Why build an imperative Push Messaging API instead of a declarative Push Notifications API?

Author: Google Chrome team
Status: public

TL;DR: declarative APIs are sometimes better for resource usage, but supporting future/advanced use cases requires huge additional work compared to imperative APIs. Furthermore, small design mistakes can nullify entire declarative APIs as we saw with AppCache.

The web platform community developed the **extensible web model** after years of learning from our mistakes. The moral was that exposing **low-level primitives** to a platform is a much better approach than trying to design a high level API which may quickly become outdated as use cases or platforms change. We designed **AppCache** declaratively, **failing to observe these lessons** and have only now solved the problem with the imperative service workers.

Fusing Push and Notifications into a single **declarative API** seems easy but it **provides a poor user experience** even for the most common use case unless many subtle considerations are made. When considering more advanced use cases (including ones that haven't emerged yet) it is clear that we won't get this right on first try, and **second attempts take years** in the world of web standards. Allowing developers to build higher level abstractions on low level primitives solves this problem.

Summary of pros vs cons of an imperative Push API

Pros of an imperative API

- More flexible when dealing with design flaws, missing features, or a change in requirements
- Composable with other APIs, both existing and future ones e.g. Notifications API and IndexedDB API
- Polyfillable (possible for parts of the Push API e.g. the PushMessageData methods)

Cons of an imperative API

- Higher resource usage due to loading a JS interpreter etc (although in this case an imperative Push API was shown to be within acceptable limits in Chrome on Android)
- More complex to use for the simplest use case (though this quickly changes when edge cases need to be handled)

Note we may standardize a declarative API in the future for the most common uses of an imperative API that we see in order to get the resource usage benefits.

Getting the declarative API design right now would be hard/impossible

Even the simplest use case requires many subtle considerations which we could easily get wrong. It is impossible to design a good API at this stage for more advanced use cases like receiving RTC calls and

the cost of developing use-case specific declarative APIs is extremely high both in terms of time and money. Hence we believe an imperative API is the way to go.

Today's primary use case

Today's most obvious use case is delivering a notification because something happened, like a new email. Sounds simple right? It turns out that a declarative API designed to simply allow an app server to trigger a native notification is a bad user experience because payloads aren't large enough (4kb on GCM, 2kb on APNS) to include the updated data the app would need to update its cache. Imagine seeing a notification about a new email and tapping on it whilst experiencing spotty connectivity. The mail client will open but won't be able to find the email, despite having shown a notification about it. This is a very frustrating experience, further more, without a way for the page to signal success/failure, the notification is now gone.

There are other extra complications we need to consider:

- Sometimes webapps shouldn't show a notification in response to a push, for example when a user already has relevant content focused and visible (e.g. their inbox is the active tab).
- We'd like webapps to be able to dismiss a notification on other devices when the user dismisses it on one device.

So now our declarative API needs to support syncing some data when the push message is received, allowing developers to specify some cases where the notification shouldn't be shown, and some way to hide the notification later. This is already getting complicated and it's likely that more edge cases would be discovered in the future just to support the most basic use case. If we missed an edge case it could dramatically reduce the value of the whole API, or give users a subpar experience for years to come.

Using Declarative Push to deliver an RTC call

To support the use case of receiving an RTC call the declarative Push API (or a calling-specific API) would need to support the following flow:

- Open a tab to the calling page if one isn't currently open, else use the existing one
- Focus that tab forcing it over any lock screen (we don't have a way to do this yet)
- postMessage the page telling it who's calling

If the call is answered on a particular device, you'll want to push to other devices to stop "ringing".

That's just one rough design for what may happen but other people may have better ideas (like a richer Notifications API). Now is clearly not the time to immortalize a flow like that in a high level API.

Summary

We don't think it is impossible to build a declarative API to support some basic notifications use cases, but we will undoubtedly get some slightly wrong or forget about some use cases. The approach of requiring the API designers to perfectly predict every use case is not a good one, especially in the slow moving world of web standards where a minor mistake can cost you years.

We believe it is best to follow the extensible web model and expose the primitives, then build higher level APIs based on common patterns that are observed. These high level APIs will undoubtedly provide many benefits like lower resource usage, but we should get the primitives right first rather than getting a declarative API slightly wrong and making the whole thing pretty useless (e.g. AppCache).