













[cobalt](#) / [cobalt](#) / [release\\_11](#) / [.](#) / **src**

tree: d55d717320aecd5b1c04fba1c51dc7f4bd58806e [[path history](#)] [[tgz](#)]

-  [.clang-format](#)
-  [AUTHORS](#)
-  [CONTRIBUTING.md](#)
-  [LICENSE](#)
-  [README.md](#)
-  [base/](#)
-  [build/](#)
-  [cobalt/](#)
-  [content/](#)
-  [crypto/](#)
-  [glimp/](#)
-  [googleurl/](#)
-  [nb/](#)
-  [net/](#)
-  [sql/](#)
-  [starboard/](#)
-  [testing/](#)
-  [third\\_party/](#)
-  [tools/](#)
-  [ui/](#)

---

src/README.md

# Cobalt

## Overview

Cobalt is a lightweight application container (i.e. an application runtime, like a JVM or the Flash Player) that is compatible with a subset of the W3C HTML5 specifications. If you author a single-page web application (SPA) that complies with the Cobalt Subset of W3C standards, it will run as well as possible on all the devices that Cobalt supports.

## Motivation

The Cobalt Authors originally maintained a port of Chromium called H5VCC, the HTML5 Video Container for Consoles, ported to each of the major game consoles, designed to run our HTML5-based video browse and play application. This took a long time to port to each platform, consisted of 9 million lines

of C++ code (before we touched it), was dangerous to modify without unintended consequences, and was thoroughly designed for a resource-rich, multi-process environment (e.g. a desktop, laptop, or modern smartphone).

After wrestling with this for several years, we imagined an environment that was not designed for traditional scrolling web content, but was intended to be a runtime environment for rich client applications built with the same technologies -- HTML, CSS, JavaScript -- and designed from the ground-up to run on constrained, embedded, Living Room Consumer Electronics (CE) devices, such as Game Consoles, Set-Top Boxes (e.g. Cable, Satellite), OTT devices (e.g. Roku, Apple TV, Chromecast, Fire TV), Blu-ray Disc Players, and Smart TVs.

These constraints (not intended to be a canonical list) make this device spectrum vastly different from the desktop computer environment targeted by Chromium, FireFox, and IE:

- **Limited Memory.** All except the very latest, expensive CE devices have a very small amount of memory available for applications. This usually is somewhere in the ballpark of 200MB-500MB, including graphics and media memory, as opposed to multiple gigabytes of CPU memory (and more gigabytes of GPU memory) in modern desktop and laptop computers, and mobile devices.
- **Slow CPUs.** Most CE devices have much slower CPUs than what is available on even a budget desktop computer. Minor performance concerns can be greatly exaggerated, which seriously affects priorities.
- **Fewer cores.** CE System-on-a-Chip (SoC) processors often do not have as many processor cores as we are used to in modern computers. Many deployed devices still only have a single core.
- **Sometimes No GPU.** Not all CE devices have a monster GPU to throw shaders at to offload CPU work. A different strategy is required to maximize leverage of an accelerated blitter, which is all some older devices have. Some newer CE devices have a GPU, but it's not nearly as powerful as what one would even see on a laptop.
- **Sometimes No JIT.** Many CE devices are dealing with "High-Value Content," and, as such, are very sensitive to security concerns. Ensuring that writable pages are not executable is a strong security protocol that can prevent a wide spectrum of attacks. But, as a side effect, this also means no ability to JIT.
- **Heterogenous Development Environments.** This is slowly evening out, but all CE devices run on custom hardware, often with proprietary methods of building, packaging, deploying, and running programs. Almost all CE devices have ARM or MIPS processors instead of the more familiar x86. Sometimes the toolchain doesn't support contemporary C++11 features. Sometimes the OS isn't POSIX, or it tries to be, but it is only partially implemented. Sometimes the program entry point is in another language or architecture that requires a "trampoline" over to native binary code.
- **No navigation.** The point of a Single-Page Application is that you don't go through the HTTP page dance every time you switch screens. It's slow, and provides poor user feedback, not to mention a jarring transition. Instead, one loads data from an XMLHttpRequest (XHR), and then updates one's DOM to reflect the new data. AJAX! Web 2.0!!
- **No scrolling.** Well, full-screen, 10-foot UI SPA apps might scroll, but not like traditional web pages, with scroll bars and a mouse wheel. Scrolling is generally built into the app very carefully, with support for a Directional Pad and a focus cursor.

## Architecture

The Cobalt Authors forked H5VCC, removed most of the Chromium code -- in particular WebCore and the Chrome Renderer and Compositor -- and built up from scratch an implementation of a simplified subset of HTML, the CSS Box Model for layout, and the Web APIs that were really needed to build a full-screen SPA browse and play application.

The Cobalt technology stack has these major components, roughly in a high-level application to a low-level platform order:

- **Web Implementation** - This is where the W3C standards are implemented, ultimately producing an annotated DOM tree that can be passed into the Layout Engine to produce a Render Tree.
- **JavaScript Engine** - We have, perhaps surprisingly, *not* written our own JavaScript Engine from scratch. Because of the JITing constraint, we have to be flexible with which engine(s) we work with. We have a bindings layer that interfaces with the JS Engine so that application script can interface with natively-backed objects (like DOM elements).
- **Layout Engine** - The Layout Engine takes an annotated DOM Document produced by the Web Implementation and JavaScript Engine working together, and calculates a tree of rendering commands to send to the renderer (i.e. a Render Tree). It caches intermediate layout artifacts so that subsequent incremental layouts can be sped up.
- **Renderer/Skia** - The Renderer walks a Render Tree produced by the Layout Engine, rasterizes it using the third-party graphics library Skia, and swaps it to the front buffer. There are two major paths here, one using Hardware Skia on OpenGL ES 2.0, and one using Software Skia combined with the hardware-accelerated Starboard Blitter. Note that the renderer runs in a different thread from the Layout Engine, and can interpolate animations that do not require re-layout. This decouples rendering from Layout and JavaScript, allowing for smooth, consistent animations on platforms with a variety of capabilities.
- **Net / Media** - These are Chromium's Network and Media engines. We are using them directly, as they don't cause any particular problems with the extra constraints listed above.
- **Base** - This is Chromium's "Base" library, which contains a wide variety of useful things used throughout Cobalt, Net, and Media. Cobalt uses a combination of standard C++ containers (e.g. vector, string) and Base as the foundation library for all of its code.
- **Other Third-party Libraries** - Most of these are venerable, straight-C, open-source libraries that are commonly included in other open-source software. Mostly format decoders and parsers (e.g. libpng, libxml2, zlib). We fork these from Chromium, as we want them to be the most battle-tested versions of these libraries.
- **Starboard/Glimp/ANGLE - Starboard** is the Cobalt porting interface. One major difference between Cobalt and Chromium is that we have created a hard straight-C porting layer, and ported ALL of the compiled code, including Base and all third-party libraries, to use it instead of directly using POSIX standard libraries, which are not consistent, even on modern systems (see Android, Windows, MacOS X, and iOS). Additionally, Starboard includes APIs that haven't been effectively standardized across platforms, such as display Window creation, Input events, and Media playback. **Glimp** is an OpenGL ES 2.0 implementation framework, built by the Cobalt team directly on Starboard, designed to adapt proprietary 3D APIs to GLES2. **ANGLE** is a third-party library that adapts DirectX to GLES2, similar to Glimp, but only for DirectX.

Cobalt is like a flaky layered pastry - perhaps Baklava. It shouldn't be too difficult to rip the Web Implementation and Layout off the top, and just use the Renderer, or even to just use Base + Starboard + GLES2 as the basis of a new project.

## The Cobalt Subset

Oh, we got both kinds of HTML tags,  
we got `<span>` and `<div>` !

See the [Cobalt Subset specification](#) for more details on which tags, properties, and Web APIs are supported in Cobalt.

*More to come.*

## Interesting Source Locations

All source locations are specified relative to `src/` (this directory).

- `base/` - Chromium's Base library. Contains common utilities, and a light platform abstraction, which has been superseded in Cobalt by Starboard.
- `net/` - Chromium's Network library. Contains enough infrastructure to support the network needs of an HTTP User-Agent (like Chromium or Cobalt), an HTTP server, a DIAL server, and several abstractions for networking primitives. Also contains SPDY and QUIC implementations.
- `media/` - Chromium's Media library. Contains all the code that parses, processes, and manages buffers of video and audio data. Media decoding is passed off to decoding hardware, wherever possible.
- `cobalt/` - The home of all Cobalt application code. This includes the Web Implementation, Layout Engine, Renderer, and some other Cobalt-specific features.
  - `cobalt/build/` - The core build generation system, `gyp_cobalt`, and configurations for supported platforms. (NOTE: This should eventually be mostly moved into `starboard/`.)
- `starboard/` - Cobalt's porting layer. Please see Starboard's [README.md](#) for more detailed information about porting Starboard (and Cobalt) to a new platform.
- `third_party/` - Where all of Cobalt's third-party dependencies live. We don't mean to be perjorative, we love our third-party libraries! This location is dictated by Google OSS release management rules...
  - `third_party/starboard/` - The location for third-party ports. This directory will be scanned automatically by `gyp_cobalt` for available Starboard ports.

## Building and Running the Code

Here's a quick and dirty guide to get to build the code on Linux.

1. Pull `depot_tools` into your favorite directory. It has been slightly modified from Chromium's `depot_tools`.

```
git clone https://cobalt.googlesource.com/depot_tools
```

2. Add that directory to the end of your `$PATH`.
3. Ensure you have these packages installed:

```
sudo apt-get install libgles2-mesa-dev libpulse-dev libavformat-dev \
libavresample-dev libasound2-dev libxrender-dev libxcomposite-dev
```

4. Ensure you have the standard C++ header files installed (e.g. `libstdc++-4.8-dev`).
5. For now, we also require ruby:

```
sudo apt-get install ruby
```

6. Remove bison-3 and install bison-2.7. (NOTE: We plan on moving to bison-3 in the future.)

```
$ sudo apt-get remove bison
$ sudo apt-get install m4
$ wget http://ftp.gnu.org/gnu/bison/bison-2.7.1.tar.gz
$ tar xzf bison-2.7.1.tar.gz
$ cd bison-2.7.1
$ sh configure && make && sudo make install
$ which bison
/usr/local/bin/bison
$ bison --version
bison (GNU Bison) 2.7.12-4996
```

7. (From this directory) run GYP:

```
cobalt/build/gyp_cobalt -C debug linux-x64x11
```

8. Run Ninja:

```
ninja -C out/linux-x64x11_debug cobalt
```

9. Run Cobalt:

```
out/linux-x64x11_debug/cobalt [--url=<url>]
```

- If you want to use `http` instead of `https`, you must pass the `--allow_http` flag to the Cobalt command-line.
- If you want to connect to an `https` host that doesn't have a certificate validatable by our set of root CAs, you must pass the `--ignore_certificate_errors` flag to the Cobalt command-line.
- See [cobalt/browser/switches.cc](http://cobalt/browser/switches.cc) for more command-line options.

## Build Types

Cobalt has four build optimization levels, going from the slowest, least optimized, with the most debug information at the top (debug) to the fastest, most optimized, and with the least debug information at the bottom (gold):

Type	Optimizations	Logging	Asserts	Debug Info	Console
debug	None	Full	Full	Full	Enabled
devel	Full	Full	Full	Full	Enabled
qa	Full	Limited	None	None	Enabled
gold	Full	None	None	None	Disabled

When building for release, you should always use a gold build for the final product.

```
$ cobalt/build/gyp_cobalt -C gold linux-x64x11  
$ ninja -C out/linux-x64x11_gold cobalt  
$ out/linux-x64x11_gold/cobalt
```

## Origin of this Repository

This is a fork of the chromium repository at <http://git.chromium.org/git/chromium.git>

Powered by [Gitiles](#) | [Privacy](#)

[txt](#) [json](#)