Contributing to Chrome DevTools Protocol

This guide assumes the reader already knows how to contribute to Chromium.

DevTools protocol is a debugging protocol supported by Chromium. This article explains how the DevTools protocol fits into the Chromium architecture and aims to help contributors to find their way in the codebase.

> **NOTE:** Interactive protocol viewers are available at: https://chromedevtools.github.io/devtools-protocol/ (official) and https://vanilla.aslushnikov.com (unofficial).

Overview

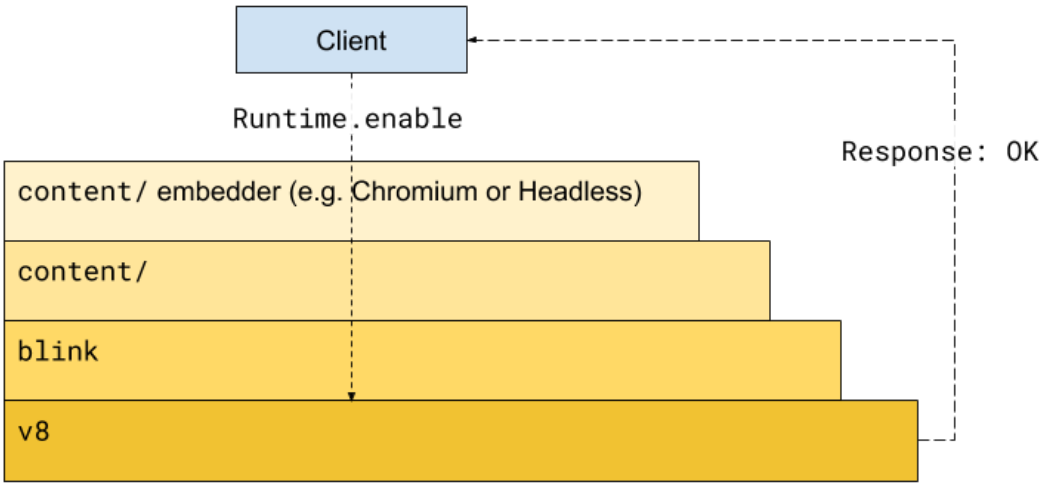
First of all, let's agree on the terminology we use when we talk about the Chrome DevTools protocol:

- `target`: a debuggable entity, typically a page or a service worker. Targets could be top-level (like pages) or nested (like web workers).
- `client`: anyone who wants to communicate with the browser over the DevTools protocol
- `session`: a debugging session representing a connection between a `client` and a `target`. Multiple sessions could be attached to the same target.
- `handler` or `agent`: a DevTools Protocol handler. Historically, blink-level protocol handlers are called “agents,” and everywhere else we use “handlers.”

The first thing every client does is connect to a target. Once connected, a `session` object is created and gets configured with protocol handlers relevant to the target. Once the session is closed, it gets destroyed and removes all its protocol handlers.

> **NOTE:** Different targets define different sets of protocol handlers. (For example, compare session setups for browser target and frame target.)

Once connected, the client starts sending protocol commands. Depending on the command, it can be handled on multiple levels:



It's up to the protocol implementor to decide where they want to handle protocol messages.
The rule of thumb is "debugging support for feature *FOO* should be right next to its implementation."

A few examples:

- All debugger methods, such as `Runtime.evaluate`, should be implemented in `v8` so that they're available in node.js. This way we can debug node.js applications with the DevTools front-end.
- Web Platform inspections are encapsulated in `blink`.
- Browser-related stuff, such as targets and tracing, is implemented in the `content/` layer.
- Rarely, functionality depends heavily on the `content/` embedders. In this case, at a minimum it must be implemented in both the Chromium and Headless embedders.

Protocol Anatomy

The protocol consists of three parts: Definition, Configuration and Implementation.

Definition describes a set of methods and events supported by the debugging protocol. The protocol is defined in two files:

- `browser_protocol.pdl` – describes browser-related operations and lives in blink
- `js_protocol.pdl` – describes javascript-related operations and lives in v8

> **NOTE** “PDL” (pronounced as "ˈpo͞odl") is a home-grown format to describe the DevTools protocol. PDL support, such as Sublime syntax highlighting and the json converter, is available at https://github.com/pavelfeldman/pdl.

Configuration files exist in every Chromium layer and they whitelist protocol methods that should be handled in the layer it belongs to.

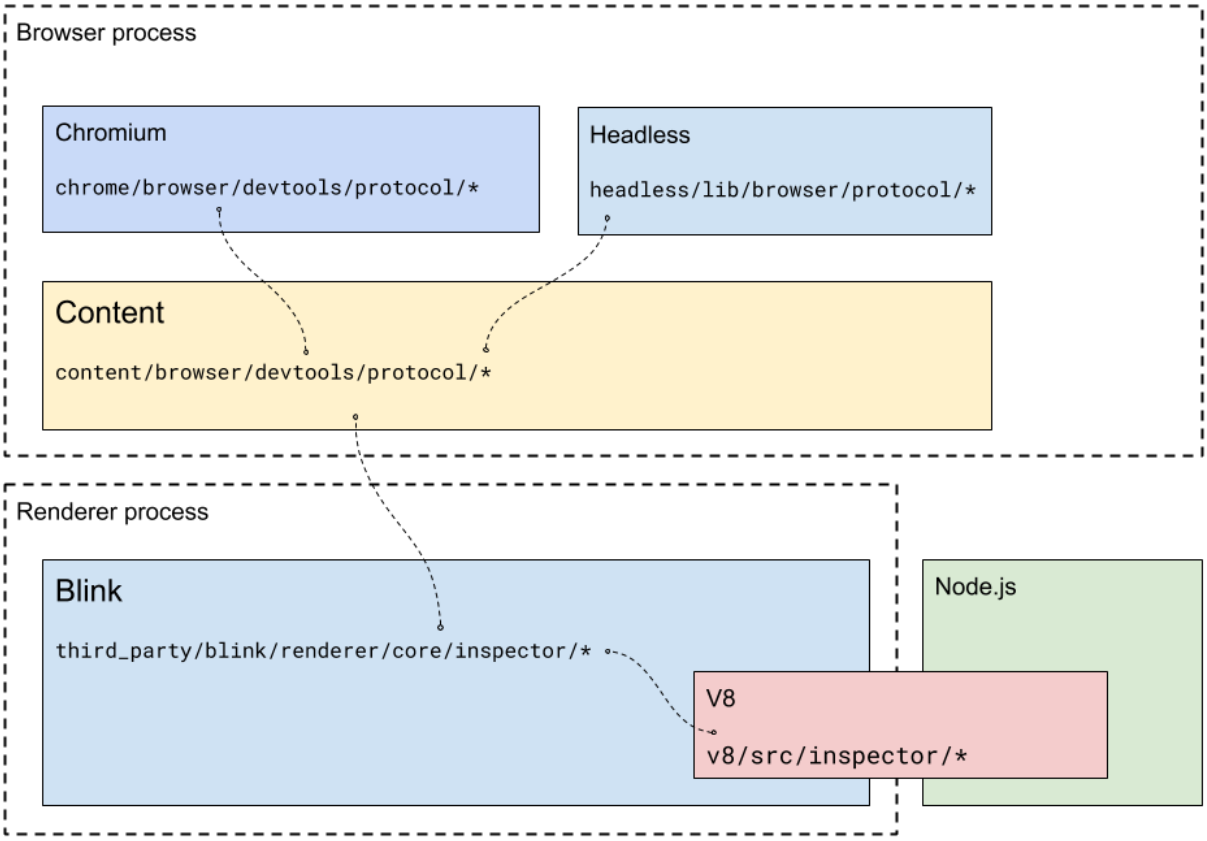
List of existing configuration files:

- `chrome/` layer: inspector_protocol_config.json
- `headless/` layer: protocol_config.json
- `content/` layer: protocol_config.json
- `blink/` layer: inspector_protocol_config.json

Definition and **Configuration** files are used to generate C++ base classes for protocol implementation. They are also used to typecheck the DevTools front-end.

Implementation spans multiple layers and can be found in the following locations:

- `chrome/` layer: chrome/browser/devtools/protocol/
- `headless/` layer: headless/lib/browser/protocol/
- `content/` layer content/browser/devtools/protocol/
- `blink/` layer: renderer/core/inspector/
- `v8/` layer: v8/src/inspector/



Examples of protocol handlers (implementations):

- `chrome/` layer handler: window_manager_handler.h
- `headless/` layer handler: headless_handler.h
- `content/` layer handler: tracing_handler.h
- `blink/` layer handler: inspector_css_agent.h
- `v8/` layer handler: v8-runtime-agent-impl.h

Protocol Generation

Protocol definition and configuration files are used to generate base C++ classes for protocol implementation. Protocol implementations extend generated base classes and override stub implementations.

It's convenient to look into generated classes to get method signatures. Here are the locations of the generated files, assuming the Chromium build folder is `out/Debug`:

- `chrome/` layer: out/Debug/gen/chrome/browser/devtools/protocol/
- `headless/` layer: Debug/gen/headless/lib/browser/protocol/

- `content/` layer: out/Debug/gen/content/browser/devtools/protocol
- `blink/` layer: out/Debug/gen/third_party/blink/renderer/core/inspector/protocol/
- `v8/` layer: out/Debug/gen/v8/src/inspector/protocol/

Handler Lifecycle

Each protocol handler gets created when a debugging session is started, and gets destroyed when the session it belongs to goes away.

However, with Chromium adopting out-of-process iframes, a cross-process navigation results in a renderer process swapped with another one. Subsequently, renderer-based protocol handlers, such as `blink/` and `v8/`, are destroyed in an old process and re-created in a new one.

In order to carry a state from an old process into a new one, render-based protocol handlers should store the state of interest in a `state_` variable. Once the agent is re-created due to a cross-process navigation, its `::restore` method gets called, giving it a chance to restore. Example: https://crrev.com/548598

Best practices for **all protocol handlers:**

- All initialization is done in handler constructors.
- Tearing down must be done in a `::disable` method.

Best practices for **renderer-based protocol handlers:**

- Use `state_` to keep the state that should survive cross-process navigation.
- Restore handler state in the `::restore` method.

Protocol Tests

Since DevTools protocol implementation spans multiple layers, it reuses testing infrastructure specific to each layer.

Layer	Harness	Test Location
`chrome/`	C++ GUnit that drives JS tests	C++: devtools_sanity_browsertest.cc JS: devtools/front_end/Tests.js
`headless/`	C++ GUnit	headless_devtools_client_browsertest.cpp
`content/` and	DevTools Harness	http/tests/inspector-protocol/

`blink/`	(JavaScript)	inspector-protocol/
`v8/`	V8 harness (JavaScript)	v8/test/inspector/

Adding a Protocol Method

1. Add a new method to the protocol definition.
    a. Use js_protocol.pdl if doing a debugger method, or browser_protocol.pdl otherwise.
2. Whitelist the method in a relevant protocol configuration file (see above).
3. Add method implementation to the relevant protocol handler.
4. Add a inspector-protocol test to validate the functionality.

Adding a Protocol Domain

There are a lot of details here. Contact the DevTools owners.

Example CLs

While examining the CLs below, pay attention to:

- protocol PDL and configuration files
- agents and handlers
- newly added or changed tests

All of the following examples add protocol methods with implementations in different layers.

- `chrome/` and `headless/` layers: https://crrev.com/556977
- `content/` layer: https://crrev.com/540120
- `blink/` layer: https://crrev.com/548598
- `v8/` layer: https://chromium-review.googlesource.com/c/v8/v8/+/1077662

Code Notes

- In a browser process, targets are represented as instances of DevToolsAgentHost subclasses.