# 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.

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.


Please send errors to: errata@smashingmagazine.com

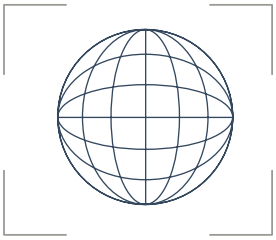
*Dedicated to the brilliant minds moving the web forward.*

*You inspire us every day.*

# CONTENTS

Case studies and interviews
with the people who made it happen

This Table of Contents represents the print edition. An expanded PDF with even more case studies is available free to everyone who purchases the print book!



# 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. The Instagram case study provides an example of using preloads for dynamically loaded JavaScript and xhr GraphQL requests for data (pp. 20 – 36).

**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). The Wix Accessibility case study discusses using ARIA roles and attributes to make components accessible to screen readers (pp. 148 – 160).

**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). The Smashing Core Web Vitals case study discusses using the Chrome User Experience Report (CrUX) to gather field data on site performance metrics from real users (pp. 37 – 60). The eBay.com case study looks at how analyzing CrUX reports helped eBay meet performance budgets and increase site speeds (pp. 69 – 78). See examples of the CrUX dashboard (https://smashed.by/cruxlauncher) on page 48.

**CI/CD**: Continuous integration/continuous delivery, a modern software development practice where frequent code changes are introduced, integrated, and deployed, often using automation. The Facebook case study discusses implementing CI/CD and testing processes while transitioning to a new tech stack (pp. 223 – 240)

**GraphQL**: A query language for APIs and a runtime for fulfilling those queries with your existing data (https://graphql.org/). The Facebook case study looks at transitioning data fetching to Relay and GraphQL while rebuilding the Facebook.com front-end (pp. 223 – 240).

**Lazy loading**: A strategy to load noncritical resources only when they are actually needed. The Instagram case study provides an example of lazy loading images that are out of view using `requestIdleCallback` (pp. 20 – 36).

**Lighthouse**: An open-source diagnostic tool from Google Chrome for improving the performance, quality, and correctness of your web apps (https://smashed.by/lighthouse). The Smashing Magazine case study focuses extensively on using Lighthouse to audit web performance and identify areas for improvement (pp. 37 – 60).

**Next.js**: A React framework for creating full-stack web applications at scale (https://nextjs.org/). Frameworks are discussed throughout the DevEx section beginning on page 215.

**NPM**: Package manager for Node.js (https://www.npmjs.com/). The Bloomberg case study discusses challenges with dependency management at scale and enforcing consistency of npm packages (pp. 241 – 263). BundlePhobia is a tool that provides the cost (in MBs) of adding an npm package to your bundle (https://bundlephobia.com/).

**Performance tools**: Tools such as Lighthouse used to measure the performance of web sites. Other common tools: WebPageTest, PageSpeed Insights, Google Search Console. The Smashing Magazine case study mentions various performance tools like Lighthouse, CrUX, and web-vitals to measure and improve site speed (pp. 37 – 60).

**PWA**: Progressive web applications that can provide an app-like experience on the web (https://smashed.by/pwasguide). PWAs are discussed extensively in the Capabilities section starting on page 87, but more specifically in the Pinterest (p. 103) and Excalidraw (p. 120) case studies.

**React**: Popular library for building web and native user interfaces (https://reactjs.org/). The Spotify case study discusses rebuilding Spotify's web player using React and the developer experience benefits (pp. 110 – 119).

**React Native** (https://reactnative.dev/) is used for developing native applications for Android/iOS devices. The Photoshop case study looks at porting Photoshop to the web, including using React Native for building native apps (pp. 95 – 102).

**Service workers**: Proxy servers that sit between web applications, the browser, and the network (when available) (https://smashed.by/serviceworkersprimer). Both the Photoshop (p. 94) and Excalidraw (p. 120) case studies include specifics on the use of service workers.

**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. The Figma case study discusses optimizing performance for Figma's comments SPA (pp. 61 – 68).

**TypeScript**: A language that extends JavaScript by adding types to the language (https://www.typescriptlang.org/). The Bloomberg case study provides a detailed look at lessons from adopting TypeScript across Bloomberg's massive codebase (pp. 241 – 263).

**WCAG**: Web Content Accessibility Guidelines 2.2 (https://smashed.by/wcagquickref). The entire Accessibillty section – beginning on page 137 – is full of references to WCAG, but The LinkedIn case study specifically mentions accessibility testing using WCAG 2.0 guidelines (pp. 161 – 174).

**Web app manifest**: A JSON file where developers can specify the behavior for the PWA after it's installed on a device. Mentioned in more detail in Excalidraw case study (pp. 120 – 134)

**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) - to be replaced with Interaction To Next Paint (INP), and Cumulative Layout Shift ( CLS). Other diagnostic metrics that may be helpful are Time To First Byte (TTFB), First Contentful Paint (FCP), Time To Interactive (TTI), and Total Blocking Time (TBT). The Smashing Magazine case study focuses on identifying and optimizing Web Vitals metrics using Chrome UX Report and Lighthouse (pp. 37 – 60).

**Workbox**: Production-ready service worker libraries and tooling. (https://smashed.by/workbox). The Instagram case study demonstrates using Workbox to control service worker caching strategies (pp. 20 – 36).


PERFORMANCE
001420
000400
000100
1
2
3

PERFORMANCE

# 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.

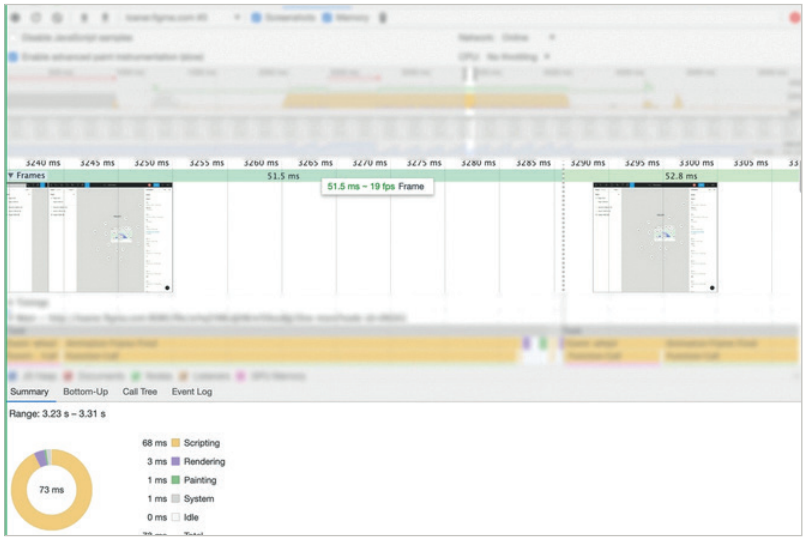
## Getting to the Bottom of Slow Performance

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

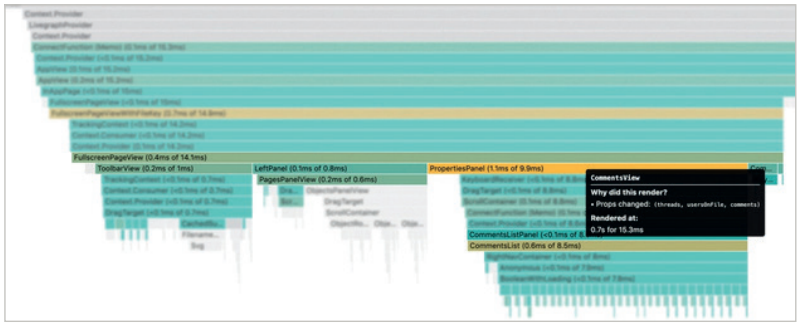
The profile generated from the Chrome performance tools shows that most of the time was spent on JavaScript (JS). About 68 ms per frame is spent on JS on a page with 30 comments, and only a small portion of the computing time per frame is spent on rendering and painting. Scripting refers to JS events 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



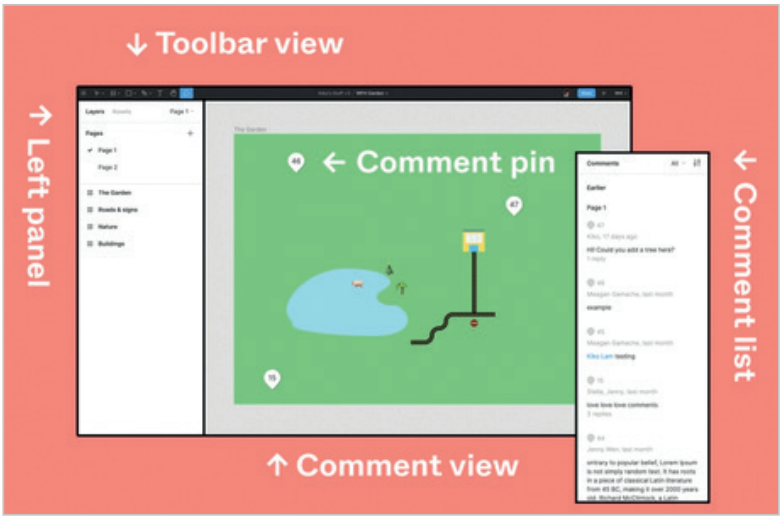

*Chrome's performance tool showed we spent the majority of time on scripting and rendered the comments view at 19 fps with 30 comment threads.*

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




*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

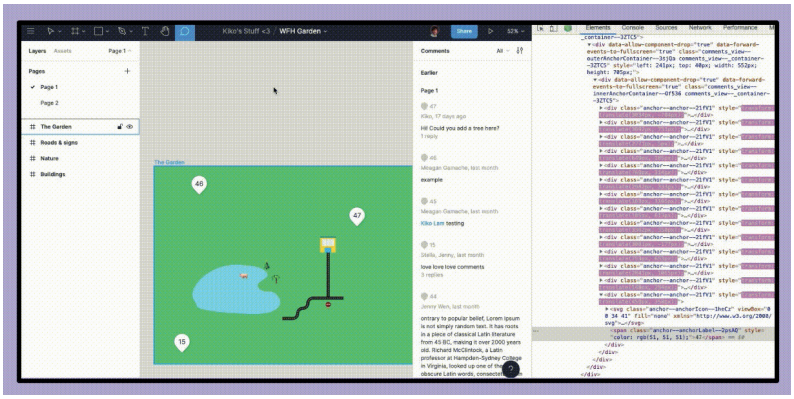
At this point, we ran the Chrome performance tool and React Profiler again. We saw that the constant re-rendering had stopped and the frame rate of the comment view had significantly improved from 15 fps to 50 fps with 50 comment pins. However, we still weren't quite at our goal of 60 fps. We also observed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

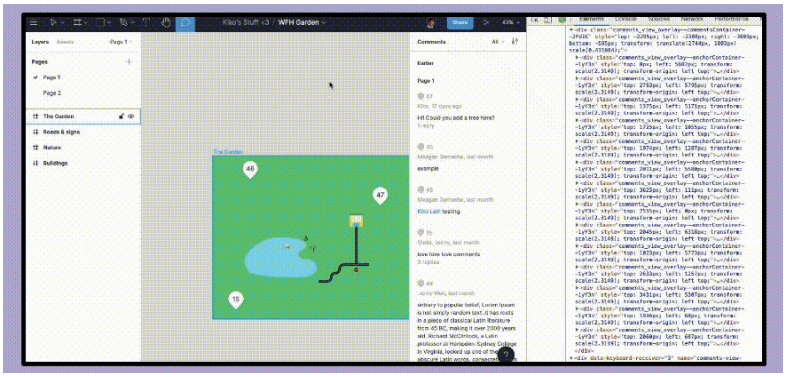
5 https://smashed.by/eventemitters

*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.



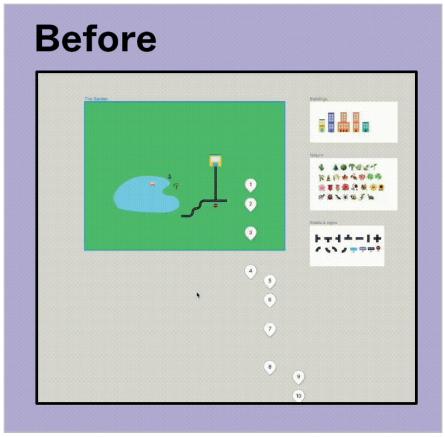
*With every viewport change, each of the comment pin components recomputed its pin position and performed a* `transform-style` *action.*

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.



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

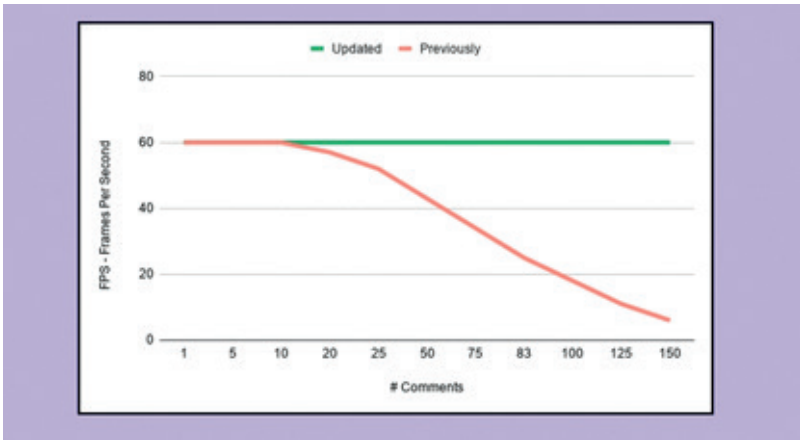
We created this overlay container by creating a box around the most top-left pin and the most bottom-right pin. This means every time a new comment is added, we have to recompute this top-left/bottom-right boundary box. This trade-off is worth it because: a) comments are added less often than panning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




*Panning is much smoother now..*

## 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.


1
2
3
1
2
3
CAPABILITIES

CAPABILITIES

# 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]

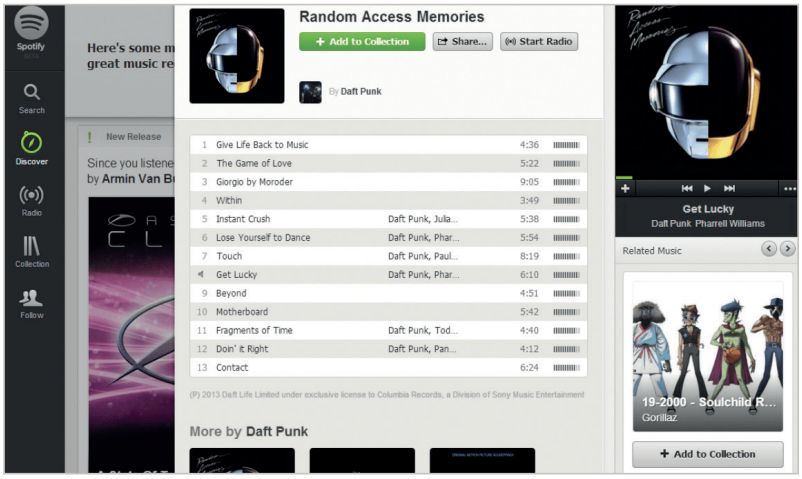
## Using the Web to Implement Spotify Applications at Spotify

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




*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.

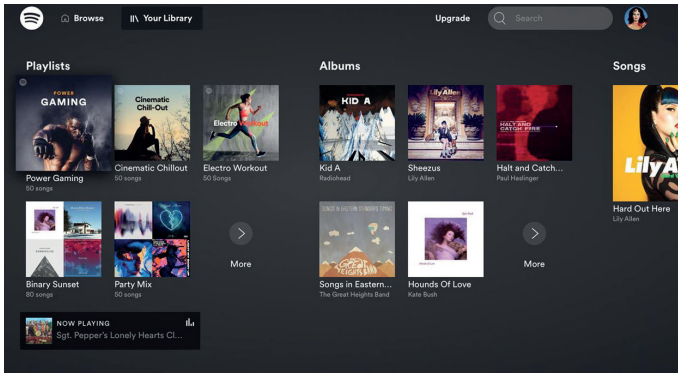
The architecture of the web player was ideal for consistency between platforms, and fit how the company was organized in feature teams. It also had its drawbacks.

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:

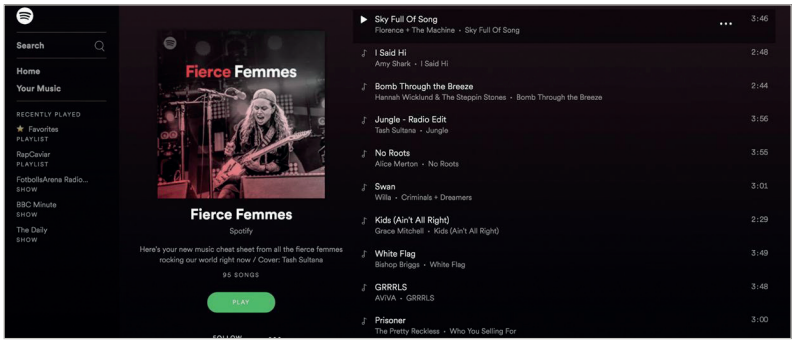
- The system to deliver the code for the views, which worked in isolation from one another, wasn't used by the desktop application anymore, and it was too complex for the web player use case.
- The web player was based on lots of libraries and frameworks that were quite outdated. Giving every team an 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-




*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. Us-

age 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.


1
2
3
ACCESSIBILITY

ACCESSIBILITY

# 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.[2] 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

2 I use the phrase "disabled person" to describe myself, but you may prefer "person with a disability."

dynamic and situational, and impacts every disabled person's experiences differently.

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.

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

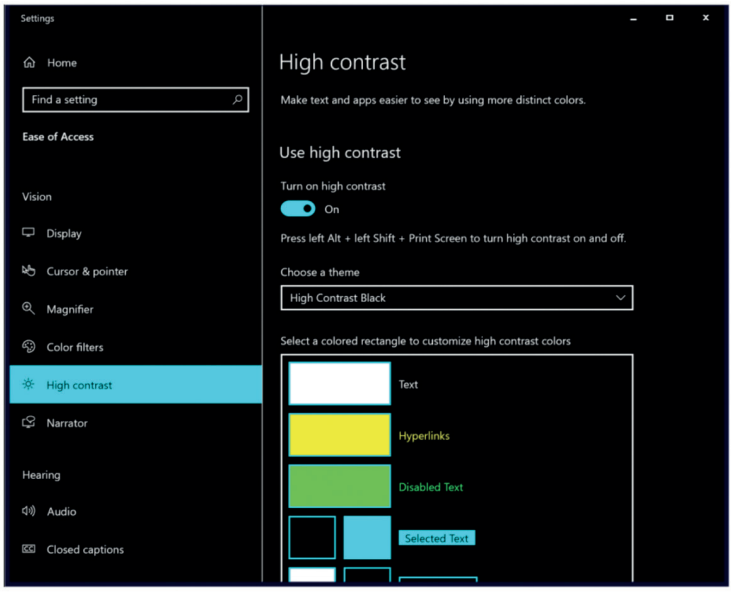
One of the most common categories of assistive tech is for people with dexterity and mobility issues. These might be caused by an injury (even temporary ones), limb difference, paralysis, or chronic pain. (Over 22% of Americans have arthritis, fibromyalgia, or similar chronic pain disorders,[9] so chronic 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.




*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.

> One option is to build your own "virtual" assistive tech lab based on the tech that your customers (or potential customers) are probably using.

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

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.

resources and standards around accessibility are technical and oriented towards testing. And, since so much accessibility work focuses 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.

11 https://smashed.by/wcag
12 https://overlayfactsheet.com/

Instead, try:

- Holistic, continuing education about how disability and technology intersect.
- 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.



## 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.

### 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.

... adding accessibility to project requirements is also a huge step. This allows teams to formally acknowledge wins at demos, town halls, or whatever other rituals your organization has around your workflows and processes.

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.

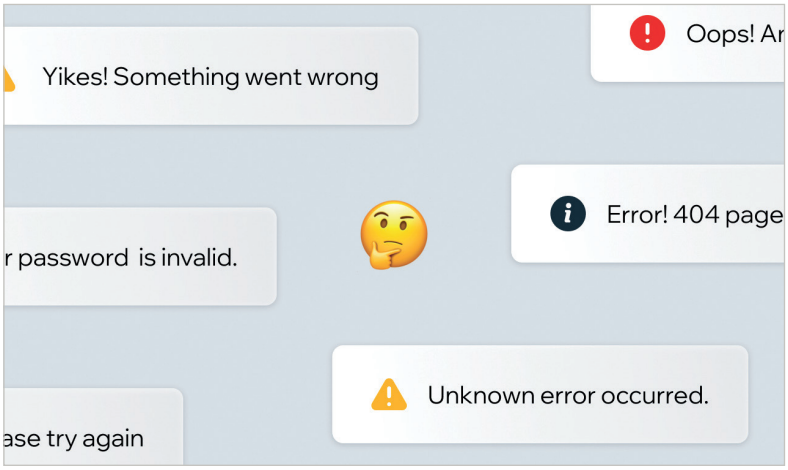
Accessibility should not be a future goal. Start now. Aim to become literate in accessibility, not an expert, and your users and products will benefit exponentially from the experiences you design and consistently improve.


097
1
2
3
1
2
3
DEVELOPER EXPERIENCE

# 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?*

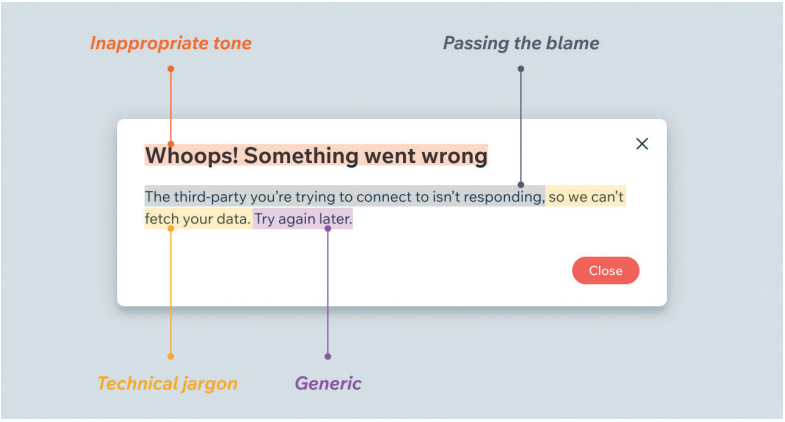
About a year ago at Wix, we abruptly realized that too often we were not giving users the answers to these questions. When we got this wake-up call, we felt compelled to act swiftly, and not just to address the one error message that woke us up.

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




*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.

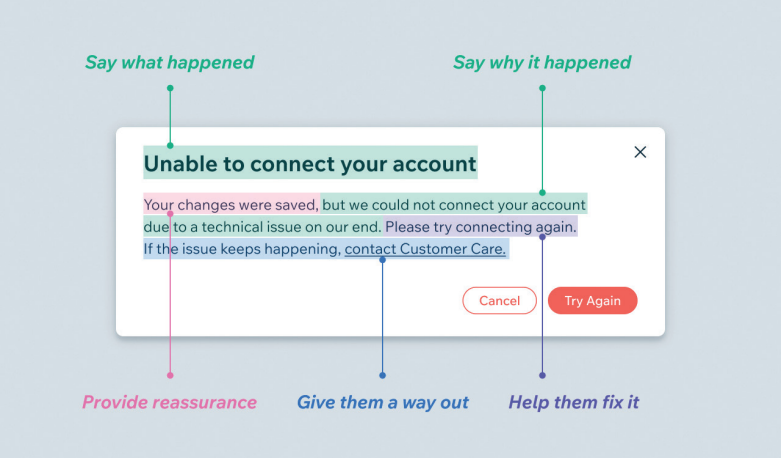
**Technical jargon**: Even in today's world of user-centered design, technical jargon still sneaks its way into error messages. You couldn't fetch my data? My credentials were denied? What? The technical stuff is not important to the user; they just want to know what went wrong and how to fix it.

**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




*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.

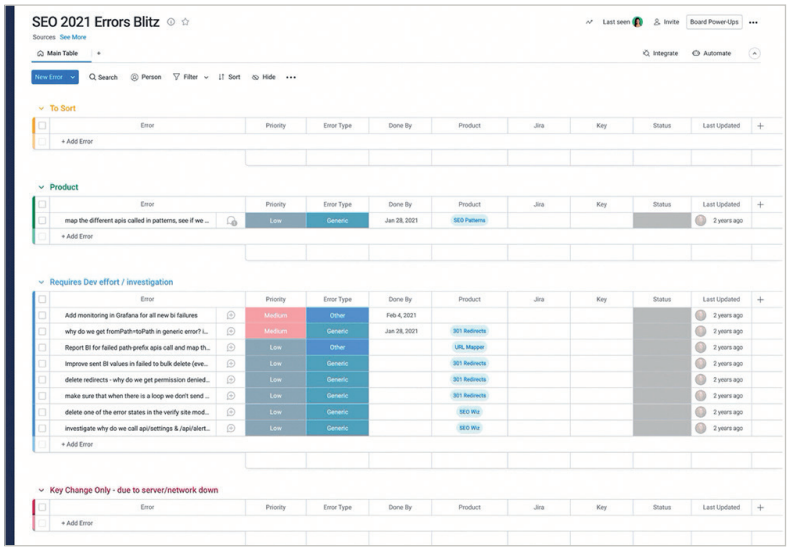
Now that we had defined what made a good or a bad error message, we had to start getting rid of the bad ones.

## 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.




*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.*

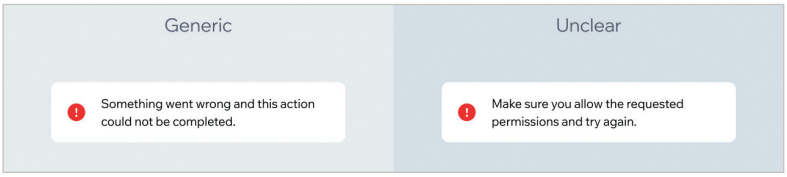
Developers went message by message and mapped where each was being triggered in the code. They looked at what was causing the message to show, how frequently it occurred, and what could be done to resolve the issue.

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.




*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.

SMASHING MAGAZINE

# Smashing Library

Expert authors & timely topics for truly **Smashing Readers**.

## Our Latest Books

Crafted with care for you, and for the Web!


**Understanding Privacy**

by Heather Burns


**Touch Design for Mobile Interfaces**

by Steven Hoober


**TypeScript in 50 Lessons**

by Stefan Baumgartner


**Image Optimization**

by Addy Osmani


**The Ethical Design Handbook**

by Trine Falbe, Martin Michael Frederiksen and Kim Andersen


**Click!**
**How to Encourage Clicks Without Shady Tricks**

by Paul Boag

See all of our titles at smashed.by/library

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