Implementing a layered web platform

ojan@ 5/4/15

Building an extensible, layered platform for the web has many advantages (e.g. the end result tends to tends to be faster due to fewer special-cases in low-level code, authors get more power and control when they need to dig below the surface layers, etc.). A potential downside to this approach for a multi-vendor platform is that each layer needs to be specified in a way that can be interoperably implemented by different browser vendors. Figuring out the exact right API takes time and is dependent on other vendors prioritizing the same features at the same time.

Goals

- Ensure new high-level features are explainable in terms of primitives exposed to the platform.
- Ensure new low-level features actually explain the high-level features they purport to explain.
- Maintain good layering in the codebase where low-level code does not need to have intimate knowledge of high-level features.

Non-goals

- Exposing a C++ API to web authors. This proposal is entirely about implementation details that web authors should be oblivious to.

This is a proposal to enable shipping new high-level features in a way that gives us high confidence that they can eventually be built entirely on top of lower-level primitives. It gets us a balance of being able to ship faster without baking more magic into the platform that slows us down forever. Specifically, we should **create a C++ API that maps to the primitives we want to expose** (or already have exposed) to the web and **build higher-level APIs on top of them.**

Execution of this approach will be somewhat slower in the short-term, but faster in the long-term partially because we will be forced not to accrue technical debt. In the short-term, some new features would be gated on implementing primitives first. In the long-term, it will keep the whole codebase simple and decoupled in a way that will help our forward velocity. For example, implementing a built-in pull-to-refresh API would be simpler because it wouldn't need to integrate with scrolling code that needs to handle snap-points and all the other scrolling related features.

    - A small example of a case where this sort of simplifications helped our forward velocity was rewriting marquee in Blink-in-JS. Before the rewrite, the marquee code actually had a dedicated renderer subclass that hooked into the system in crazy ways that were hard to reason about. We frequently would try to make changes to unrelated parts of the system that would pass all the tests (because marquee was barely tested) and then get reverted down the line because they'd

break real content. Now we can move forward never needing to think about marquee and not needing to do the land-revert-land dance that is such a waste of everyone's time.

Despite all of Shadow DOM's struggles in the spec world, one of it's big successes was that we started with reimplementing Blink's form controls on top of Shadow DOM. In that case, we created some backdoors to get the thing to ship that have come back to haunt us. For example, not being forced to go through the exposed API surface, we never needed to worry about host styling -- we just created a new type of a RenderObject, like RenderVideo. This later was discovered by Polymer folks, who wanted to do the same, but couldn't. So :host was born.

Proposal

1. Create an auto-generated gen/webapi directory from our regular, web-exposed IDL files that exposes C++ APIs. For example, Element.idl would end up generating an gen/webapi/.../Element.cpp file that exposes the same API to C++ that we expose to JavaScript. That way we expose to our internal C++ code the same API that we expose to JavaScript.
2. Create a webmodules directory. This directory contains only higher-level APIs that are built on other primitives in the platform (e.g. mouse events can be built on top of pointer events). Use the appropriate guards so that this directory can only include files from gen/webapi and other webmodules. This is different from the modules directory, which can depend on core directly.

Lifecycle of a feature

1. Implement the primitives (in core or webmodules as appropriate) you need and expose them via a flag (e.g. --experimental-web-platform-features) as if we were going to ship them.
2. Expose those primitives in the webapi directory (not behind a flag).
3. Implement your high-level feature on top of the new primitives (in core or webmodules as appropriate).

Example 1: PointerEvents

We've decided to ship pointer events, but we don't forever want to be stuck with 3 event models on the web. We justify this by saying that you can implement mouse and touch events on top of pointer events, but invariably, it's the edge cases that you only run into via implementation that expose the areas where this isn't true. Thankfully, pointer events is still open to API changes, so as we run into problems here, we might be able to get them fixed.

The implementation strategy here would be:

- Bake pointer events into the Blink core code
- Expose pointer events to C++ via gen/webapi
- Re-implement mouse/touch events on top of pointer events in the webmodules directory
- Delete the old mouse/touch code from core

Example 2: snap-points

Every other browser vendor will soon be shipping snap-points. Even though we strongly believe that scroll customization is the feature we really care about (e.g. you could build snap-points on top of scroll customization, CompositorWorker, custom layout and custom CSS properties), we may need to ship snap-points before we can get agreement on and implement the lower-level APIs.

The implementation strategy here would be:

- Implement CompositorWorker in core
- Implement scroll customization in core
- Implement custom layout in core
- Implement custom CSS properties in core
- Expose the above to C++ via gen/webapi
- Implement snappoints on top of the above in webmodules

Autogenerated code

Autogenerating the code means that we actually are able to expose things fairly easily to both the web and to our webmodules directory. It also makes it impossible to accidentally start relying on a backdoor. You'd need to actually modify the web-exposed IDL to add a backdoor.

The autogenerated code will most likely use the same wrapper pattern as the web/ directory in Blink, e.g. WebNode. We could even expose this API to content directly instead of the web layer we expose now, but we probably don't want to give all that power to embedders. They'll abuse it and we'll be sad.

Patch that adds the initial plumbing: https://codereview.chromium.org/1248043003.

This section needs much more fleshing out, but here are a few thoughts:

- Some APIs may actually expose a way to run JS in a context (e.g. CompositorWorker). In that case, part of the feature could actually be implemented in JS.
- Can mimic JS callbacks with std::function or something like it.
- Can mostly just generate C++ that you'd use instead of JS. Might need wrapper objects in some cases, but they won't participate in GC or anything, so there won't be any complicated interactions with V8.
- Exceptions will be a bit of a pain since we won't want to use C++ exceptions. We'll need to handle them manually maybe (e.g. appendChild won't throw, you'll need to check that it succeeded). Should be able to create helpers to make this not too painful.
- V1 does not need to support 100% of the IDL we expose to the web. We can expose just the bits we need as each new API needs it (e.g. if we don't need to use any APIs from webmodules that use Dictionaries, we don't need to implement support for that quite yet).

- Need to provide a way for webmodules to associate state with Nodes and ExecutionContexts. On the web authors just assign variables inside a closure and use them in the closures for event handlers. C++ means more complicated ownership is going to be required. Can we use WeakMap like things and Oilpan? Could we code generate "global state" object that has typed properties for each webmodule?
- Might need some way for modules to register which APIs they want access to so that we don't have to create C++ bindings for every API exposed to the web for the sake of compile times and binary size.

Web platform...the good parts

The web platform is littered with APIs that we wouldn't actually want to build on. As such, we want to be careful about what parts of the API we actually want to expose via the C++ bindings. Instead of just exposing all the APIs, we'll whitelist the APIs we want to expose via an IDL annotation. In this way, we can expose just the subset of the platform that we think is the good set of primitives.

On a case-by-case basis, we can decide what features to expose. In cases where an API exists that meets a given need, but does so in a less than perfect way, we'll need to consider the engineering tradeoffs of coming up with a replacement that we could plausibly ship to web authors. We're still **only** exposing APIs to webmodules that we either expose to web authors today or intend to do so in the future.

The goal here is to stop the bleeding of adding new, non-layered features to the platform. There are some existing features of the platform that we will never be able to reimplement on top of primitives because they require primitives that we wouldn't expose (e.g. due to privacy concerns). A good example is visited links. In order to build those on primitives, we'd need to expose your navigation history to web authors. Hopefully, we'll never need to add private backdoors to the C++ bindings APIs because we'll only add new features to the platform the are explainable in terms of primitives.

Making practical compromises

In an ideal world, we'd be 100% purist about this approach and only build high-level APIs on top of low-level ones, but some of these efforts will conflict with product timeline requirements. For a high-level feature we should:
For each primitive that we're lacking, we should do the following:

- Create a proposal of how to implement the high-level API on top of (existing and/or new) primitives.
- For each missing primitive, create a 1-2 page design of how it would work.
- Rendering-leads, along with the engineers involved, will then need to evaluate the engineering versus product tradeoffs. If there are concerns that the new primitives would take too long to build, then we create a proposal of how to implement the API without primitives to see if we are comfortable with the less layered implementation. **In some**

cases, we may decide to implement on as many primitives as we can, but to make an exception for an exceptionally difficult to implement one.

The primary case where a sense of urgency to ship may outweigh the purist application of only implementing a layered platform is when other browsers have all (or mostly all) shipped an API already, e.g. snap-points. We don't want to be responsible for holding the web back, even at the benefit of building a better, layered platform.

Why not double down on Blink-in-JS

Blink-in-JS is the purest form of this effort. It **really** is the thing we expose to authors. However, Blink-in-JS has considerable risk factors that would then be transferred to any features that depend on it:

- To not expose implementation details to main world script, Blink-in-JS needs to fall back to C++, switch to the isolated world, and re-enter JS. This is slow. It's not clear how much work will be needed to make the performance acceptable for critical code.
- V8/JS doesn't give enough control over things like memory usage, GC behavior, object packing, doesn't understand how to devirtualize calls, etc. Maybe strong mode will fix this, but that's not something features we need to ship ASAP can wait on.
- V8/JS can't share code pages between processes like C++ can. Hopefully we can fix this, but that's not even on the radar of the V8 team and the JS standards teams.

As per the above, Blink-in-JS is good for replacing large things like XSLT. It works less well for very performance sensitive things like mouse/touch events.

Future plans

This effort is a practical, incremental step towards a larger goal. Long-term, what we'd really like to see is that we move all of this C++ code over to JavaScript and **actually** host this code on top of the platform. At that point, it's not hard to imagine that we could share this code with other browser vendors in a standard library. Once we have a standard library, we can make developers import new features that we add instead of having everything available to you by default. That way, you only pay the cost for the features you use.

Next steps

1. Agree that this is the approach we want to take towards implementing a layered platform.
2. Make an engineering plan to implement this infrastructure.
3. Once the above are done, start evaluating the engineering/product tradeoffs of implementing new features on this infrastructure.