Platform Frameworks

esprehn@, Jan. 2017

Introduction

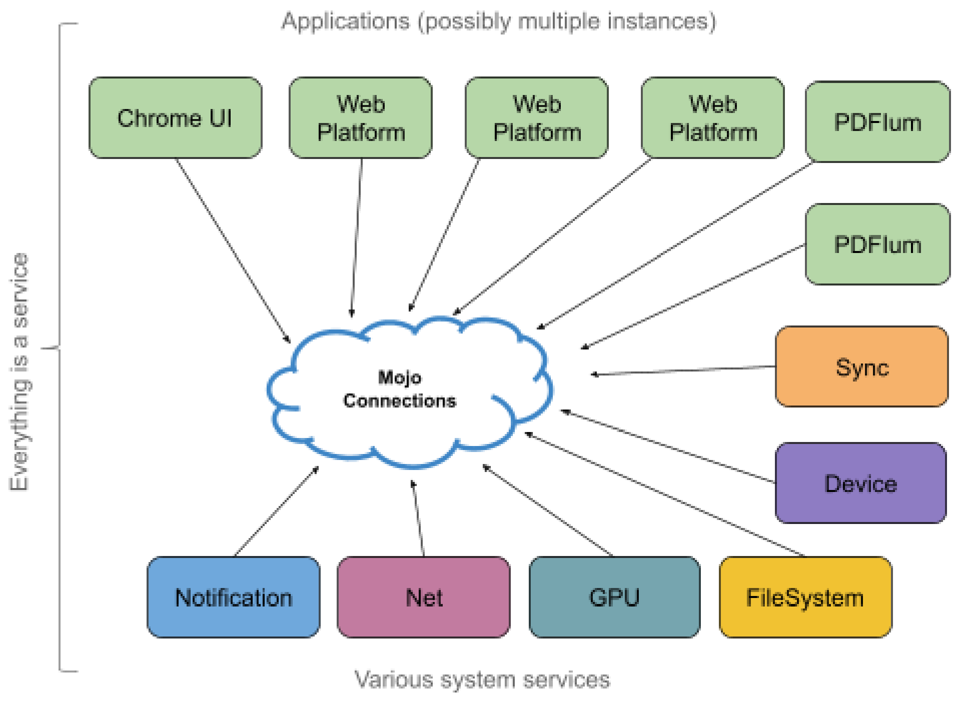
As the web has evolved from a simple system to view documents into a full application platform so too have web browsers evolved from mere document viewers into something quite similar to an entire operating system.

Recently we've begun the process of converting the monolithic Chromium codebase into a set of services connected through Mojo IPC, a process called *servicification*. In this way Chromium is further becoming like popular operating systems: Android is a set of services connected through Binder, macOS a set of services connected through Mach. Both these platforms also have an opinionated set of libraries that sit on top the of IPC layer and embody a set of best practices for services within the operating system. Both of these platforms call them Frameworks!

This document outlines the principles and strategy by which we take the work done during the Onion Soup project and the technology that has been developed within the blink codebase and create opinionated frameworks for writing services in the new mojo connected system. These frameworks can then be used to write any service within the Chromium system, not just the web platform itself. We call this process *frameworkification*.

Mission

At the end result of *servicification* Chromium will be transformed into a collection of services that communicate through mojo. Some services will be singletons mapping to discrete local resources like the GPU, while others will have multiple instances like web platform with an instance for each tab.



To be a service in this new system there's only two requirements:

1. Each service must be capable of communicating by mojo so it can connect to other services and be started by the service manager.

2. Each service must be a good citizen of the overall system by sharing the limited local resources of CPU, Disk, Network, GPU and Power.

This brings a great deal of flexibility to the system. For example, we could write new services in any programming language that supports mojo bindings. This flexibility also comes with great responsibility in requirement #2. We could task each service owner with figuring out how to meet this goal independently, but this is hard to scale without a brightly lit path to success. The platform frameworks will provide this path.

For example, a new service might be written in JS. The *JavaScript framework* should provide the necessary tools to build this service while avoiding memory leaks, being sensitive to memory pressure, respecting the task scheduler when the user is touching the screen, and more.

The mission of the platform frameworks is to ensure that all services written with them are the fastest, most secure, and respectful on the system. *Frameworkification* is successful when service authors reach to them first because they trust the frameworks to make their service awesome.

Principles

A key feature of frameworks over typical libraries is having strong opinions. A framework embodies a set of best practices so consumers of it are implicitly making good choices both for themselves and for other services running on the system.

The guiding principles are *Friendly*, *Fun* and *Fast*:

Be a good citizen

Raw performance is not enough, a service must also not monopolize system resources starving other services. It's often possible to get better raw performance in a micro benchmark by dedicating the entire system to a single service, but this works out poorly once there's more than one service in use (ex. Audio and IndexedDB at the same time).

The platform frameworks should assume there's competing services and be respectful in their in resource consumption. For example they should never allocate caches without connecting them to a memory coordination system that can balance their cache sizes against that of other systems.

We call this being **Friendly**.

Be joyful to use

Complex and subtle code is painful to maintain and reduces productivity of the entire organization. A powerful tool of frameworks is making writing code fun by combining low level concepts into high level abstractions that are fun to use.

The platform frameworks should attempt to create joyful and fun abstractions with ample documentation that make code beautiful and avoid leaking low level details about how the underlying service architecture works.

We call this being **Fun**.

Be fast by default

Low level systems generally provide flexible and powerful APIs that allow trading convenience against speed or resource consumption. A good framework makes these trade offs on behalf of the user so they don't need to remember which options to pass into the low level system to get the best behavior.

The platform frameworks choose the fast by default behavior so services written with them are great right from the start. Being fast by default also means being mindful of power consumption since the system will become slow when heat is high or battery is low. This might mean a slight learning curve by having an opinionated API surface, but the frameworks mitigate this by having robust DCHECK coverage and ample documentation.

We call this being **Fast**.

Dependencies

Another key feature of frameworks is having a set of opinionated dependencies. The underlying system often provides multiple choices for how to solve a problem, it's the job of a framework to take an opinion on a subset of them. This makes meeting the guiding principles easy and empowers collaboration between the framework's various components.

The frameworks have the following dependencies:

Mojo

Mojo is an IPC system with a declarative language for describing messages and a code generator to allow creating bindings against any target language. This abstraction allows services within the Chromium system to be written against any language that is capable of handling mojo messages.

All platform frameworks assume you’re using Mojo as the IPC system to communicate with external services, and themselves use it to communicate with the underlying coordination systems.

PartitionAlloc

PartitionAlloc (which is in base) is a security and performance minded memory allocator. Using it removes uncertainty in malloc performance that comes from using the system allocator which is provided by a different vendor on every platform. This gives predictable memory usage and performance, and improves security by using ASLR techniques within the allocator.

All platform frameworks use PartitionAlloc as much as possible.

Oilpan

The Oilpan garbage collector allows vastly simplified lifetime management throughout the codebase improving performance and security. Further it lets the system perform heap compaction which can result in substantial reductions in memory consumption.

All platform frameworks assume that the Oilpan garbage collector exists and leverage garbage collected data types.

WTF + base

WTF and base are utility libraries that provide abstractions over the standard library and operating system specific functionality. By using them developers avoid needing to worry about platform and deployment target differences. WTF also provides deep integration with the Oilpan garbage collector, and a set of memory and performance optimized data structures for use with it.

All platform frameworks use WTF whenever possible for the best integration with Oilpan. This also enables frameworks to give performance and memory guarantees across all of the deployment and compiler targets.

Task Scheduler

The Lucky Luke task scheduler is a low level library provided by base/ which can manage thread pools, task posting and priority scheduling. Using it allows the effective use of CPU resources and threads.

All of the platform frameworks assume that Lucky Luke is available so they don't need to manually manage low level task scheduling and thread pooling.

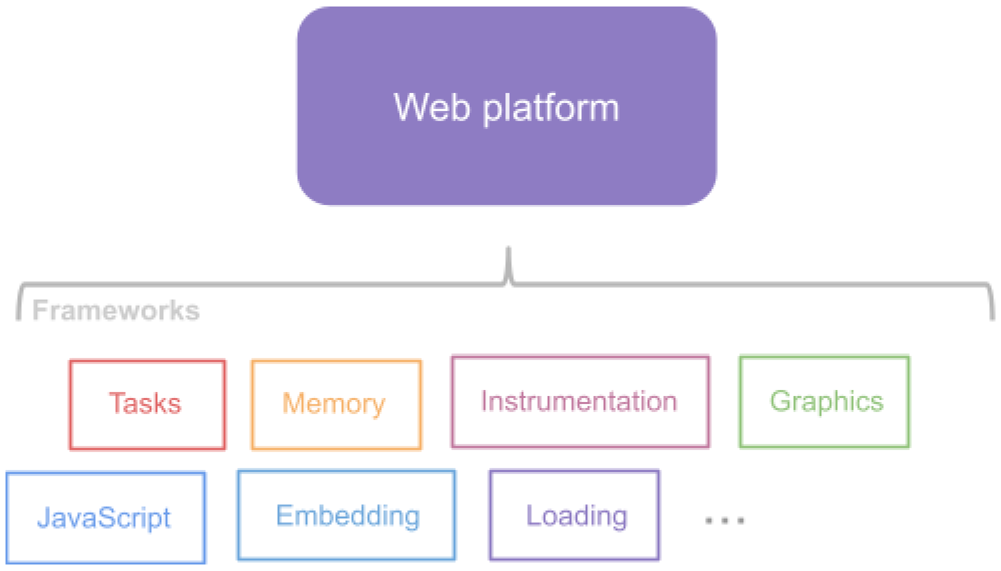
<Any other hard dependencies we should assume?>

Size and ownership

To achieve the principles and mission each framework must be large and self contained enough to provide a full stack experience when writing a service. Each one should also have a clear TL, bug component, and set of documentation. This limits the number of possible frameworks that should exist to a handful that represent high level application building concepts.

Getting started

The blink codebase already has the beginnings of each one of the frameworks depicted below, the task of frameworkification is to incubate each framework inside blink and then refactor it into a self contained library. In the process the code gains an owner, a well defined interface, and documentation.



Tasks framework

This framework provides an abstraction over the Task Scheduler. It's opinionated in how it posts tasks and schedules them, often making choices that favor responsiveness over raw throughput. It uses Oilpan to simplify the lifetime management of tasks and their associated resources. It also requires all posted tasks to contain information about the task source (ex. Frame url) to allow the framework to integrate with the global scheduling system that coordinates all services.

All other frameworks depend on the *Tasks framework* to post tasks and handle asynchronous work.

This framework emerges from the blink scheduler.

Memory framework

This framework provides tools for allocating memory, creating caches, and dealing with various memory situations like low memory or background tabs. The framework provides high level concepts related to caching so service authors don’t waste memory and respect system memory constraints.

For example instead of creating a cache as a HashMap, a service author may use a CacheMap which is setup with policy information and a connection to the memory manager mojo service so it gets purged appropriately. In a low memory situation the cache can be changed to always “miss” so it stops growing and can then be reenabled when the memory manager decices there’s spare resources.

All other frameworks depend on the *Memory framework* to handle caching and memory resource management.

This framework emerges from WTF, Oilpan and the Memory Coordinator.

Instrumentation framework

This framework provides an opinionated interface over the PCU system coordinator, tracing, UMA, DevTools, and various other metrics and instrumentation systems in the codebase. It provides Oilpan integration for these systems, and hides the details of the various metrics systems behind a single unified API surface.

All other frameworks depend on the instrumentation framework and leverage its API to provide information to global coordinator (ex. PCU).

This framework emerges from platform/instrumentation, tracing, PerformanceMonitor, the DevTools code, the UMA code, etc.

Graphics framework

This framework provides high level abstractions over the low level Skia and cc graphics libraries. It uses Oilpan to simplify the lifetime management of resources and leverages the memory framework for caches. It contains opinionated high level display list and drawing APIs and hides all details of GPU vs Software raster from the consumers.

This framework emerges from platform/graphics and the paint code in core.

JavaScript framework

This framework provides the tools to build a service written in JavaScript. It contains a high performance bindings system, wrapper tracing and management for C++ bindings, code generators, mojo bindings for JS, and all the various tools required to use the lower level v8 API correctly.

This framework can be leveraged by PDFium, the web platform, and any other service that requires running JS.

This framework emerges from the blink bindings system, code generators, traceWrapper, upcoming mojo bindings in blink, etc.

Embedding framework

This framework contains abstractions over the concepts of navigation and the frame tree. It communicates with a remote service that contains the canonical frame tree and abstracts over the Remote vs Local frame concepts so service authors can generally ignore the difference.

The service provides tools for embedding new applications inside the view of the current service. This can include PDFium, web content, NaCl, or any other service capable of vending a frame instance. The framework itself contains a high level API that the web platform uses to register content type handles (ex. text/html -> a web frame).

This framework emerges from the core/frames, web/ layer, content/ RenderFrameImpl and friends, and all other code related to the frame tree and navigation.

Loading framework

This framework contains abstractions over the network and loading concepts. Similar to how JS developers use fetch() to communicate with the network, services will use the *Loading framework*. It provides both a friendly interface for loading resources and a powerful caching system that's integrated with the *Memory framework*. It uses Oilpan types to simplify lifetime management and uses the *Tasks framework* to ensure that all network related work is handled on the right task queues.

This framework emerges from the platform/network, platform/weborigin, upcoming platform/loader, core/fetch and the related loading code spread across the codebase.

FAQ

Is this list of frameworks exhaustive?
Nope! It's entirely possible that while we're developing these frameworks we realize some other ones need to exist. That said, there should be a pretty small number of frameworks in general. Each needs to be large enough to support a defined API, documentation and a TL. Each should represent a fairly sizable concept needed when developing a service.