to Small


TOP STORIES
Meta is about to start its next round of layoffs
Ron DeSantis is plotting revenge on Disney
Beef's and David Choe's sexual assault controversy, explained
Fox pays $787 million for its 2020 election lies
The pregnancy risks of Ozempic and Wegovy need more attention
What we know about Elon Musk's plan to turn Twitter into a super app
TOP STORIES

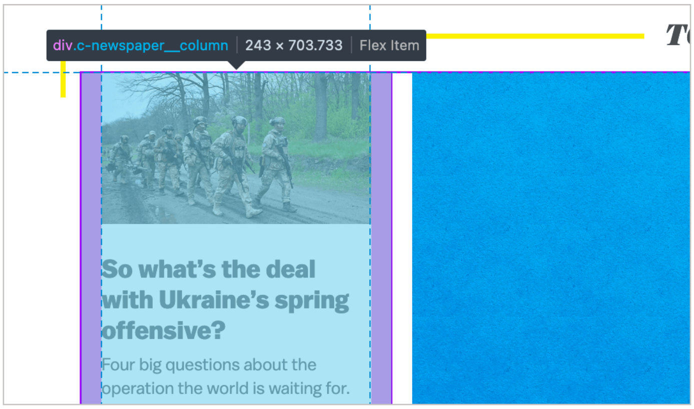
- All articles will switch to the horizontal style with the thumbnail shown for each one.

- The featured article will become horizontal, but with a larger thumbnail to differentiate it from the rest of the articles.

With that in mind, we have a basic outline of how the layout behaves at different viewport sizes. The next step is to build the layout and handle the ordering of the columns.

BUILDING THE MAIN LAYOUT

In Vox.com, CSS Flexbox is used to handle the layout. I'm not a fan of using flexbox for such a purpose as this feels more like a CSS grid use case. I believe the Vox team used flexbox since it was better supported at the time of building the layout.


div.c-newspaper__column | 243 × 703.733 | Flex Item
So what's the deal with Ukraine's spring offensive?
Four big questions about the operation the world is waiting for.

```
@media (min-width: 880px)
    .c-newspaper__column {
        width: 22.5%;
        padding: 0 16px;
    }
}
```

The CSS above is responsible for the following:

- Setting the width of the column. Using the `width` property for that works fine, but we can also use the `flex` property.
- Adding padding on the left and right sides is an old way to introduce a gap between columns. Now we have the `gap` property!

We can use the `flex` property like this:

```
@media (min-width: 880px)
  .c-newspaper__column {
    flex: 0 0 22.5%;
    padding: 0 16px;
  }
}
```

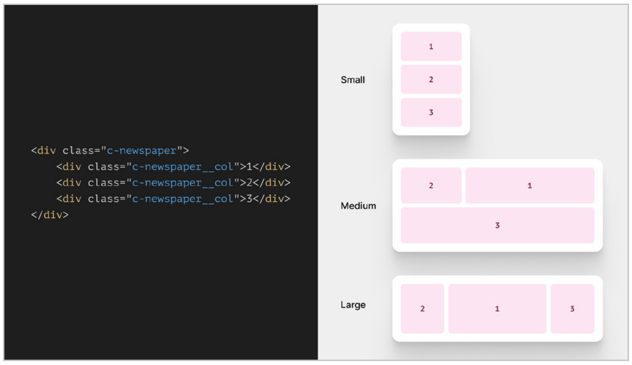
But the good news is that we don't have to use Flexbox.

Nowadays, CSS Grid has excellent browser support and it's easier to deal with the sizing and spacing. Also, I'm an advocate of using Grid for layouts and Flexbox for components.[2]

Consider the following HTML markup:

```
<div class="c-newspaper">
  <!-- Featured column -->
    <div class="c-newspaper__col">1</div>
  <!-- Other columns -->
    <div class="c-newspaper__col">2</div>
    <div class="c-newspaper__col">3</div>
</div>
```

I added numbers for illustrating how each layout column will be reordered on different viewport sizes.



```
<div class="c-newspaper">
    <div class="c-newspaper__col">1</div>
    <div class="c-newspaper__col">2</div>
    <div class="c-newspaper__col">3</div>
</div>
```

CSS Grid sounds perfect for the above, right?

2 https://smashed.by/flexcomponents

First, we need to set up the grid for all sizes.

```
.c-newspaper {
  display: grid;
  grid-template-columns: 1fr;
  gap: 1rem;
}
@media (min-width: 550px) {
  .c-newspaper {
    grid-template-columns: 1fr 1fr 1fr;
  }
}
@media (min-width: 880px) {
  .c-newspaper {
    grid-template-columns: 1fr 2fr 1fr;
  }
}
```

A few things to keep in mind:

- Initially, the grid has only one column. I used CSS Grid to get the benefit of the gap property for spacing.
- When the viewport width is `550px` or larger, the grid will have three columns. The same happens on the larger viewport `880px`, but the second column is double the size of its sibling columns.

The Vox.com styles for the columns are built with the `order` property to reposition the columns on different sizes.

```
@media (min-width: 880px) {
  .c-newspaper__column:first-child {
    order: 1;
  }
  .c-newspaper__column:last-child {
    order: 3;
  }
}
```

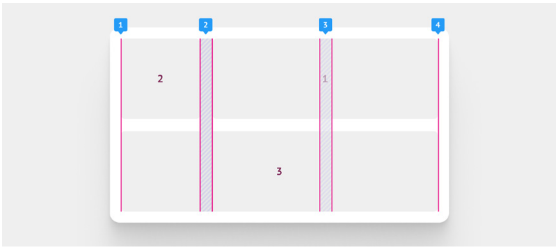
With CSS Grid, the above isn't needed at all as we can reorder the layout by positioning an element on any grid lines we want. Let's explore how to place the layout columns with CSS Grid.

THE MEDIUM VIEWPORT SIZE

We need to position the columns as per the viewport width. For the medium size:

- The first column is placed from line 2 to line 4.
- The second column is placed from line 1 to line 2.
- The third column is placed from line 1 to line 4 (spanning the full width).

```
@media (min-width: 550px) {
  .c-newspaper {
    grid-template-columns: 1fr 1fr 1fr;
  }
  .c-newspaper__col:first-child {
    grid-column: 2/4;
  }
  .c-newspaper__col:nth-child(2) {
    grid-column: 1/2;
    grid-row: 1;
  }
  .c-newspaper__col:last-child {
    display: flex;
    grid-column: 1/4;
  }
}
```

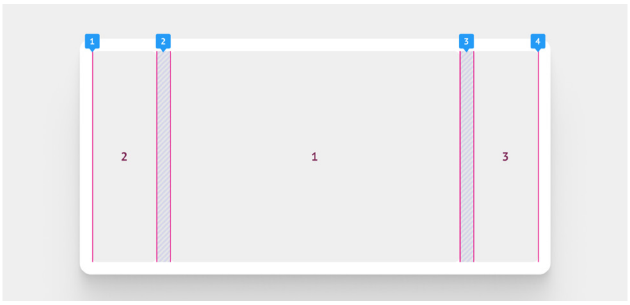


THE LARGE VIEWPORT SIZE

And for the large size, remember that the second column is now `2fr`, so it will have to double the size of the side column.

- The first column is placed from line 2 to line 3.
- The second line stays within the same placement.
- The last column is placed from line 3 to line 4.

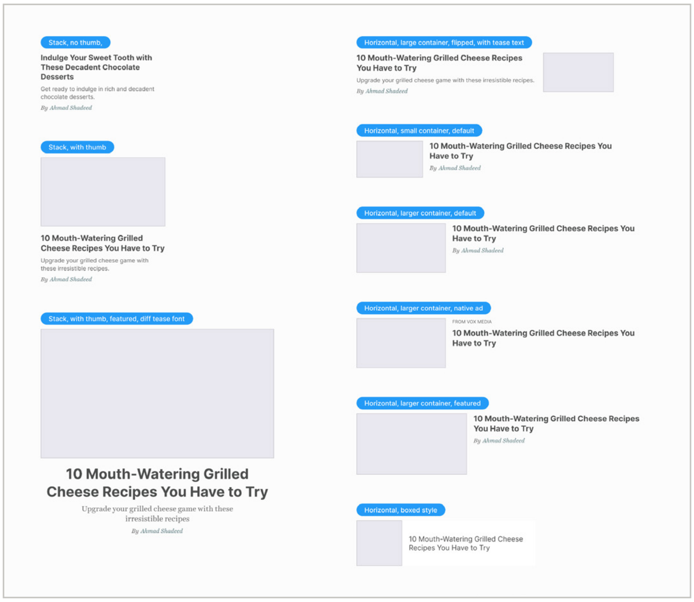
```
@media (min-width: 880px) {
  .c-newspaper {
    grid-template-columns: 1fr 2fr 1fr;
  }
  .c-newspaper__col:first-child {
    grid-column: 2/3;
  }
  .c-newspaper__col:last-child {
    grid-column: 3/4;
  }
}
```



Now that we have a working grid, we can start thinking about the inner components and how to build them.

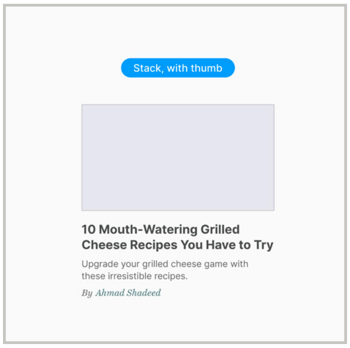
Card Component

This is the core focus of this article, the card component. I compiled a visual of all the variations we have:



Stack, no thumb,
Indulge Your Sweet Tooth with These Decadent Chocolate Desserts
Get ready to indulge in rich and decadent chocolate desserts.
By Ahmad Shadeed
Stack, with thumb
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
Stack, with thumb, featured, diff tease font
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed
Horizontal, large container, flipped, with tease text
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
Horizontal, small container, default
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed
Horizontal, larger container, default
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed
Horizontal, larger container, native ad
FROM VOX MEDIA
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Horizontal, larger container, featured
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed
Horizontal, boxed style
10 Mouth-Watering Grilled Cheese Recipes You Have to Try

All of those can live within the featured section but with a different design variation for each card. Let's take the default card as an example:



Stack, with thumb
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

In Vox.com HTML, the card has the following CSS classes:

```
<div
  class="c-entry-box--compact c-entry-box--compact--article
c-entry-box--compact--hero c-entry-box--compact--2"
></div>
```

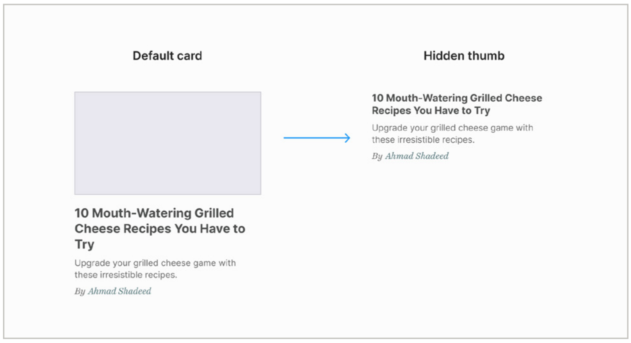
That is a long list of CSS classes, and the class name itself is lengthy, too.

A Look at a Few Details on Vox Layout

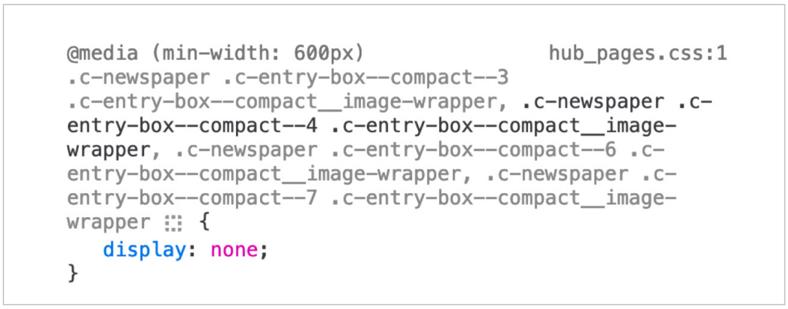
CARD THUMBNAIL

The card component is built in a way that uses a lot of variation classes. For example, here is how the thumbnail is hidden in the plain card:

```
.c-entry-box--compact--7 .c-entry-box--compact__image-wrapper
{
  display: none;
}
```


Default card
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
Hidden thumb
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

A custom variation class is used **for every single card** in the featured section. In total, the CSS looks like this:

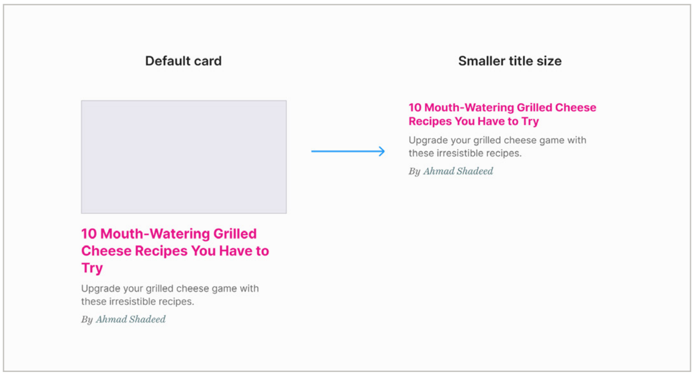


```
@media (min-width: 600px)                    hub_pages.css:1
.c-newspaper .c-entry-box--compact--3
.c-entry-box--compact__image-wrapper, .c-newspaper .c-
entry-box--compact--4 .c-entry-box--compact__image-
wrapper, .c-newspaper .c-entry-box--compact--6 .c-
entry-box--compact__image-wrapper, .c-newspaper .c-
entry-box--compact--7 .c-entry-box--compact__image-
wrapper ⁝⁝ {
   display: none;
}
```

That is too much, I think.

CARD TITLE SIZE

The title size for the default card is `'20px'` and `'16px'` for the plain card (without a thumbnail).



10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

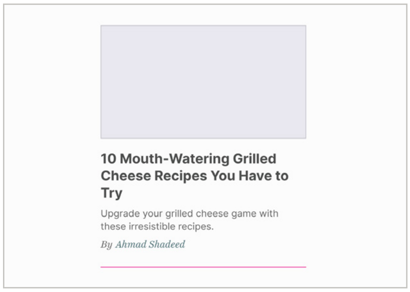
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

Here is how that is handled on Vox.com:

```
@media (min-width: 880px)
  .c-newspaper .c-entry-box--compact__title {
      font-size: .9em;
  }
}
```

The `.c-newspaper` is the main element that contains all the cards, so using it like that to tag the title element doesn't look right to me. What if that needs to be used in another container that doesn't have the class `.c-newspaper`?

SEPARATOR


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

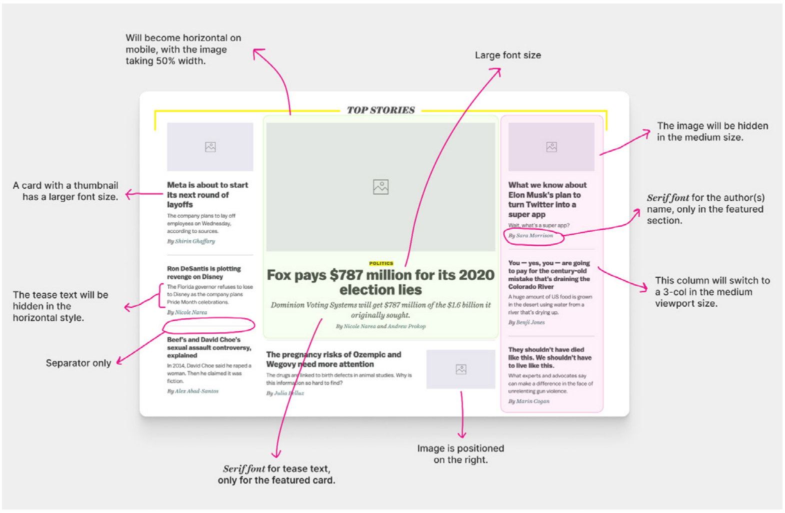
There is a line separate between cards. It's being handled in the CSS like this:

```
.c-newspaper .c-entry-box--compact {
  border-bottom: 1px solid #d1d1d1;
}
```

Two things that don't look good to me here:

- Using `.c-newspaper` element to select the card.
- Adding the separator directly to the card itself. This is a conditional style that isn't related to the card.

Rethinking the Card with Modern CSS


Will become horizontal on mobile, with the image taking 50% width.
Large font size
TOP STORIES
The image will be hidden in the medium size.
A card with a thumbnail has a larger font size.
Serif font for the author(s) name, only in the featured section.
This column will switch to a 3-col in the medium viewport size.
The tease text will be hidden in the horizontal style.
Separator only
Fox pays $787 million for its 2020 election lies
Image is positioned on the right.
Serif font for tease text, only for the featured card.

The main motivation for this article is the card component. When I started thinking about it, I got the idea to use some or all of these features:

- CSS Grid
- `aspect-ratio`
- `text wrap: balance`
- CSS `:has`
- Fluid sizing and spacing
- Size container queries
- Style container queries

I already explored using CSS grid for the main layout. Here is what the HTML markup looks like:

```
<div class="c-newspaper">
  <div class="c-newspaper__col">
    <div class="c-newspaper__item">
      <article class="c-card">
        <!-- Card component -->
      </article>
    </div>
    <div class="c-newspaper__item"></div>
    <div class="c-newspaper__item"></div>
  </div>
  <!-- Other columns -->
</div>
```

The card component lives within the `.c-newspaper__item`, which acts as the card container.

Generally speaking, I like to wrap the component in an abstract container. This is useful for:

- adding borders,
- controlling the spacing, and it
- works well for size container queries.

CARD META FONT FAMILY

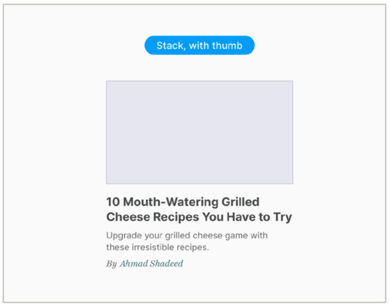
When the card component is within the featured section, the font family of the author's name is different. To do that, we can check if the following container query works, and if yes, the font will be applied.

```
@container main (min-width: 1px) {
  .c-card__meta {
    font-family: "Playfair Display", serif;
  }
}
```

DEFAULT CARD STYLE

We need to set a default card style that we can style. In this case, both the horizontal and stacked styles are used equally, but I will assume that the stacked card is used more, just for the sake of the article.

```
<article class="c-card">
  <div class="c-card__thumb"></div>
  <div class="c-card__content">
    <h3 class="c-card__title"></h3>
    <p class="c-card__tease"></p>
    <p class="c-card__meta”></p>
  </div>
</article>
```


Stack, with thumb
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

Cool! Let’s go from there for the rest of the variations.

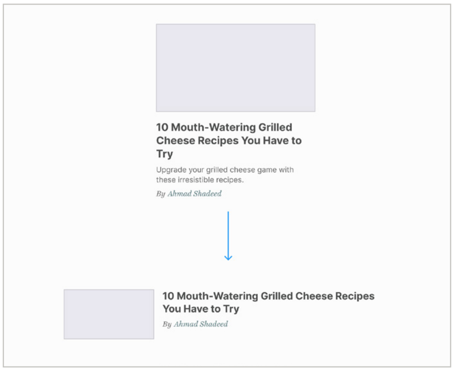
HORIZONTAL STYLE

The card will flip to the horizontal style when its container is larger than `300px` and the CSS variable `--horizontal: true` has been set on the container.

```
<div class=”c-newspaper__item” style=”--horizontal: true;”>
  <article class=”c-card”></article>
</div>
```

```
.c-newspaper__item {
  container-type: inline-size;
  container-name: card;
}
@container card (min-width: 300px) and style(--horizontal:
true) {
  .c-card {
    display: flex;
    gap: 1rem;
  }
}
```

Notice that I combined a size and a style container query. The size query works based on the container width, while the style query works by checking if the CSS variable is there.


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed

We also have the same variation but with the card thumbnail positioned being flipped. We can do that via the `order` property.

To query that, we need to add the variable `--flipped: true`.

```
<div
  class="c-newspaper__item"
  style="--horizontal: true;
         --flipped: true"
></div>
```

At first, I tried the following CSS but it didn't work as expected. It's not possible to merge two container queries for different containers. In my case, the containers are main and card.

```
/* That didn't work */
@container main (min-width: 550px) and card style(--flipped:
true) {
}
```

After reading the spec[3] I noticed the following:

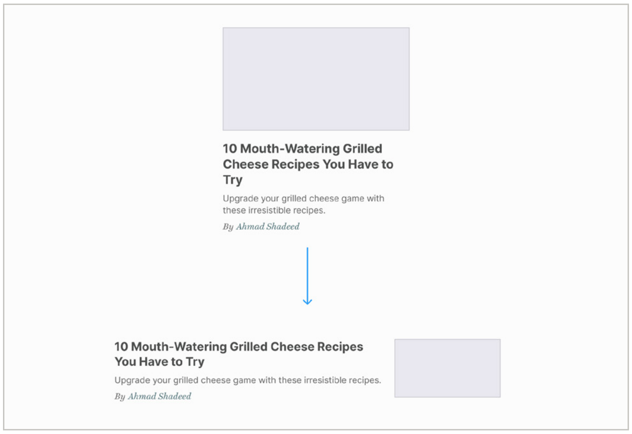
> *While it is not possible to query multiple containers in a single container query, that can be achieved by nesting multiple queries*

I nested the style query inside another container query. In plain words, that is like saying:

> *When the container main width is equal to or larger than 550px and the CSS variable --flipped is set on the cards container, apply the following CSS.*

```
.wrapper {
  max-width: 1120px;
  margin: 1rem auto;
  padding-inline: 1rem;
  container-name: main;
  container-type: inline-size;
}
@container main (min-width: 550px) {
  @container card style(--flipped: true) {
    .c-card__thumb {
      order: 2;
    }
  }
}
```

3 https://smashed.by/containerrule


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

To learn more about container queries, here are a few write-ups on the topic:

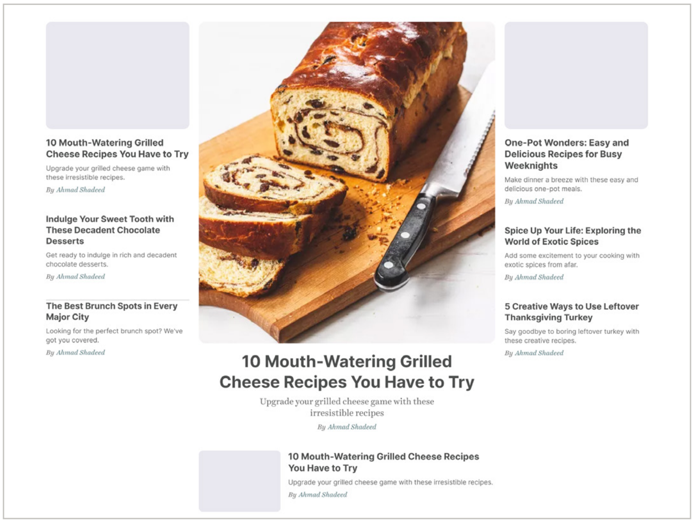
- Say Hello To CSS Container Queries
  https://ishadeed.com/article/say-hello-to-css-container-queries/

- CSS Style Queries
  https://ishadeed.com/article/css-container-style-queries/

CARD THUMBNAIL ASPECT RATIO

The current way of implementing the card thumbnail doesn't account for when there is an image with a different aspect ratio. We can use the CSS `aspect-ratio` property to force the card thumb to have the same aspect ratio.

Let's assume that I added a large image that has a different aspect ratio. We'll end up with something like this:


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
Indulge Your Sweet Tooth with These Decadent Chocolate Desserts
Get ready to indulge in rich and decadent chocolate desserts.
By Ahmad Shadeed
The Best Brunch Spots in Every Major City
Looking for the perfect brunch spot? We've got you covered.
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
One-Pot Wonders: Easy and Delicious Recipes for Busy Weeknights
Make dinner a breeze with these easy and delicious one-pot meals.
By Ahmad Shadeed
Spice Up Your Life: Exploring the World of Exotic Spices
Add some excitement to your cooking with exotic spices from afar.
By Ahmad Shadeed
5 Creative Ways to Use Leftover Thanksgiving Turkey
Say goodbye to boring leftover turkey with these creative recipes.
By Ahmad Shadeed

To avoid that, we can define an aspect ratio:

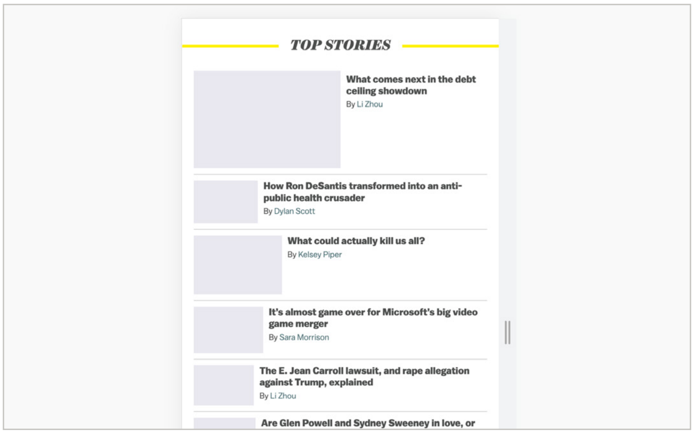
```
.c-card__thumb img {
  aspect-ratio: 5/3;
  object-fit: cover;
}
```

CARD HORIZONTAL STYLE

On Vox.com, the horizontal card style was built in a way that feels a bit unnecessary.

```
/* css from vox.com */
.c-entry-box--compact__image-wrapper {
  width: 30%;
}
.c-entry-box--compact__body {
  flex-grow: 1;
  width: 70%;
}
```

Why is that? I guess that is to avoid having such a UI behavior:


TOP STORIES

What comes next in the debt ceiling showdown
By Li Zhou

How Ron DeSantis transformed into an anti-public health crusader
By Dylan Scott

What could actually kill us all?
By Kelsey Piper

It's almost game over for Microsoft's big video game merger
By Sara Morrison

The E. Jean Carroll lawsuit, and rape allegation against Trump, explained
By Li Zhou

Are Glen Powell and Sydney Sweeney in love, or

Notice that I mentioned "UI behavior," not a bug. The above is a default behavior for flexbox. We need to force the image to have a fixed and consistent size.

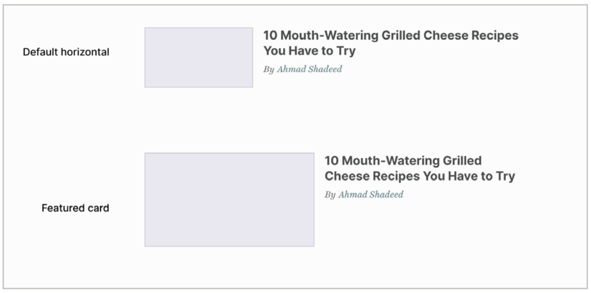
```
.c-entry-box--compact__image-wrapper {
  flex: 0 0 30%;
}
.c-entry-box--compact__body {
  flex-grow: 1;
}
```

We can fix that by simply using the `flex` property. No need to use the width.

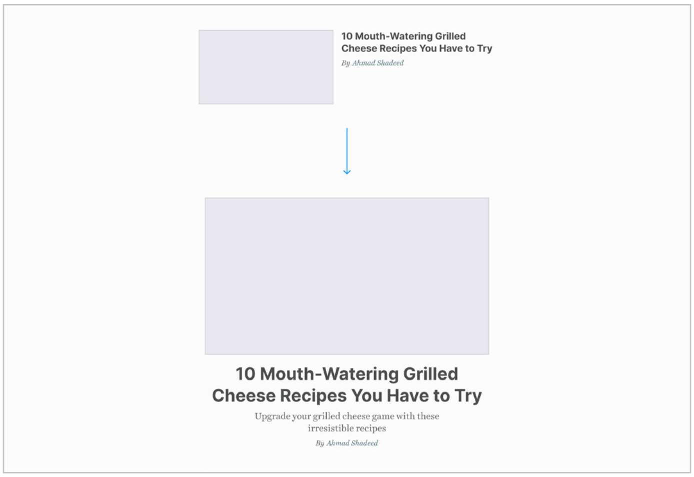
FEATURED STYLE

The featured card is displayed horizontally when the container width is small and will change to the stacked styles on larger sizes. In this case, the thumbnail becomes larger and takes 50% of the width.

Here is a comparison between a default horizontal style and the featured one.


Default horizontal
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed
Featured card
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed

When the container width becomes larger, the card style will become stacked.


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed

To implement that, I used the `--featured` variable on the card's container.

```
<div class="c-newspaper__item" style="--featured: true;">
</div>
```

Firstly, I added the horizontal style as default.

- Added `flex` to turn on the horizontal design.
- The card thumb takes 50% of the available width.

Changed the font family to a serif font and a larger size, as per the design.

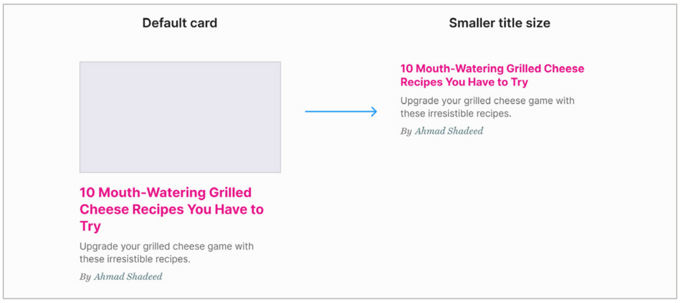
```
@container style(--featured: true) {
  .c-card {
    display: flex;
    gap: 1rem;
  }
  .c-card__thumb {
    flex: 0 0 50%;
  }
  .c-card__tease {
    font-family: "Playfair Display", serif;
    font-size: 19px;
  }
}
```

When the container size gets larger, the browser will apply the stacked styling to the card.

```
@container main (min-width: 550px) {
  @container card style(--featured: true) {
    .c-card {
      flex-direction: column;
      gap: 0;
    }
    .c-card__title {
      font-size: calc(1rem + 2.5cqw);
    }
    .c-card__content {
      text-align: center;
    }
    .c-card__thumb {
      flex: initial;
    }
  }
}
```

PLAIN CARD

In this variation, the font size gets smaller. That happens when the image is hidden. At first, I thought about using CSS `:has` to check if the card thumb is displayed or not.


Default card
Smaller title size
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

In Vox.com, the card thumb is hidden via CSS, so it's not possible to use :has as it will be valid even if the thumb is hidden.

```
<article class="c-card">
  <div class="c-card__thumb"></div>
  <div class="c-card__content"></div>
</article>
```

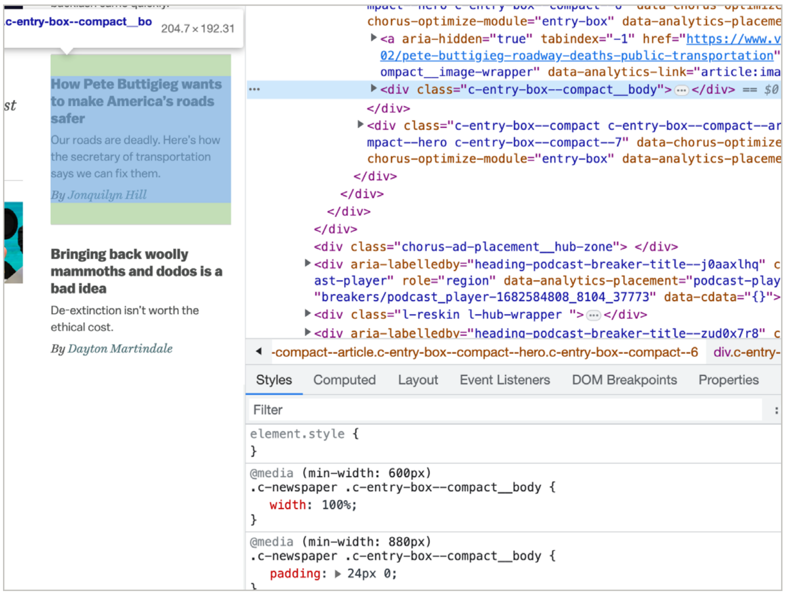
```
.c-card__thumb {
  display: none;
}
/* This will always work. */
.c-card:has(.c-card__thumb) .c-card__title {
  font-size: 19px;
}
```

If the image can be conditionally added via JavaScript, then we can use `:has`. Otherwise, I will default to a style query.

```
@container main (min-width: 550px) {
  @container card style(--compact: 2) {
    .c-card__title {
      font-size: 19px;
    }
  }
}
```

SPACING AND SEPARATORS

The current way in Vox.com to handle the spacing is by adding padding directly to the card. I don't prefer that. The card styles shouldn't depend on where it lives. The spacing should be added to the card's wrapper instead.


c-entry-box--compact__bo
204.7 × 192.31
How Pete Buttigieg wants to make America's roads safer
Our roads are deadly. Here's how the secretary of transportation says we can fix them.
By Jonquilyn Hill
Bringing back woolly mammoths and dodos is a bad idea
De-extinction isn't worth the ethical cost.
By Dayton Martindale

```
          chorus-optimize-module="entry-box" data-analytics-placeme
          ▶<a aria-hidden="true" tabindex="-1" href="https://www.v
           02/pete-buttigieg-roadway-deaths-public-transportation"
           ompact__image-wrapper" data-analytics-link="article:ima
...       ▶<div class="c-entry-box--compact__body">⋯</div> == $0
          </div>
         ▶<div class="c-entry-box--compact c-entry-box--compact--ar
          mpact--hero c-entry-box--compact--7" data-chorus-optimize
          chorus-optimize-module="entry-box" data-analytics-placeme
        </div>
      </div>
    </div>
  </div>
  <div class="chorus-ad-placement__hub-zone"> </div>
 ▶<div aria-labelledby="heading-podcast-breaker-title--j0aaxlhq" c
  ast-player" role="region" data-analytics-placement="podcast-play
  "breakers/podcast_player-1682584808_8104_37773" data-cdata="{}">
 ▶<div class="l-reskin l-hub-wrapper ">⋯</div>
 ▶<div aria-labelledby="heading-podcast-breaker-title--zud0x7r8" c
◀ -compact--article.c-entry-box--compact--hero.c-entry-box--compact--6   div.c-entry-
```

Styles   Computed   Layout   Event Listeners   DOM Breakpoints   Properties

Filter

```
element.style {
}
@media (min-width: 600px)
.c-newspaper .c-entry-box--compact__body {
  width: 100%;
}
@media (min-width: 880px)
.c-newspaper .c-entry-box--compact__body {
  padding: ▶ 24px 0;
```

To make things easier, I added a CSS variable `--gap` to each column.

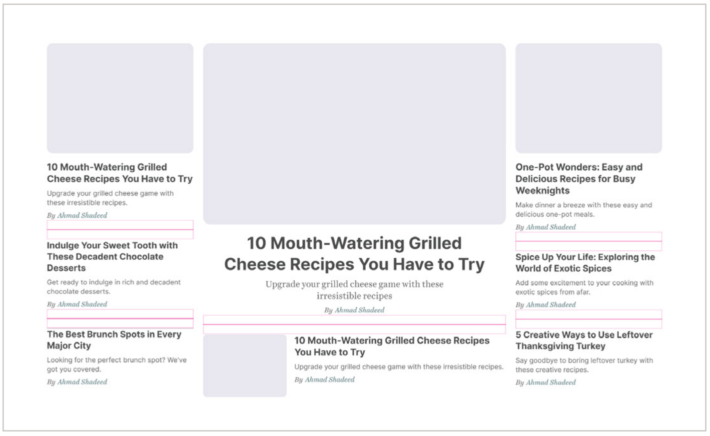
```
.c-newspaper__col {
  --gap: 20px;
  display: flex;
  flex-direction: column;
}
```

I added a `margin-block` to each card wrapper.

- On small viewports, there are no separators.
- When the size is medium, there are separates for the first two columns, and one border for the last one.

The CSS property `margin-block` is a logical property that means both `margin-top` and `margin-bottom`.

```
@media (min-width: 550px) {
  .c-newspaper__item:not(:last-child):after {
    content: "";
    display: block;
    height: 1px;
    background-color: lightgrey;
    margin-block: var(--gap);
  }
  .c-newspaper__col:last-child {
    border-top: 1px solid lightgrey;
    padding-top: var(--gap);
  }
}
@media (min-width: 880px) {
  .c-newspaper__col:last-child {
    padding-top: 0;
    border-top: 0;
  }
  /* Add separators to the last column */
  .c-newspaper__col:last-child
    .c-newspaper__item:not(:last-child):after {
    content: "";
    display: block;
    height: 1px;
    background-color: lightgrey;
    margin-block: var(--gap);
  }
}
```


10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

Indulge Your Sweet Tooth with These Decadent Chocolate Desserts
Get ready to indulge in rich and decadent chocolate desserts.
By Ahmad Shadeed

The Best Brunch Spots in Every Major City
Looking for the perfect brunch spot? We've got you covered.
By Ahmad Shadeed

10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed

10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes.
By Ahmad Shadeed

One-Pot Wonders: Easy and Delicious Recipes for Busy Weeknights
Make dinner a breeze with these easy and delicious one-pot meals.
By Ahmad Shadeed

Spice Up Your Life: Exploring the World of Exotic Spices
Add some excitement to your cooking with exotic spices from afar.
By Ahmad Shadeed

5 Creative Ways to Use Leftover Thanksgiving Turkey
Say goodbye to boring leftover turkey with these creative recipes.
By Ahmad Shadeed

You might be thinking, why not use `gap`? The reason is that I won't use modern CSS for the sake of using it. It's not useful here because:

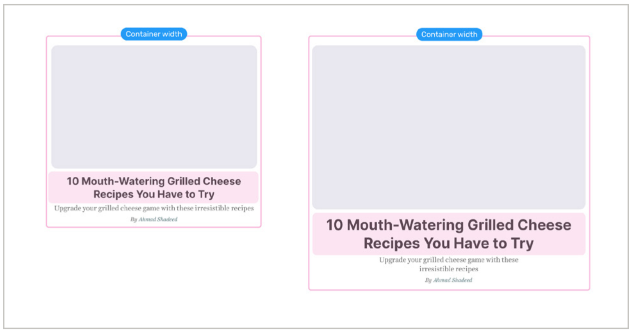
- It only works for one part of the spacing, and I have to use `margin-top` with it.
- I wish there was a native CSS way to add borders, just like the CSS property `column-rule` in CSS columns.[4]

CONTAINER UNITS

One thing that I like about container queries is the ability to use container units. They are like viewport units but for a specific container. Isn't that powerful?

4 https://smashed.by/columnrule

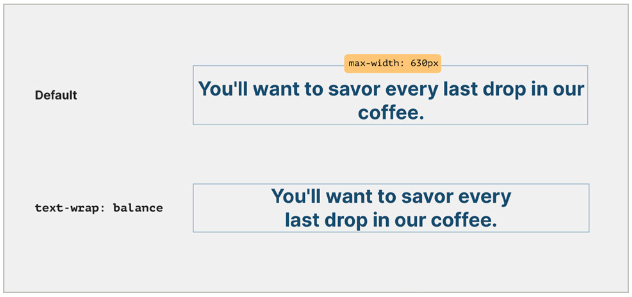
```
@container main (min-width: 550px) {
  @container card style(--featured: true) {
    .c-card__title {
      font-size: clamp(1rem, 6cqw, 2rem);
    }
  }
}
```


Container width
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed
Container width
10 Mouth-Watering Grilled Cheese Recipes You Have to Try
Upgrade your grilled cheese game with these irresistible recipes
By Ahmad Shadeed

Learn more about container query units.[5]

TEXT BALANCING

Recently, I wrote about the new CSS feature `text-wrap: balance`, which is almost fully supported at the time of this writing. Safari has yet to adopt it, but has implemented it in Safari Technology Preview, meaning it could release at any time.


Default
max-width: 630px
You'll want to savor every last drop in our coffee.
text-wrap: balance
You'll want to savor every last drop in our coffee.

5 https://smashed.by/queryunits

In the layout that I'm building, we can leverage that for all the text content. It can make the layout look more organized.

```
{% include video.html
    url=”../../assets/featured-section/text-wrap-balance.mp4”
    caption = “”
%}
```

Learn more about text wrap balancing.[6]

The final demo for this case study is available on CodePen.[7]

Disclaimer: The design isn't identical to Vox, this demo focuses more on the layout and components implementation.

Vox News Key Takeaways

Embracing modern CSS techniques can lead to more efficient, responsive, and visually appealing designs.

- Modern CSS techniques like Flexbox and Grid can greatly simplify complex layouts.
- Features like CSS variables and custom properties can enhance reusability and maintainability.
- Building a responsive design becomes more straightforward with modern CSS features.
- Embracing modern CSS can improve performance because of reduced reliance on JavaScript for layout and design tasks.
- Careful planning, testing, and iteration are required to ensure designs are robust across different devices and browsers.

6 https://smashed.by/textwrapbalance
7 https://smashed.by/voxnewscodepen

Auto Trader: Around the Artifacts of Design Systems

By Dan Donald

It can be easy to assume that everyone needs a design system,[1] that you can pick one off the shelf or put one together pretty quickly, and your problems are over. As with many things on the web, your mileage may vary. What I want to share with you are some observations from the last few years, not just from myself but from people that have been part of our design systems journey at Auto Trader.

We started in a very different place to where we find ourselves today: loads of inconsistencies, duplication, communication that needed to be improved, and ultimately the quality and speed of output weren't what they should be. Our practices today have dramatically improved on so many fronts, one aspect being our design system.

The Problem Space

There are so many really great articles out there about design systems, how to design and code them; but let's step back to the beginning.

- What problems do you feel that a design system might address?
- Why do you think a design system might ameliorate them?

If you're clear on your specific issues, it helps not only to inform your solution but also provides a narrative that might resonate with stakeholders.

1 The original version of this case study was published in April 2022: https://smashed.by/autotradercasestudy

Even at this early stage, language matters. What do you think a design system is? As part of your proposed solution, is it actually a style guide, a component library, a design library, or a more well-rounded system? You might not need something all singing and dancing for your project or organization. Starting with one aspect doesn't mean you can't evolve into something else later!

So, you have a sense at this stage that you may be on a greenfield project – or as we were, adding in the foundations to a large traffic site that already existed – and you may have a view on what form your design system might take. Before diving in and starting to plan anything out, we can be clearer with ourselves about where the benefits and the risks might be.

You could start as simply as making lists.

Potential benefits

- Encourages greater communication between disciplines.
- Greater quality and consistency of our output.
- Should be able to get new content or features to market quicker.

Potential risks

- Other colleagues might not want to use it.
- It takes too much time to get it to a place where it produces value.
- We use the wrong tooling or software.

You could take this further if, in your organization, you need a business case and use a SWOT analysis[2] as a way to form something robust. This allows you to look at the strengths, weaknesses, opportunities, and threats, and it is often used in business planning. It doesn't have to be a laborious task, but it can help to look at your potential system from other viewpoints.

2 https://smashed.by/swotanalysis

Mitigating avoidable risks is important. Being clear with yourself about what could go wrong, and what you can do about it helps to make your solution more robust. On the flip side, the potential benefits can lead us into thinking about what our definition of success might look like – our "North Star". What might good look like for your system?

From the vision piece, you need to be able to start somewhere, which leads us to more avenues of questioning:

- What kind of a system helps you to progress in a sustainable way for the needs and resources you have?
- Could a style guide be all you need?
- Is some form of a component library enough to get you working in a better way?
- What current and potential audiences might the system have?

You may start with it being a design project to address consistency across a design team, but acknowledging that for it to evolve it needs a wider range of skills, which is a kind of debt that will be built-in. In some cases, much as we did, it might make sense for an initial solution to be replaced by something else further down the line. Not being wedded to a particular solution can be hard, especially if it's your baby, but going back to that North Star and the purpose of the system gives us a healthy reminder of when it can be time to let go.

What to Measure?

Many things may be measurable, from the sentiment of attitudes towards the system and experience using it, through to the number of components in the system and what's actually used on the site (or in your app). Actually, looking at where a given component is used can have some useful benefits when it comes to later stages of working with your system, as it helps you to gauge the risk of a proposed change and where you might see some impact.

From your picture of what success looks like, are there measurables that can help tell your story to stakeholders or that give you a sense of how well you're doing? While in our scenario we didn't set out a list of KPIs, we were clear that it should power the majority of the consumer website and look into how or when our native apps may work with it. On the back of our refresh project, we'd be taking care of most of the site outside the focus of consumer journeys where our team would act more as support for the teams around us. That gave us confidence that if we were able to deliver output to the site on a regular basis, we would be able to meet that vision we'd set out.

You might often hear the "fail fast, fail often" mantra thrown around, but in the early days of validating how your design system might work, this can be invaluable.

Assuming that you have a well-rounded design system, you may have designers, content designers, test engineers, front-end and back-end developers, product people, and delivery folks all potentially finding this as a part of their lives. Again, the language we use matters to ensure there's a shared understanding across domains.

What's in a Name?

One key thing that helped us was naming things clearly across disciplines, so we referred to the same thing and were clear on its intent.

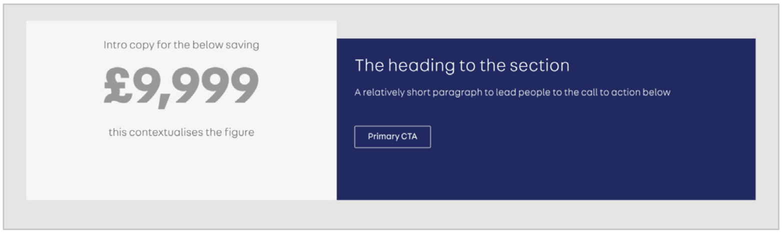
That purpose transcends specialisms and helps give clarity to what an object in your design system is for, what problem it solves, and what role it plays. That clarity of purpose pays off in many ways over time.

Using a well-worn example, a button is a button, isn't it?
Well, not always.

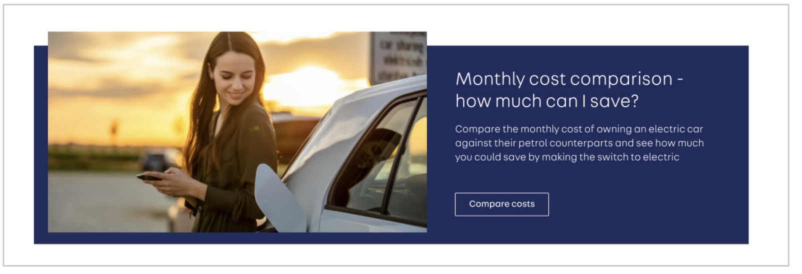
While the visual asset may have the appearance of a button, when it comes to applying it to code, is this a button tag? Is it a link that looks like a button (which is a debate in itself!)? In your framework, is it an internal router link?

There is room for debate, but we ended up in a place where the visual asset gave us the language to talk about it, and the technical execution might differ based on a use case. We're all now talking about buttons, even when the actual code differs.

We have a component, originally called the "Promo Section"; it was intended for calling out key parts of a value proposition. The following example shows where intent and language differ, as it's now become a more generic content block. The work now is to look at use cases in the wild and choose as a team: whether we accept that's what it is and capture it or look at whether it needs to be more than one component based on those use cases.


Intro copy for the below saving
£9,999
this contextualises the figure
The heading to the section
A relatively short paragraph to lead people to the call to action below
Primary CTA

A link that looks like a button


Monthly cost comparison - how much can I save?
Compare the monthly cost of owning an electric car against their petrol counterparts and see how much you could save by making the switch to electric
Compare costs

Here again, what looks like a button isn't really one.

There's a lot to think about, and we've not even touched on the design files or what your code might look like yet!

People-Powered

You might start as a team of one or have a group around you to make this happen, but defining roles and responsibilities early on is important. Having an amazing Figma library is great, but if it doesn't at some point become code and have a release process, it's a collection of pictures of what could be. Underneath a lot of what a design system appears to be is a mechanism for fostering better communication and understanding. A new system might start from any discipline, but it needs others to be truly impactful.

Not everyone is fortunate enough to have dedicated time or resources to make a design system, so there's often a balance to be struck between the value it might offer and how much time it might need to produce that value. You might not be able to start with the ideal system you have in mind but can find a way of communicating the value proposition you believe the design system has. Going back to the problem space, what might be a great example to start with? Looking at your "North Star", what demonstrates the potential through a hack or experiment?

Ownership and Community

I've felt less like an owner of the system and increasingly more like a shepherd. The system isn't *mine* – it's shared. The responsibility of ensuring it persists shouldn't rest on any individual, as if things go well, this design system will be a major asset to your organization.

Designing the "people bit" and structures around the system itself pays off. Initially, we kept those directly involved to a minimum, through availability and, in part, through choice. There were some fundamental decisions to make and test out and try to break.

With some great support from our engineering colleagues, we worked on both output (making the first landing pages) and road-testing (and breaking) the design system mechanism itself. Alongside, we'd started talking more broadly around the business about what the design system was, how it was different from previous projects, and to set and manage some expectations.

As we weren't just making the design system but working through refreshing the site, we worked in the open, with our plan and breakdown on a board, so whenever people came by our area of the office, we could talk them through it and get some initial feedback. At that early stage, we also talked through the refresh of the site (and so the design system) with departments that might not normally be in the loop.

Outreach

Taking time to explain to different audiences, in their domain language, what the design system is and why it matters can help you to gain advocates, and the advocacy model can help foster a community around the system as it grows and matures.

Having people who aren't a part of the immediate team enthuse about it helps spread that word on your behalf.

In the earlier days, having "good news stories" or case studies around what the design system has helped us to achieve amplified the messaging we were trying to spread. We were able to be far more reactive with content, because our design system had solved problems that mapped to that kind of project. We could then have quick conversations focused on what the content was trying to achieve, and we were able to get something live far quicker than previously and to the same quality as other parts of the site.

Creating a community around the design system can't be forced. Communities don't just arise, and they can take time to nurture. Part of helping it along is that sense of shared ownership, which can mean many things, including being able to actively contribute or participate. How can people outside the immediate team around the system propose new additions or changes? Is there a clear feedback mechanism? One of the best examples I remember seeing was in GDS (the GOV.UK Design System[3]).

There's some inherent tension, as there has to be a balance between community and engagement, contribution, and a governance process that has rules and processes. How this coalesces around something that works for you and your circumstances will differ. Governance (how you define responsibilities and workflows) can become as much established through the community as imposed on it. Having something as a starting point that can be critiqued is often easier than a blank canvas!

3 https://smashed.by/govukdesignsystem

Note: Although it's a few years old, it's worth checking out Brad Frost's article, "A Design System Governance Process," from 2019.[4]

While there's a huge amount of care and effort involved in creating and maintaining, everything in the design system needs to be up for debate and a challenge from the community you aim to form around it. All of this helps with engagement, but it also helps make what's in the system more robust, whether that's through design changes, code improvements, or just providing better documentation.

It's about supporting people to understand that the design system is not something set in stone but a way of describing and facilitating solutions to problems.

Workflow, Communication, and Evolution

This flex between what a specialism needs within its domain and what allows for broad communication is really important. How your work goes from inception to somehow appearing on a live website is big stuff. This workflow doesn't yet dictate the tooling or presentation of your system but underpins its value.

You might start mapping out a workflow like this:

- A need emerges.
- The team (TBS) discusses it, and the outcome of this forms a proposal.
- This proposal is worked on and taken to a critique session.
- Once complete, the component is reviewed and made available to the system.

4 https://smashed.by/governance

- Once tested, it can be used.
- When used *in situ*, gather feedback and see how it performs in the wild.

Considering how to manage change isn't always easy. We chose to start our components in an opinionated way: they'd do one job and wouldn't include much logic. While you could craft them in a more futureproof way, you can't predict what change is needed, but enabling and facilitating change is an important part of any design system. A new bit of content needs to be passed in a different style of a call to action.

How should we update not just the component itself but its uses all over the site? Baking in some assumptions that changes are not just possible but actually desirable is really important. How do you roll out a breaking change across your codebase? If you update your component in your design tool of choice, what knock-on effects are there?

Each item in a proposed workflow might have the depth to be explored. For us, we wanted to make it feel like a "push" when we updated a component, so the live site would always be up to date – every change would be versioned and released immediately to the site like we "pushed" it out. Under the hood, we use conventional commits[5] to automatically trigger versioning of our code, which runs our build processes that include automated tests.

Once pulled into an app in one of our React apps, future builds of the design system would trigger upstream builds for that app, so every intentional change is pushed out to the site and, in theory, the components should always be up to date, unless a developer specifically needs otherwise. Explaining that process hopes to illustrate the simple workflow intent; to "feel like a push" actually involved a chunk of engineering to ensure pipelines and build process worked

5 https://smashed.by/commits

as we'd expect. We also have the ability to flag a potentially breaking change as a beta release to manually pull into apps to validate before making that change permanent.

Understanding Change and Evolution

Some components might start off looking or functioning in similar ways, but how might this play out over time? It might make sense to build them from a shared look and feel or set of functionality. A better way to solve a given problem emerges, and so that link to another component no longer makes sense.

There's an element of understanding that we bake potential tech or design debt into the components we create. While there's a contract in the code between your component and its context of use that needs to be preserved, the way it's constructed can change. So, the balance lies between not over-engineering every component when it's created and considering how change might happen with a sense of how scenarios might play out.

Back to our example of the "Promo Section" I mentioned earlier. This has been used for more than its initial intent, but we can learn from that. There are some options we can explore:

- Do we keep it as it is and expand on its intended purpose?
- Do we look at what use cases have emerged and split this into multiple components?
- Is it something that needs reevaluating entirely?

That second option is worth exploring. Maybe this split into two components means that they still look the same? If they do, maybe they use some of the same mark-up and styles and share that common base. Following this through, we might have better or different ways of solving problems these components cause today. Using the same base is a practical short-term solution and, realistically, we can't know if we'll ever change them in the future. If one diverges its presentation, have we baked in some debt, or do we accept that and factor it into future changes?

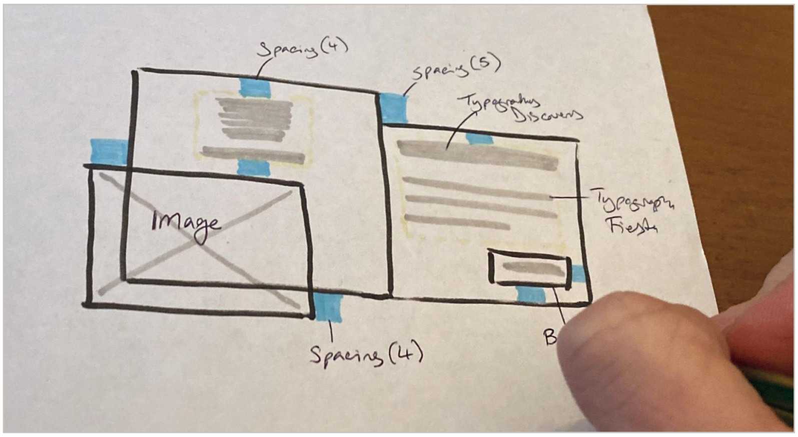
The Source of Truth and Making Proposals

One early principle we had was that live code trumps the design. That's a controversial statement in some circles, so I'll explain: our users are actively using and experiencing our design system components. As good as the work and thinking are in your design tool, until it makes its way through to the live website, it's unrealized potential. Keeping naming, structure, and change tight prevents design work from diverging too far too soon for the system. That doesn't mean that play and experimentation are in any way limited, just that they should exist outside of the system until the concept is ready to be promoted. And so there's another aspect of change, when change is needed or new components are required.

We've done some work around our proposal structure (which as I write is in its early days), but so far, so good. A proposal starts by recognizing the component's purpose: that is, what problem it was created to solve. That actually starts the basics of the documentation for it too, and capturing that early. We have regular open-invite design system-focused sessions, where proposals can be discussed and challenged, and from that designs can be critiqued, and code could be submitted as pull requests. As a collective, we try not to make the process too labored but also hold a proposal to some level of rigor.

A proposal might include some of these:

- **Purpose/Intent**
  What problem does the suggested component solve?
- **Use Cases**
  Often a component would be proposed to the design system if it was needed in more than one place. What use(s) does it have, and how does the use case map to its intended purpose?
- **Anatomy**
  What are the elements, spacing units, and typography that make the component? (Early experiments with this seem useful for us.)
- **Related Components**
  If this isn't a fit for what you want, what others do similar jobs?



An example of a design system component's anatomy with spacing units and typography

What we've found is that this actually becomes part of the documentation, before a component is actually added to the system!

In the early days or thinking about our approach, we also had the notion of "lenses" to look at components and design patterns – different ways to turn the work around in your mind and see them from different angles. Some might be reminders, some technical, some not.

Testing	What kinds of testing give us confidence in this component? Visual regression (VRT), automated tests, manual testing?
Tracking	Is this something that should be tracked in some way? What should we be tracking? Is tracking dependent on a state or interaction?
Accessibility	How much can we bake in to ensure that everything is as inclusive as we can make it? Is the mark-up semantic? Does it need to provide options to ensure it is based on context of use?
Content	What do our content designers need from this component? Is there guidance we can add with how to get the best use of it?
SEO	Is there anything this component should do to consider how it can support search engines beyond the content and accessibility lenses? Is there a relevant schema that may be worth including?
Performance	Is there anything we need to consider about how this performs? Does it use a third party or assets that aren't already present? How can we moderate its impact? What is the component's responsibility or that of the app it's consumed in?
Motion	Should it have any animations or transitions in the component or a state of it? Ensure it works without (prefers-reduced-motion)
States	Loading/unloading. Interaction states (focus, hover, disabled, etc.). View states (is it in or out of the visible viewport? See Intersection Observer).
Triggers and actions	Should some functionality be triggered? Often this would be linked to an interaction state but could be more open than that.
Viewport events	Has the resize or orientation change event been triggered on the viewport?
Coding defensively	What if we don't have the data or content we expect to be passed to it? What if we have too much? Can the component fail in a graceful way?

By no means is this an exhaustive list, but it might help with how you can think about your components differently. Think of some useful prompts of your own, based on how your site works – forming that together might be a great way to bring some different disciplines together!

Conclusion

There's a lot to consider, but it doesn't have to all be done at the beginning. Governance, workflow, communication, and community are all really important and, more often than not, need to be considered as a part of the design system itself. These are the things that enable contributions and manage change. It allows for a challenge to establish patterns as much as it helps roll out work using a raft of solved problems. Acknowledging when decisions will lead to technical design debt and being clear on what level of debt is acceptable might not be something that's clear from the beginning, but discussing it helps with making informed decisions.

Governance, workflow, communication, and community are all really important and, more often than not, need to be considered as a part of the design system itself.

Some form of design system might be a part of your work, from a freelancer to a massive multi-department organization, so your mileage may vary, but hopefully you'll look a little further when starting a design system. The aspects around the community and the kinds of debt you can accrue might resonate across all kinds of systems.

Like many things, a design system isn't ever a finished thing – it's a journey. How we go about that journey can affect the things we produce along the way. While we've learned a lot, there's still a lot further to go. There will always be new challenges, and change is good. As Ryan DeBeasi said in his article:

> *A design system isn't just code, or designs, or documentation. It's all of these things, plus relationships between the people who make the system and the people who use it*
> *— Ryan DeBeasi, "Design Systems Are About Relationships"*[6]

Auto Trader Key Takeaways

A well-executed design system can significantly enhance the developer experience by increasing efficiency and ensuring consistency across products.

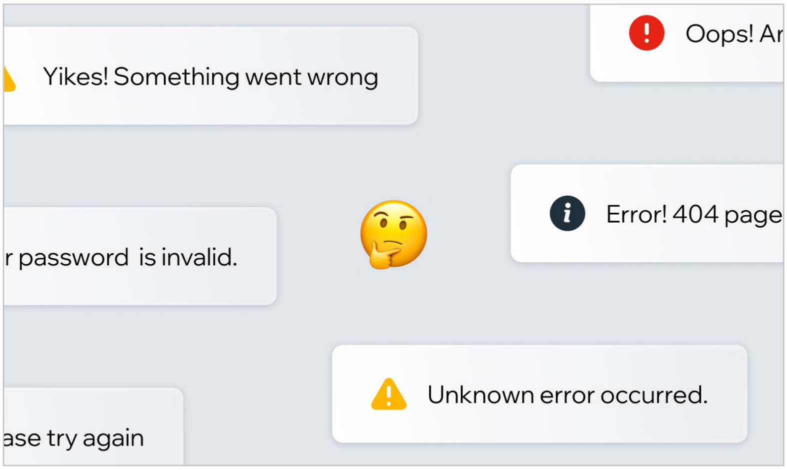
- A design system helps maintain visual and functional consistency across the product suite.
- Developers can work more efficiently by reusing components and styles, reducing duplicate efforts.
- Regular audits and maintenance of the design system are vital to keep it relevant and useful.
- A design system should be accompanied by comprehensive documentation and style guides.
- The design system should be built with scalability and future growth in mind, allowing easy addition and modification of components.

6 https://smashed.by/relationships

Wix: When Life Gives You Lemons, Write Better Error Messages

By Jenni Nadler

Error messages are part of our daily lives online.[1] Every time a server is down or we don't have an internet connection, or we forget to add some info in a form, we get an error message. "Something went wrong" is the classic. But what went wrong? What happened? And, most importantly, how can I fix it?



We encounter error messages all the time, but how often do they actually help us understand what went wrong and how to fix it?

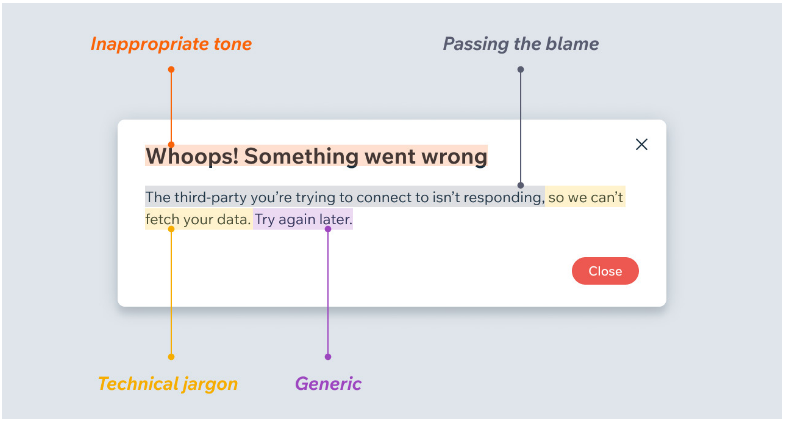
About a year ago at Wix, we abruptly realized that too often we were not giving users the answers to these questions. When we got this wake-up call, we felt compelled to act swiftly, and not just to address the one error message that woke us up.

Welcome, folks, to Errorgate 2021. Or, that time we changed thousands of error messages across Wix in just a month.

1 The original version of this case study was published September 2022: https://smashed.by/bettererrormessages

To complete this effort, we first had to define what counted as a bad error message and what counted as a good error message.

What Makes a Bad Error Message



Whoops! Something went wrong

The third-party you're trying to connect to isn't responding, so we can't fetch your data. Try again later.

This is an example of a bad error message. It uses an inappropriate tone, passes the blame, speaks in technical jargon, and is too generic.

Inappropriate tone: Imagine a doctor performing a procedure and then suddenly saying "Oops! Something went wrong." That is the last thing anyone wants to hear when the stakes are high, whether it's surgery or someone's source of income. That is not the time to be cutesy or fluffy. We want to show the users that we know it's serious and we understand it's important to them.

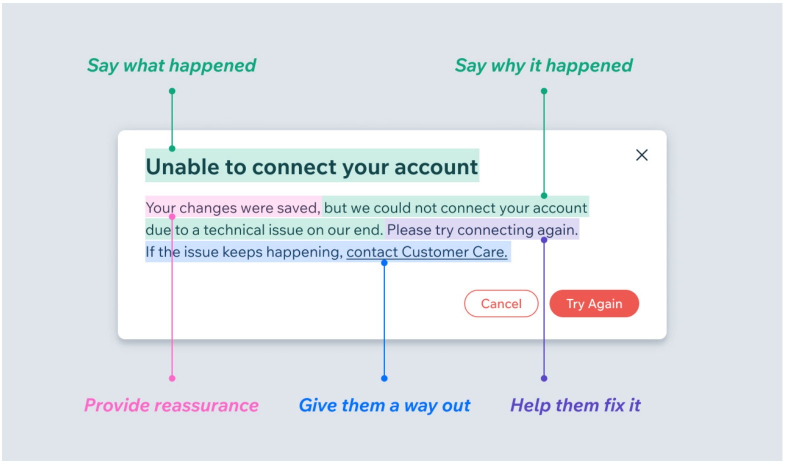
Technical jargon: Even in today's world of user-centered design, technical jargon still sneaks its way into error messages. You couldn't fetch my data? My credentials were denied? What? The technical stuff is not important to the user; they just want to know what went wrong and how to fix it.

Passing the blame: Try to focus on the problem, rather than the action that led to the problem. We don't want to shame users, even if something they did is why they're seeing a certain error message.

We also made the decision not to pass blame on to third parties because it makes us look unprofessional, even if it would have taken some of the burden off of Wix. The user came to Wix as a trusted platform; they don't want to think about other platforms. While we can say something like, "We're having trouble connecting to Z", we wouldn't say something like, "Z isn't responding right now."

Generic for no reason: Sometimes we don't know what caused the error… and sometimes we do. If we know what caused it and we're not telling them, we're doing our users the ultimate disservice.

What Makes a Good Error Message



Unable to connect your account

Your changes were saved, but we could not connect your account due to a technical issue on our end. Please try connecting again. If the issue keeps happening, contact Customer Care.

This is an example of a good error message. It explains what happened and why, provides reassurance, is empathetic, helps the user fix the issue, and gives the user a way out.

Say what happened and why: Make it super clear what did or didn't happen. This can be done with a combination of visuals and text. Explain why the user got this error, even if the only explanation is that there was a technical issue. At Wix, we made the decision to say "an issue on our end" if we have the space, to really reiterate that it's not the user's fault.

Provide reassurance: Where possible, let them know what was not affected by the error. For example, were their changes still saved as a draft, even though their email wasn't sent?

Be empathetic: While we don't want to be overly apologetic, we decided that we did still want to use "please" if the situation warrants it. Maybe it's a really dire situation, or it's something that we absolutely can't help the user solve. In that case, we might use "please" to empathize even more.

Help them fix it: Tell them exactly what to do if there's a way to possibly fix it. Short on space? Send them to a knowledge base article with a descriptive link like, "Learn how to resolve this" or "How do I fix this?"

Always give a way out: If they can't fix the problem, or if it's possible the issue could keep happening, provide them with a way to contact customer care.

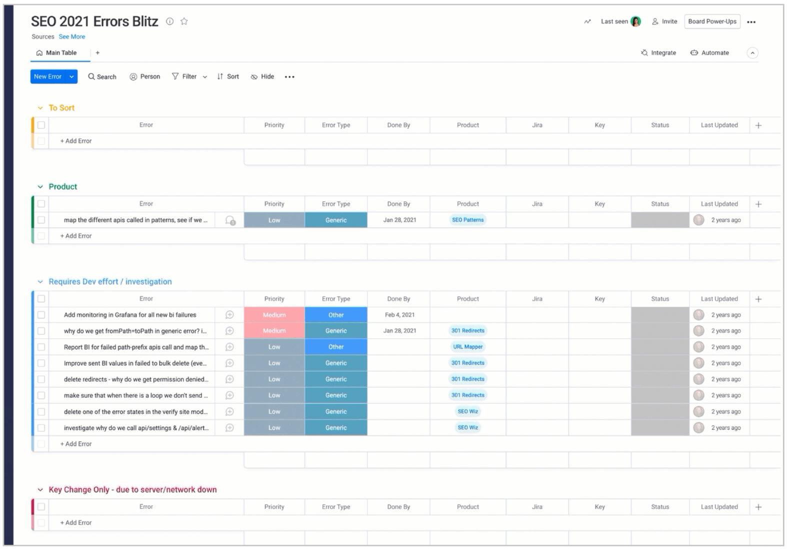
Now that we had defined what made a good or a bad error message, we had to start getting rid of the bad ones.

How We Tackled Removing Bad Error Messages

We searched our content management system and found that there were 7,643 keys with the word "error" in the key or value. That's 7,643 pieces of content that – at the very least – needed to be reviewed.

The task seemed monumental.

But we did it. We reviewed every single piece of content related to errors and decided if it was relevant for this effort. Once we had a list of all the errors we considered "generic" or "not helpful", we sent everything to developers.



SEO 2021 Errors Blitz

Sources See More

To Sort

Error	Priority	Error Type	Done By	Product	Jira	Key	Status	Last Updated
+ Add Error								

Product

Error	Priority	Error Type	Done By	Product	Jira	Key	Status	Last Updated
map the different apis called in patterns, see if we ...	Low	Generic	Jan 28, 2021	SEO Patterns				2 years ago
+ Add Error								

Requires Dev effort / investigation

Error	Priority	Error Type	Done By	Product	Jira	Key	Status	Last Updated
Add monitoring in Grafana for all new bi failures	Medium	Other	Feb 4, 2021					2 years ago
why do we get fromPath=toPath in generic error? i...	Medium	Generic	Jan 28, 2021	301 Redirects				2 years ago
Report BI for failed path-prefix apis call and map th...	Low	Other		URL Mapper				2 years ago
Improve sent BI values in failed to bulk delete (eve...	Low	Generic		301 Redirects				2 years ago
delete redirects - why do we get permission denied...	Low	Generic		301 Redirects				2 years ago
make sure that when there is a loop we don't send ...	Low	Generic		301 Redirects				2 years ago
delete one of the error states in the verify site mod...	Low	Generic		SEO Wiz				2 years ago
investigate why do we call api/settings & /api/alert...	Low	Generic		SEO Wiz				2 years ago
+ Add Error								

Key Change Only - due to server/network down

Error	Priority	Error Type	Done By	Product	Jira	Key	Status	Last Updated
+ Add Error								

This was just one of the Monday.com boards that we used to categorize every single piece of content related to errors. Boards like these helped us set priorities and due dates, and keep all disciplines in the loop.

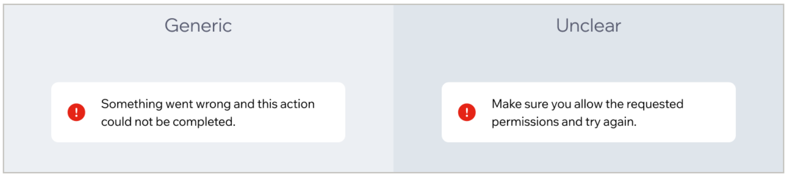
Developers went message by message and mapped where each was being triggered in the code. They looked at what was causing the message to show, how frequently it occurred, and what could be done to resolve the issue.

Based on that error mapping, the product managers, UX designers, and writers sat down and came up with solutions. We started by transferring everything from a spreadsheet to a Monday board, where we could easily track the status of things and what needed to be done. Sometimes, it was just a simple content change. In other cases, it required brand new error messages. And in lots of other instances, there was additional development work that needed to be done to fix things behind the scenes.

Then we prioritized which errors to work on first. To set priorities, we focused on how often the error was happening and if it blocked the user from completing the flow. After that, we set milestones of one to four weeks so that things didn't fall by the wayside.

What We Learned

There's a difference between generic and unclear messages. While there were certainly a lot of generic "Something went wrong" messages, there were also a lot of unclear messages. These are just as bad as generic messages and deserve the same amount of attention.



Something went wrong and this action could not be completed.

Make sure you allow the requested permissions and try again.

An example of a generic message compared to a message that is unclear. In the generic message, we're simply not telling the user anything other than something went wrong. In the unclear message, we tried to explain what went wrong, but it used confusing language.

It's not a content issue most of the time. Avishai Abrahami, our CEO and the reason this project got started, put it best in his email to all employees.

> *"Generic errors are the result of bad development and product. We must all care about it together."*

Truly everyone in Wix had to come together across all disciplines to fix these messages. Developers had to investigate and map. Product managers had to prioritize and create tasks. Designers had to provide new designs for new flows. And we, the UX writers, had to write and rewrite thousands of error messages.

We should be asking more questions. It used to be really common for a developer to say to us, "Hey, we need a generic error message here. Can you add one?" And we would say yes, thinking it would be a fallback or rare message. We didn't often stop to ask questions like, "Why are users seeing this?" and "What is happening in the background?"

We missed a learning opportunity. Unfortunately, we were reactive instead of proactive here. If this effort had been strategically planned, it could have been an amazing learning opportunity for junior writers in particular. Instead, we were scrambling to write and rewrite messages without much strategic thought.

We were being a bad friend. At Wix, we have the mantra, "Write it like you're talking to a friend." We really believe in empathizing with the user, and being a friend with them throughout their process. But it turns out that we were more like that friend who loves to gossip but doesn't pick up the phone when life gets hard. That is not the friend we want to be, so we had to really dig deep and admit that we weren't doing the best we could.

When we work together, we build better products. It's cheesy, but it's true.

What We've Changed in Our Process

Established a cross-functional team to focus on error handling. This team is made up of senior product managers, front-end and back-end developers, UX designers, and UX writers. Their goal is to make sure proper error handling is part of the product life cycle, not an afterthought.

View it as a shared responsibility. Everyone is responsible for making sure we're handling errors properly. Product managers are expected to place more emphasis on errors and edge cases, not just happy flows. Developers are expected to investigate and document errors according to platform-specific guidelines. Data scientists are expected to do better analysis on errors so we can track the events properly.

Wix Key Takeaways

Effective error handling requires clear, empathetic, and actionable error messages, and it's a collaborative effort that involves the entire team to enhance user experience.

- Avoid bad practices in error messages: Bad error messages use inappropriate tone, technical jargon, pass blame, or are too generic. These practices can confuse or frustrate users.
- Characteristics of good error messages: Good error messages explain what happened and why, provide reassurance, display empathy, help users fix the issue, and offer a way to contact customer care if needed.
- Cross-functional collaboration: Changing thousands of error messages at Wix required collaboration across disciplines, including developers, product managers, designers, and writers.
- Learning from mistakes: The reactive approach to changing error messages was a missed learning opportunity. Being proactive and strategic could have provided valuable experience, especially for junior writers.
- Ongoing review and empowerment: Wix established ongoing review processes and empowered UX writers to challenge generic errors, viewing error handling as a shared responsibility and part of the product life cycle

Review errors one month after launch. Sometimes, especially with a brand new product, we don't even know what errors to expect. So we might have to launch with generic errors, but now we have a procedure where we review the errors occurring one month after launch. This allows us to see what really are the biggest errors and write content specifically for those.

Ongoing review process. As writers, we know everything can always be optimized. So we're constantly reviewing our errors, even the ones we just updated recently.

UX writers are empowered to challenge generic errors. In case a product manager or developer ever says, "Let's just use this generic error message in all cases," we now have the power to say no. The CEO of the company has said generic errors are not acceptable, so we're not going to write them without more investigation and understanding of the problem. The power lies with us!

As writers, we know everything can always be optimized. So we're constantly reviewing our errors, even the ones we just updated recently.

All in all, we changed thousands of error messages by working together with our colleagues. It was hard work and we all had a drink or two at the end of it. But it was the right thing to do for our users, and the only way to truly live up to our value of putting the user first.

The world is a miracle. So are you.
Thanks for being smashing.
SMASHING MAGAZINE

"It's rare to find one resource with this many real-world case studies. I highly recommend the book for any web developer. A true gem!"

– **Ahmad Shadeed**, Design Engineer

SUCCESS AT SCALE

is a curated collection of case studies from successful large-scale web projects. Discover practical takeaways and insights to achieve great results for projects large and small.

ACCESSIBILITY
Provide an inclusive web experience.

DEVELOPER EXPERIENCE
Create a culture where people and projects thrive.

CAPABILITIES
Build reliable, installable, feature- rich applications.

PERFORMANCE
Optimize and sustain high site speeds.

Addy Osmani is an engineering leader working on Google Chrome. He leads up Chrome's Developer Experience organization, helping reduce the friction for developers to build great user experiences.

SMASHING MAGAZINE

9 783910 835009