# Mojo

## Contents



## Getting Started With Mojo

To get started using Mojo in applications which already support it (such as Chrome), the fastest path forward will be to look at the bindings documentation for your language of choice (**C++**, **JavaScript**, or **Java**) as well as the documentation for the **Mojom IDL and bindings generator**.

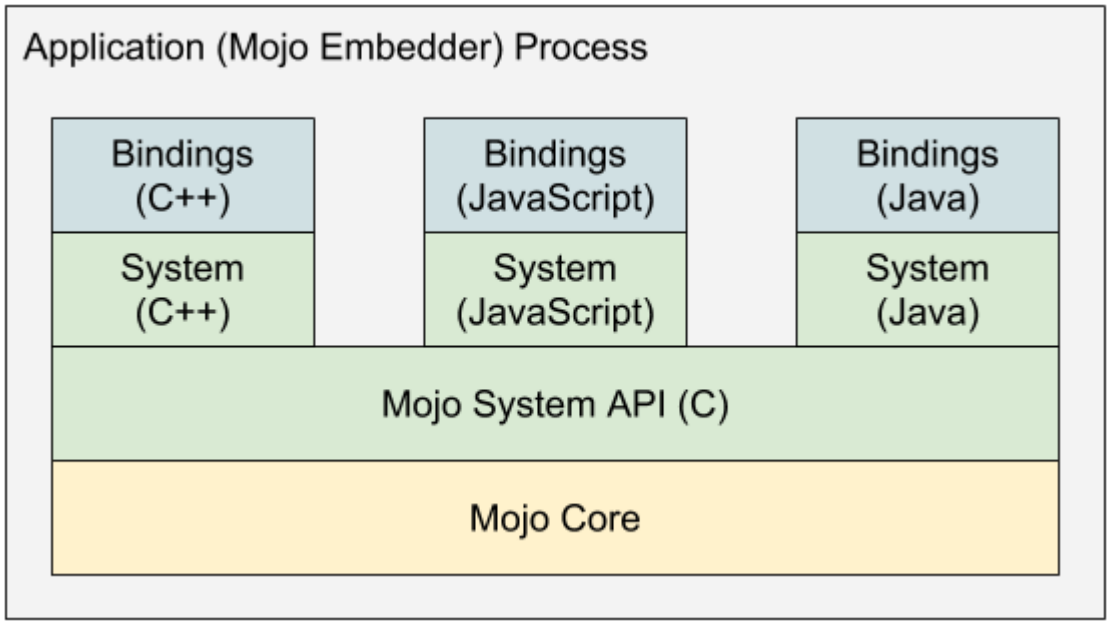
If you're looking for information on creating and/or connecting to services, see the top-level Services documentation.

For specific details regarding the conversion of old things to new things, check out Converting Legacy Chrome IPC To Mojo.

# System Overview

Mojo is a collection of runtime libraries providing a platform-agnostic abstraction of common IPC primitives, a message IDL format, and a bindings library with code generation for multiple target languages to facilitate convenient message passing across arbitrary inter- and intra-process boundaries.

The documentation here is segmented according to the different libraries comprising Mojo. The basic hierarchy of features is as follows:




# Mojo Core

In order to use any of the more interesting high-level support libraries like the System APIs or Bindings APIs, a process must first initialize Mojo Core. This is a one-time initialization which remains active for the remainder of the process's lifetime. There are two ways to initialize Mojo Core: via the Embedder API, or through a dynamically linked library.

## Embedding

Many processes to be interconnected via Mojo are **embedders**, meaning that they statically link against the `//mojo/core/embedder` target and initialize Mojo support within each process by calling `mojo::core::Init()`. See **Mojo Core Embedder API** for more details.

This is a reasonable option when you can guarantee that all interconnected process binaries are linking against precisely the same revision of Mojo Core. To support other scenarios, use dynamic linking.

## Dynamic Linking

On some platforms, it's also possible for applications to rely on a dynamically-linked Mojo Core library (`libmojo_core.so` or `mojo_core.dll`) instead of statically linking against Mojo Core.

In order to take advantage of this mechanism, the corresponding library must be present in either:

- The working directory of the application
- A directory named by the `MOJO_CORE_LIBRARY_PATH` environment variable
- A directory named explicitly by the application at runtime

Instead of calling `mojo::core::Init()` as embedders do, an application using dynamic Mojo Core instead calls `MojoInitialize()` from the C System API. This call will attempt to locate (see above) and load a Mojo Core library to support subsequent Mojo API usage within the process.

Note that the Mojo Core shared library presents a stable, forward-compatible C ABI which can support all current and future versions of the higher-level, public (and not binary-stable) System and Bindings APIs.

## C System API

Once Mojo is initialized within a process, the public **C System API** is usable on any thread for the remainder of the process's lifetime. This is a lightweight API with a relatively small, stable, forward-compatible ABI, comprising the total public API surface of the Mojo Core library.

This API is rarely used directly, but it is the foundation upon which all higher-level Mojo APIs are built. It exposes the fundamental capabilities to create and interact Mojo primitives like **message pipes**, **data pipes**, and **shared buffers**, as well as APIs to help bootstrap connections among processes.

## Platform Support API

Mojo provides a small collection of abstractions around platform-specific IPC primitives to facilitate bootstrapping Mojo IPC between two processes. See the Platform API documentation for details.

## High-Level System APIs

There is a relatively small, higher-level system API for each supported language, built upon the low-level C API. Like the C API, direct usage of these system APIs is rare compared to the bindings APIs, but it is sometimes desirable or necessary.

### C++

The **C++ System API** provides a layer of C++ helper classes and functions to make safe System API usage easier: strongly-typed handle scopers, synchronous waiting operations, system handle wrapping and unwrapping helpers, common handle operations, and utilities for more easily watching handle state changes.

### JavaScript

The **JavaScript System API** exposes the Mojo primitives to JavaScript, covering all basic functionality of the low-level C API.

### Java

The **Java System API** provides helper classes for working with Mojo primitives, covering all basic functionality of the low-level C API.

## High-Level Bindings APIs

Typically developers do not use raw message pipe I/O directly, but instead define some set of interfaces which are used to generate code that resembles an idiomatic method-calling interface in the target language of choice. This is the bindings layer.

### Mojom IDL and Bindings Generator

Interfaces are defined using the **Mojom IDL**, which can be fed to the **bindings generator** to generate code in various supported languages. Generated code manages serialization and deserialization of messages between interface clients and implementations, simplifying the code -- and ultimately hiding the message pipe -- on either side of an interface connection.

### C++ Bindings

By far the most commonly used API defined by Mojo, the **C++ Bindings API** exposes a robust set of features for interacting with message pipes via generated C++ bindings code, including support for sets of related bindings endpoints, associated interfaces, nested sync IPC, versioning, bad-message reporting, arbitrary message filter injection, and convenient test facilities.

### JavaScript Bindings

The **JavaScript Bindings API** provides helper classes for working with JavaScript code emitted by the bindings generator.

### Java Bindings

The **Java Bindings API** provides helper classes for working with Java code emitted by the bindings generator.

## FAQ

### Why not protobuf? Why a new thing?

There are number of potentially decent answers to this question, but the deal-breaker is that a useful IPC mechanism must support transfer of native object handles (*e.g.* file descriptors) across process boundaries. Other non-new IPC things that do support this capability (*e.g.* D-Bus) have their own substantial deficiencies.

### Are message pipes expensive?

No. As an implementation detail, creating a message pipe is essentially generating two random numbers and stuffing them into a hash table, along with a few tiny heap allocations.

## So really, can I create like, thousands of them?

Yes! Nobody will mind. Create millions if you like. (OK but maybe don't.)

## What are the performance characteristics of Mojo?

Compared to the old IPC in Chrome, making a Mojo call is about 1/3 faster and uses 1/3 fewer context switches. The full data is available here.

## Can I use in-process message pipes?

Yes, and message pipe usage is identical regardless of whether the pipe actually crosses a process boundary -- in fact this detail is intentionally obscured.

Message pipes which don't cross a process boundary are efficient: sent messages are never copied, and a write on one end will synchronously modify the message queue on the other end. When working with generated C++ bindings, for example, the net result is that an `InterfacePtr` on one thread sending a message to a `Binding` on another thread (or even the same thread) is effectively a `PostTask` to the `Binding`'s `TaskRunner` with the added -- but often small -- costs of serialization, deserialization, validation, and some internal routing logic.

## What about ___?

Please post questions to `chromium-mojo@chromium.org`! The list is quite responsive.