Debunking an API Design Meme

There is a meme that I have observed in a number of recent interactions with standards groups and it goes a little like this:

> "The user (application developer) is an idiot and can't be trusted with anything that is sharp or breakable."

Now, to be fair, this is a caricature of that message, which manifests as "inexpert developer" or a similar euphemism. Nonetheless, this meme is a motivator for many design decisions. Placing too much credence in the meme could lead to an imbalance in design that favours more defensive development of protocol and API features, most likely at the cost of flexibility or expressiveness.

This meme again reared its head in discussions on liveness testing for rtcweb. In this case, the meme manifests in an argument like so:

> "We can't let the application developer know about a possible failure in a flow because the outage might be transitory. If the failure is only temporary, notification might cause the developer to behave destructively. For example, they could trigger an ICE restart that could cause more packets to be sent into an unstable network."

Now, it's not entirely clear that this is a specific position that was being advocated [it certainly didn't conclude this way]. It was merely a sentiment that was expressed; a concern. Obviously, if this is the decision that is made, then some information is purposefully being withheld from an application. Some capability is lost to the meme.

In order to redress this imbalance, we first have to separate malicious and inept use. That is, we need to disassociate the harm that an inexpert developer might cause others as opposed to the harm that they cause themselves. The whole reason we have security measures in our solutions is to prevent a malicious actor from causing harm to others. As a consequence, we also protect the inept from doing so. If it were possible for the inept to cause harm, then the malicious need only pretend to be inept. That would be a security failure.

To be perfectly clear, this is only true to the extent that other actors are not asked to make a trust judgment in order to grant certain entities elevated privileges. At that point, an inept actor can be elevated into an engine of destruction far more readily than a malicious one. One great example of this sort of failure can be found here.

As long as we accept that harm is only self-inflicted, what then is our philosophy on self-harm? Obviously, we want to be seen to be stopping self-harm.

In the most extreme cases, this line of reasoning leads to a trade-off of self-harm versus empowerment. That's not constructive. "Think of the children!" arguments are neither helpful, nor rational in a mature debate.

Correction requires first addressing the concern that application developers are idiots. Or - to be fair - given the constraints imposed upon them, application developers are incapable of comprehending the problem space with sufficient nuance to make the right decisions.

The primary problem here is not the application developer, nor the innumerable constraints imposed upon them. These are real problems, but the solution is not mollycoddling. If we are to really solve this problem, we need to talk about communication.

An application developer has incentive to understand, and it would be hard to argue that they lack the capacity to learn. More often it is simply that they have limited pre-existing knowledge in the field and limited time to spend in gaining the necessary knowledge. For something like a Javascript API, we might expect a developer to be well-versed in Javascript, but not the subject of that API.

If it is necessary that a developer understand some critical information, then the failing occurs when that does not happen. It is a failure on the part of the API - and to a lesser extent, its documentation - to convey critical information, not a failing of the developer.

The key realization here is that the failure in the system lies not always with the receiver, but the transmitter. Garbage in, garbage out. There are many ways to structure an API so that it can achieve its technical goals, but a design that conveys purpose and insight is far more useful.

Unless you are privileged enough to have your API become an essential or necessary part of understanding the language, failure at communication is dire. XMLHttpRequest (or XmlHTTPRequest, I can never remember which one of these it is) has that dubious privilege, but even that status does not make it immune to misuse. There are other reasons, but wrappers for XMLHttpRequest are commonplace, and would still be even if they weren't necessary for other reasons. The node.js HTTP API is, in this respect, a far superior design.

In recent years, significant advancements have been made in understanding how design is used to communicate. We are seeing the benefits of that in computing devices due to rapid progress in that industry, but the changes are not limited to that niche. API design can use the same design philosophy as a toaster, a chair, or music player. The design languages might be different, but the underlying motivation is no different.

Names, structure, and even examples are all critical in conveying information about an API. Names are most important. A name that fails to convey its purpose is far more dangerous than an error in documentation. Documentation is secondary. It's not a joke to say "if at first you don't succeed, read the documentation" or "if all else fails, read the documentation".

This is a problem that is likely to occur more and more often. More applications are touching a wider breadth of domains and developers are going to have less time to dedicate to learning. API designs that communicate goals and constraints more effectively are going to be more important. We cannot continue to blame our problems on others when we have the chance to fix them ourselves.

It has been my observation that recent W3C documentation takes an undue focus on prescribing browser behaviour. To a large extent, this is a symptom of massive browser interoperability failures in the early days of the web. The current documentation emphasis certainly achieves interoperability. However, it fails users by burying important information under pages of complicated and irrelevant procedures. Where documentation fails, design becomes even more important.