Conformance for Frameworks

Conformance in the JavaScript framework ecosystem

Published on Tuesday, June 15, 2021

Shubhie Panicker
Tech Lead Manager on the Chrome Web Platform team

Houssein Djirdeh
Developer Advocate

Table of contents

In our introductory blog post, we covered how we've learned a lot while building and using frameworks and tools to develop and maintain large scale web applications such as Google Search, Maps, Photos, and so on. By safeguarding developers from writing code that can negatively affect user experience, we proved that frameworks can play a key role in shifting outcomes for performance and application quality.

Internally at Google, we used the term "**Conformance**" to describe this methodology, and this article covers how we plan to open-source this concept to the JavaScript framework ecosystem.

What is Conformance?

At Google, Conformance was an evolution. Teams relied on a small set of deeply experienced maintainers who did extensive code reviews, flagging things that impacted app quality and maintainability well beyond correctness issues. To scale this to growing teams of app developers, a Conformance system was developed to codify best practices in a way that is automated and enforceable. This ensured a consistently high bar for app quality and codebase maintainability regardless of the number of code contributors.

Conformance is a system that ensures that developers stay on the well-lit path; it builds confidence and ensures predictable outcomes. It makes teams productive, and becomes crucial for *scale* -- as teams grow and more features are developed simultaneously. It empowers developers to focus on building product features, freeing them from minutiae and the changing landscape in various areas such as performance, accessibility, security, etc. Anyone can opt-out of Conformance at any time, and it should be customizable to the extent that teams will have the option to enforce whatever they decide to commit to.

Conformance is founded on **strong defaults** and providing **actionable rules** that can be enforced at **authoring time**. This breaks down into the following 3 principles.

> Conformance in this article focuses on coding best practices to achieve predictable loading performance & Core Web Vital scores, but the principles apply equally to other aspects such as security, accessibility, and so forth.

1. Strong defaults

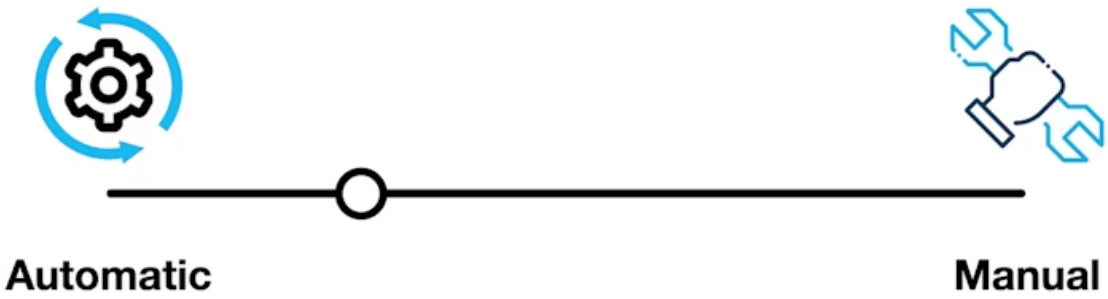
A foundational aspect of conformance is ensuring that the tools developers use have strong defaults in place. This means solutions are not only baked into frameworks, but also framework design patterns make it easy to do the right thing and hard to follow anti-patterns. The framework supports developers with application design and code structure.

For loading performance, every resource (fonts, CSS, JavaScript, images, etc.) should be optimized. This is a complex challenge involving trimming of bytes, reducing round trips, and separating out what is needed for the first render, visual readiness, and user interaction. For example, extracting critical CSS, and setting priority on important images.

2. Actionable rules

Even with foundational optimizations in place, developers still have to make choices. There's a spectrum of possibilities for optimizations when it comes to how much developer input is needed:

> Defaults that require no developer input such as inlining critical CSS.
>
> Require developer opt-in. For example, using a framework-provided image component to size and scale images.
>
> Require developer opt-in and customization. For example, tagging important images to be loaded early.
>
> Not a specific feature but things that still require developer decision. For example, avoiding fonts or synchronous scripts that delay early rendering.



Optimizations that require any decision by developers pose a risk to the performance of the application. As features are added and your team scales, even the most experienced developers cannot keep up with the constantly changing best practices, nor is it the best use of their time. For Conformance, appropriate actionable rules are as important as strong defaults to ensure that the application continues to meet a certain standard even when developers continue to make changes.

3. Authoring time

It's important to catch and *preclude* performance problems early in the development lifecycle. Authoring time, before code is committed, is ideally suited to catching and addressing problems. The later a problem is caught in the development lifecycle, the harder and more expensive it is to address it. While this applies to correctness issues, it is also true for performance issues, as many of these issues will not be retroactively addressed once committed to the codebase.

Today, most performance feedback is out-of-band via documentation, one-off audits, or it is surfaced too late via metrics regression after deployment to production. We want to bring this to authoring time.

Conformance in Frameworks

To maintain a high bar of user experience for loading performance, the following questions need to be answered:

1. What constitutes optimal loading, and what are the common issues that could impact it adversely?
2. Which solutions can be baked in that do not need any developer input?
3. How can we ensure that the developer uses these solutions and leverages them optimally?
4. What other choices could the developer make that impact loading performance?
5. What are the code patterns that can tell us about these choices (#3 and #4 above) early at authoring time?
6. What rules can we formulate to assess these code patterns? How can they be surfaced to the developer at authoring time while seamlessly integrated into their workflow?

To bring the Conformance model we have internally at Google to open-source frameworks, our team has experimented heavily in Next.js and we are excited to share our refined vision and plans. We've realized that the best set of rules that can assess code patterns will need to be a combination of **static code analysis** and **dynamic checks.** These rules can span multiple surfaces, including:

ESLint

TypeScript

Dynamic checks in the user's development server (post DOM creation)

Module bundler (webpack)

CSS tooling (still exploratory)

By taking advantage of providing rules through different tools, we can ensure they are cohesive but also encompass any user experience issues that directly impact loading performance. Additionally, these rules can also be surfaced to developers at different times:

During local development in the development server, browser and user's IDE will surface warnings, prompting developers to make small code changes.

At build time, unresolved issues will be resurfaced in the user's terminal

In a nutshell, teams will choose outcomes they care about, such as Core Web Vitals or loading performance, and enable relevant rulesets for all code contributors to follow.

While this works really well for new projects, it's not easy to upgrade large codebases to comply with full rulesets. At Google we have an extensive system for opting-out at different levels such as individual lines of source code, entire directories, legacy codebases or parts of the app that are not under active development. We are actively exploring effective strategies for bringing this to teams using open-source frameworks.

Conformance in Next.js

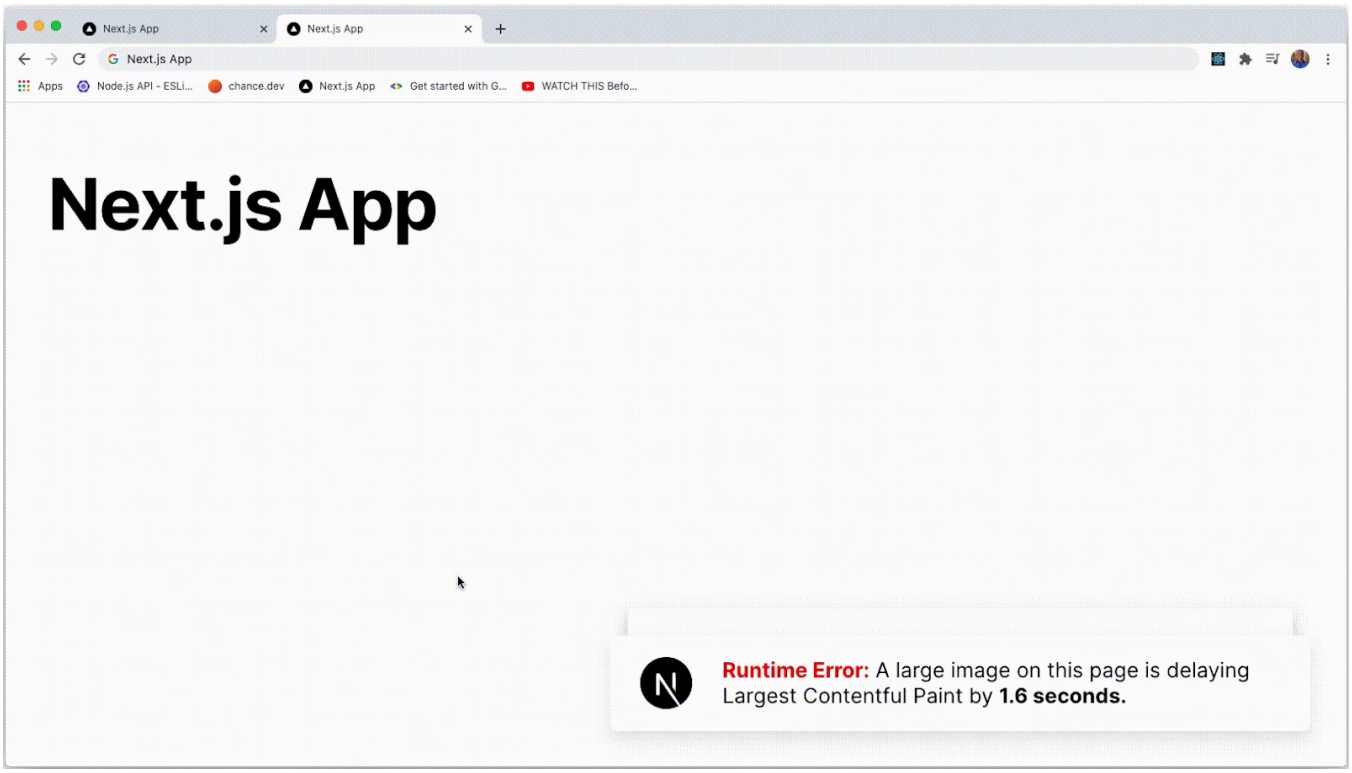
ESLint is widely used among JavaScript developers. and over 50% of Next.js applications use ESLint in some part of their build workflow. Next.js v11 introduced out-of-the-box ESLint support that includes a custom plugin and shareable configuration to make it easier to catch common framework-specific issues during development and at build time. This can help developers fix significant problems at authoring time. Examples include when a certain component is used, or not used, in a way that could harm performance as in No HTML link for page). Or, if a certain font, stylesheet, or script can negatively affect resource loading on a page. For example, No synchronous script.

In addition to ESLint, integrated type-checking in both development and production has been supported in Next.js since v9 with TypeScript support. Multiple components provided

by the framework (Image, Script, Link) have been built as an extension of HTML elements (`<img>`, `<script>`, `<a>`) to provide developers with a performant approach to adding content to a web page. Type-checking supports appropriate usage of these features by ensuring that properties and options assigned are in the acceptable scope of supported values and types. See required Image width and height for an example.

Surfacing Errors With Toasts and Overlays

As mentioned previously, Conformance rules can be surfaced in multiple areas. Toasts and overlays are currently being explored as a way to surface errors directly in the browser within the user's local development environment.



Many error-checking and auditing tools that developers rely on (Lighthouse, Chrome DevTools Issues tab) are passive and require some form of user interaction to retrieve information. Developers are more likely to act when errors are surfaced directly within their existing tooling, and when they provide concrete and specific actions that should be taken to fix the problem.

> Utilizing toasts and overlays as a UI surface for Conformance is still being explored in Next.js.

Conformance in Other Frameworks

Conformance is being explored in Next.js first with the goal of expanding to other frameworks (Nuxt, Angular, etc.). ESLint and TypeScript are already used in many frameworks in many different ways, but the concept of a cohesive, browser-level runtime system is being actively explored.

Conclusion

Conformance codifies best practices into rulesets that are actionable for developers as simple code patterns. The Aurora team has focused on loading performance, but other best practices, such as accessibility and security, are just as applicable.

Following Conformance rules should result in predictable outcomes, and achieving a high bar for user experience can become a side-effect of building on your tech stack. Conformance makes teams productive, and ensures a high quality bar for the application, even as teams and codebases grow over time.

Aurora Project

Podcasts

Connect

Twitter

YouTube

GitHub

Google Developers

Chrome Firebase All products Privacy Terms

ENGLISH (en)