Complexity management

- Managing PWA complexity
  - Separations of concerns
    - Focus on a single task
    - Composability
  - Using web workers to reduce complexity
  - Try it yourself
- Resources

In his talk Simple Made Easy, Rich Hickey discusses the qualities of simple versus complex things. He describes simple things as focusing on:

> "One role, one task, one concept, or one dimension."

But emphasizes that simplicity isn't about:

> "Having one instance or performing one operation."

Whether or not something is simple is about how interconnected it is.

Complexity comes from binding, weaving, or, to use Rich's term, complecting things together. You can calculate complexity by counting the number of roles, tasks, concepts, or dimensions something manages.

Simplicity is essential in software development because simple code is easier to understand and maintain. Simplicity is also necessary for web apps because it may help to make our app fast and accessible in every possible context.

Managing PWA complexity

All JavaScript we write for the web touches the main thread at one point. The main thread, though, comes with a lot of complexity out-of-the-box that you, as a developer, have no control over.

The main thread is:

- Responsible for drawing the page, which itself is a complex, multi-step process involving calculating styles, updating and compositing layers, and painting to screen.
- Responsible for listening to and reacting to events, including events such as scrolling.
- Responsible for loading and unloading the page.
- Managing media, security, and identity. That's all before any code you write can even execute on that thread, such as:
- Manipulating the DOM.
- Accessing sensitive APIs, for example, device capabilities, or media/security/identity.

As Surma put it in his 2019 Chrome Dev Summit talk, the main thread is overworked and underpaid.

And yet, most application code lives on the main thread too.

All that code adds to the complexity of the main thread. The main thread is the only one that the browser can use to lay out and render content on the screen. Therefore, when your code requires more and more processing power to complete, we need to run it quickly, because every second it takes to perform application logic is a second the browser can't respond to user input or redraw the page.

When interactions don't connect to input, when frames drop, or when it takes too long to use a site, users get frustrated, they feel the application is broken, and their confidence in it decreases.

The bad news? Adding complexity to the main thread is an almost sure-fire way to make meeting these goals difficult. The good news? Because what the main thread needs to do is clear: it can be used as a guide to help reduce reliance on it for the rest of your application.

Separations of concerns

There are lots of different kinds of work that web applications do, but broadly speaking, you can break it down into work that directly touches the UI and work that doesn't. UI work is work that:

- Directly touches the DOM.
- Uses APIs that touch device capabilities, for example, notifications or file system access.
- Touches identity, for example, user cookies, local, or session storage.
- Manages media, for example, images, audio, or video.
- Has security implications that would require user intervention to approve, such as the web serial API.

Non-UI work can include things such as:

- Pure calculations.
- Data access (fetch, IndexedDB, etc.).
- Crypto.
- Messaging.
- Blob or stream creation, or manipulation.

Non-UI work is often bookended by UI work: a user clicks a button that triggers a network request for an API that returns parsed results that are then used to update the DOM. When writing code, this end-to-end experience is often considered, but where each part of that flow lives usually is not. The boundaries between UI work and non-UI work are as important to consider as the end-to-end experiences, as they are the first place you can reduce main thread complexity.

Focus on a single task

One of the most straightforward ways of simplifying code is breaking out functions so each focuses on a single task. Tasks can be determined by the boundaries identified by walking through the end-to-end experience:

- First, respond to user input. This is UI work.
- Next, make an API request. This is non-UI work.
- Next, parse the API request. This again is non-UI work.
- Next, determine changes to the DOM. This may be UI work, or if you're using something such as a virtual DOM implementation, it may not be UI work.
- Finally, make changes to the DOM. This is UI work.

The first clear boundaries are between UI work and non-UI work. Then there are judgment calls that need to be made: is making and parsing an API request one task or two? If DOM changes are non-UI work, are they bundled with the API work? In the same thread? In a different thread? The proper level of separation here is key to both simplifying your codebase and being able to successfully move pieces of it off the main thread.

Composability

To break up large end-to-end workflows into smaller parts, you need to think about your codebase's composability. Taking cues from functional programming, consider:

- Categorizing the types of work your application does.
- Building common input and output interfaces for them.

For instance, all API retrieval tasks take in the API endpoint and return an array of standard objects, and all data-processing functions take in and return an array of standard objects.

JavaScript has a structured clone algorithm meant for copying complex JavaScript objects. Web workers use it when sending messages and IndexedDB uses it to store objects. Choosing interfaces that you can use with the structured cloning algorithm will make them even more flexible to run.

With this in mind, you can create a library of composable functionality by categorizing your code and creating common I/O interfaces for those categories. Composable code is a hallmark of simple codebases: loosely coupled, interchangeable pieces that can sit “next” to each other and build on each other, as opposed to complex code that is deeply interconnected and therefore cannot be easily separated. And on the web, composable code can mean the difference between overworking the main thread or not.

With composable code in hand, it's time to get some of it off the main thread.

Using web workers to reduce complexity

Web workers, an often underutilized but widely available web capability, let you move work off the main thread.

Web workers let a PWA run (some) JavaScript outside of the main thread.

There are three kinds of workers.

Dedicated workers, what's most commonly thought of when describing web workers, can be used by a single script in a single running instance of a PWA. Whenever possible, work that doesn't directly interact with the DOM should be moved to a web worker to improve performance.

Shared workers are similar to dedicated workers, except multiple scripts can share them across multiple open windows. This provides the benefits of a dedicated worker but with a shared state and internal context between windows and scripts.

A shared worker could, for instance, manage access and transactions for a PWA's IndexedDB and broadcast transaction results across all calling scripts to let them react to changes.

The final web worker is one covered extensively in this course: *service workers*, which act as a proxy for network requests and are shared between all instances of a PWA.

Try it yourself

It's code time! Build a PWA from scratch based on everything you've learned in this module.

{% Aside 'codelab' %} Progressive Web Apps: Working with Workers {% endAside %}

Resources

- Simple Made Easy (video)
- Using Web Workers

- Workers overview
- Use web workers to run JavaScript off the browser's main thread

---

□□□ #2
□ □□□ □□□ 2026-10-31 22:55:08 UTC
□ □□□ □□□ 2026-10-31 22:55:10 UTC