Debugger

Debugger domain exposes JavaScript debugging capabilities. It allows setting and removing breakpoints, stepping through execution, exploring stack traces, etc.

Types

Generally, you do not need to instantiate CDP types yourself. Instead, the API creates objects for you as return values from commands, and then you can use those objects as arguments to other commands.

class BreakpointId [source]

Breakpoint identifier.

class CallFrameId [source]

Call frame identifier.

class Location(script_id, line_number, column_number=None) [source]

Location in the source code.

script_id: ScriptId

Script identifier as reported in the `Debugger.scriptParsed`.

line_number: int

Line number in the script (0-based).

column_number: Optional[int] = None

Column number in the script (0-based).

class ScriptPosition(line_number, column_number) [source]

Location in the source code.

line_number: int

column_number: int

class LocationRange(script_id, start, end) [source]

Location range within one script.

script_id: ScriptId

start: ScriptPosition

end: ScriptPosition

class CallFrame(call_frame_id, function_name, location, url, scope_chain, this, function_location=None, return_value=None, can_be_restarted=None) [source]

JavaScript call frame. Array of call frames form the call stack.

call_frame_id: `CallFrameId`

Call frame identifier. This identifier is only valid while the virtual machine is paused.

function_name: `str`

Name of the JavaScript function called on this call frame.

location: `Location`

Location in the source code.

url: `str`

JavaScript script name or url. Deprecated in favor of using the `location.scriptId` to resolve the URL via a previously sent `Debugger.scriptParsed` event.

scope_chain: `List[Scope]`

Scope chain for this call frame.

this: `RemoteObject`

`this` object for this call frame.

function_location: `Optional[Location] = None`

Location in the source code.

return_value: `Optional[RemoteObject] = None`

The value being returned, if the function is at return point.

can_be_restarted: `Optional[bool] = None`

Valid only while the VM is paused and indicates whether this frame can be restarted or not. Note that a `true` value here does not guarantee that Debugger#restartFrame with this CallFrameId will be successful, but it is very likely.

```
class Scope(type_, object_, name=None, start_location=None, end_location=None)
```
[source]

Scope description.

type_: str

Scope type.

object_: RemoteObject

Object representing the scope. For `global` and `with` scopes it represents the actual object; for the rest of the scopes, it is artificial transient object enumerating scope variables as its properties.

name: Optional[str] = None

start_location: Optional[Location] = None

Location in the source code where scope starts

end_location: Optional[Location] = None

Location in the source code where scope ends

class SearchMatch(line_number, line_content) [source]

Search match for resource.

line_number: float

Line number in resource content.

line_content: str

Line with match content.

class BreakLocation(script_id, line_number, column_number=None, type_=None) [source]

script_id: ScriptId

Script identifier as reported in the `Debugger.scriptParsed`.

line_number: int

Line number in the script (0-based).

column_number: Optional[int] = None

Column number in the script (0-based).

type_: Optional[str] = None

class WasmDisassemblyChunk(lines, bytecode_offsets) [source]

lines: List[str]

The next chunk of disassembled lines.

bytecode_offsets: List[int]

The bytecode offsets describing the start of each line.

class **ScriptLanguage**(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None) [source]

Enum of possible script languages.

JAVA_SCRIPT = 'JavaScript'

WEB_ASSEMBLY = 'WebAssembly'

class **DebugSymbols**(type_, external_url=None) [source]

Debug symbols available for a wasm script.

type_: str

Type of the debug symbols.

external_url: Optional[str] = None

URL of the external symbol source.

Commands

Each command is a generator function. The return type `Generator[x, y, z]` indicates that the generator *yields* arguments of type `x`, it must be resumed with an argument of type `y`, and it returns type `z`. In this library, types `x` and `y` are the same for all commands, and `z` is the return type you should pay attention to. For more information, see Getting Started: Commands.

`continue_to_location(location, target_call_frames=None)` [source]

Continues execution until specific location is reached.

PARAMETERS:

- **location** (`Location`) – Location to continue to.
- **target_call_frames** (`Optional[str]`) – *(Optional)*

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

`disable()` [source]

Disables debugger for given page.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

`disassemble_wasm_module(script_id)` [source]

EXPERIMENTAL

PARAMETERS:

script_id (`ScriptId`) – Id of the script to disassemble

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], Tuple[Optional[str], int, List[int], WasmDisassemblyChunk]]`

RETURNS:

A tuple with the following items:

0. **streamId** - *(Optional)* For large modules, return a stream from which additional chunks of disassembly can be read successively.
1. **totalNumberOfLines** - The total number of lines in the disassembly text.
2. **functionBodyOffsets** - The offsets of all function bodies, in the format [start1, end1, start2, end2, ...] where all ends are exclusive.
3. **chunk** - The first chunk of disassembly.

`enable(max_scripts_cache_size=None)` [source]

Enables debugger for the given page. Clients should not assume that the debugging has been enabled until the result for this command is received.

PARAMETERS:

max_scripts_cache_size (`Optional`[`float`]) – **(EXPERIMENTAL)** *(Optional)* The maximum size in bytes of collected scripts (not referenced by other heap objects) the debugger can hold. Puts no limit if parameter is omitted.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `UniqueDebuggerId`]

RETURNS:

Unique identifier of the debugger.

```
evaluate_on_call_frame(call_frame_id, expression, object_group=None,
    include_command_line_api=None, silent=None, return_by_value=None,
    generate_preview=None, throw_on_side_effect=None, timeout=None)      [source]
```

Evaluates expression on a given call frame.

PARAMETERS:

- **call_frame_id** (`CallFrameId`) – Call frame identifier to evaluate on.
- **expression** (`str`) – Expression to evaluate.
- **object_group** (`Optional`[`str`]) – *(Optional)* String object group name to put result into (allows rapid releasing resulting object handles using ``releaseObjectGroup``).
- **include_command_line_api** (`Optional`[`bool`]) – *(Optional)* Specifies whether command line API should be available to the evaluated expression, defaults to false.
- **silent** (`Optional`[`bool`]) – *(Optional)* In silent mode exceptions thrown during evaluation are not reported and do not pause execution. Overrides ``setPauseOnException`` state.
- **return_by_value** (`Optional`[`bool`]) – *(Optional)* Whether the result is expected to be a JSON object that should be sent by value.
- **generate_preview** (`Optional`[`bool`]) – **(EXPERIMENTAL)** *(Optional)* Whether preview should be generated for the result.
- **throw_on_side_effect** (`Optional`[`bool`]) – *(Optional)* Whether to throw an exception if side effect cannot be ruled out during evaluation.
- **timeout** (`Optional`[`TimeDelta`]) – **(EXPERIMENTAL)** *(Optional)* Terminate execution after timing out (number of milliseconds).

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `Tuple`[`RemoteObject`, `Optional`[`ExceptionDetails`]]]

RETURNS:

A tuple with the following items:

0. **result** - Object wrapper for the evaluation result.
1. **exceptionDetails** - *(Optional)* Exception details.

get_possible_breakpoints(start, end=None, restrict_to_function=None) [source]

Returns possible locations for breakpoint. scriptId in start and end range locations

should be the same.

PARAMETERS:

- **start** (`Location`) – Start of range to search possible breakpoint locations in.
- **end** (`Optional`[`Location`]) – *(Optional)* End of range to search possible breakpoint locations in (excluding). When not specified, end of scripts is used as end of range.
- **restrict_to_function** (`Optional`[`bool`]) – *(Optional)* Only consider locations which are in the same (non-nested) function as start.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `List`[`BreakLocation`]]

RETURNS:

List of the possible breakpoint locations.

get_script_source(script_id) [source]

Returns source for the script with given id.

PARAMETERS:

script_id (`ScriptId`) – Id of the script to get source for.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `Tuple`[`str`, `Optional`[`str`]]]

RETURNS:

A tuple with the following items:

0. **scriptSource** - Script source (empty in case of Wasm bytecode).
1. **bytecode** - *(Optional)* Wasm bytecode. (Encoded as a base64 string when passed over JSON)

get_stack_trace(stack_trace_id) [source]

Returns stack trace with given `stackTraceId`.

EXPERIMENTAL

PARAMETERS:

stack_trace_id (`StackTraceId`) –

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `StackTrace`]

RETURNS:

get_wasm_bytecode(script_id) [source]

This command is deprecated. Use getScriptSource instead.

Deprecated since version 1.3.

PARAMETERS:

script_id (`ScriptId`) – Id of the Wasm script to get source for.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], str]`

RETURNS:

Script source. (Encoded as a base64 string when passed over JSON)

Deprecated since version 1.3.

next_wasm_disassembly_chunk(stream_id) [source]

Disassemble the next chunk of lines for the module corresponding to the stream. If disassembly is complete, this API will invalidate the streamId and return an empty chunk. Any subsequent calls for the now invalid stream will return errors.

EXPERIMENTAL

PARAMETERS:

stream_id (`str`) –

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], WasmDisassemblyChunk]`

RETURNS:

The next chunk of disassembly.

pause() [source]

Stops on the next JavaScript statement.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

pause_on_async_call(parent_stack_trace_id) [source]

Deprecated since version 1.3.

EXPERIMENTAL

PARAMETERS:

parent_stack_trace_id (`StackTraceId`) – Debugger will pause when async call with given stack trace is started.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

Deprecated since version 1.3.

remove_breakpoint(breakpoint_id) [source]

Removes JavaScript breakpoint.

PARAMETERS:

breakpoint_id (`BreakpointId`) –

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

restart_frame(call_frame_id, mode=None) [source]

Restarts particular call frame from the beginning. The old, deprecated behavior of `restartFrame` is to stay paused and allow further CDP commands after a restart was scheduled. This can cause problems with restarting, so we now continue execution immediatly after it has been scheduled until we reach the beginning of the restarted frame.

To stay back-wards compatible, `restartFrame` now expects a `mode` parameter to be present. If the `mode` parameter is missing, `restartFrame` errors out.

The various return values are deprecated and `callFrames` is always empty. Use the call frames from the `Debugger#paused` events instead, that fires once V8 pauses at the beginning of the restarted function.

PARAMETERS:

- **call_frame_id** (`CallFrameId`) – Call frame identifier to evaluate on.
- **mode** (`Optional[str]`) – **(EXPERIMENTAL)** *(Optional)* The ``` `mode` ``` parameter must be present and set to ‘StepInto’, otherwise ``` ``restartFrame` ``` will error out.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], Tuple[List[CallFrame], Optional[StackTrace], Optional[StackTraceId]]]`

RETURNS:

A tuple with the following items:

0. **callFrames** - New stack trace.
1. **asyncStackTrace** - *(Optional)* Async stack trace, if any.
2. **asyncStackTraceId** - *(Optional)* Async stack trace, if any.

resume(terminate_on_resume=None) [source]

Resumes JavaScript execution.

PARAMETERS:

terminate_on_resume (`Optional`[`bool`]) – *(Optional)* Set to true to terminate execution upon resuming execution. In contrast to Runtime.terminateExecution, this will allows to execute further JavaScript (i.e. via evaluation) until execution of the paused code is actually resumed, at which point termination is triggered. If execution is currently not paused, this parameter has no effect.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `None`]

search_in_content(script_id, query, case_sensitive=None, is_regex=None) [source]

Searches for given string in script content.

PARAMETERS:

- **script_id** (`ScriptId`) – Id of the script to search in.
- **query** (`str`) – String to search for.
- **case_sensitive** (`Optional`[`bool`]) – *(Optional)* If true, search is case sensitive.
- **is_regex** (`Optional`[`bool`]) – *(Optional)* If true, treats string parameter as regex.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `List`[`SearchMatch`]]

RETURNS:

List of search matches.

set_async_call_stack_depth(max_depth) [source]

Enables or disables async call stacks tracking.

PARAMETERS:

max_depth (`int`) – Maximum depth of async call stacks. Setting to `` `0` `` will effectively disable collecting async call stacks (default).

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `None`]

set_blackbox_execution_contexts(unique_ids) [source]

Replace previous blackbox execution contexts with passed ones. Forces backend to skip stepping/pausing in scripts in these execution contexts. VM will try to leave blackboxed script by performing ‘step in’ several times, finally resorting to ‘step out’ if unsuccessful.

EXPERIMENTAL

PARAMETERS:

unique_ids (`List[str]`) – Array of execution context unique ids for the debugger to ignore.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

`set_blackbox_patterns(patterns, skip_anonymous=None)` [source]

Replace previous blackbox patterns with passed ones. Forces backend to skip stepping/pausing in scripts with url matching one of the patterns. VM will try to leave blackboxed script by performing ‘step in’ several times, finally resorting to ‘step out’ if unsuccessful.

EXPERIMENTAL

PARAMETERS:

- **patterns** (`List[str]`) – Array of regexps that will be used to check script url for blackbox state.
- **skip_anonymous** (`Optional[bool]`) – *(Optional)* If true, also ignore scripts with no source url.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

`set_blackboxed_ranges(script_id, positions)` [source]

Makes backend skip steps in the script in blackboxed ranges. VM will try leave blacklisted scripts by performing ‘step in’ several times, finally resorting to ‘step out’ if unsuccessful. Positions array contains positions where blackbox state is changed. First interval isn’t blackboxed. Array should be sorted.

EXPERIMENTAL

PARAMETERS:

- **script_id** (`ScriptId`) – Id of the script.
- **positions** (`List[ScriptPosition]`) –

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

`set_breakpoint(location, condition=None)` [source]

Sets JavaScript breakpoint at a given location.

PARAMETERS:

- **location** (`Location`) – Location to set breakpoint in.
- **condition** (`Optional`[`str`]) – *(Optional)* Expression to use as a breakpoint condition. When specified, debugger will only stop on the breakpoint if this expression evaluates to true.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `Tuple`[`BreakpointId`, `Location`]]

RETURNS:

A tuple with the following items:

0. **breakpointId** - Id of the created breakpoint for further reference.
1. **actualLocation** - Location this breakpoint resolved into.

```
set_breakpoint_by_url(line_number, url=None, url_regex=None, script_hash=None,
    column_number=None, condition=None)                                [source]
```

Sets JavaScript breakpoint at given location specified either by URL or URL regex. Once this command is issued, all existing parsed scripts will have breakpoints resolved and returned in `locations` property. Further matching script parsing will

result in subsequent `breakpointResolved` events issued. This logical breakpoint will survive page reloads.

PARAMETERS:

- **line_number** (`int`) – Line number to set breakpoint at.
- **url** (`Optional`[`str`]) – *(Optional)* URL of the resources to set breakpoint on.
- **url_regex** (`Optional`[`str`]) – *(Optional)* Regex pattern for the URLs of the resources to set breakpoints on. Either `` `url`` `` or `` ``urlRegex` `` must be specified.
- **script_hash** (`Optional`[`str`]) – *(Optional)* Script hash of the resources to set breakpoint on.
- **column_number** (`Optional`[`int`]) – *(Optional)* Offset in the line to set breakpoint at.
- **condition** (`Optional`[`str`]) – *(Optional)* Expression to use as a breakpoint condition. When specified, debugger will only stop on the breakpoint if this expression evaluates to true.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `Tuple`[`BreakpointId`, `List`[`Location`]]]

RETURNS:

A tuple with the following items:

0. **breakpointId** - Id of the created breakpoint for further reference.
1. **locations** - List of the locations this breakpoint resolved into upon addition.

`set_breakpoint_on_function_call(object_id, condition=None)` [source]

Sets JavaScript breakpoint before each call to the given function. If another function was created from the same source as a given one, calling it will also trigger the breakpoint.

EXPERIMENTAL

PARAMETERS:

- **object_id** (`RemoteObjectId`) – Function object id.
- **condition** (`Optional`[`str`]) – *(Optional)* Expression to use as a breakpoint condition. When specified, debugger will stop on the breakpoint if this expression evaluates to true.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `BreakpointId`]

RETURNS:

Id of the created breakpoint for further reference.

`set_breakpoints_active(active)` [source]

Activates / deactivates all breakpoints on the page.

PARAMETERS:

active (`bool`) – New value for breakpoints active state.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

set_instrumentation_breakpoint(instrumentation) [source]

Sets instrumentation breakpoint.

PARAMETERS:

instrumentation (`str`) – Instrumentation name.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], BreakpointId]`

RETURNS:

Id of the created breakpoint for further reference.

set_pause_on_exceptions(state) [source]

Defines pause on exceptions state. Can be set to stop on all exceptions, uncaught exceptions, or caught exceptions, no exceptions. Initial pause on exceptions state is `none`.

PARAMETERS:

state (`str`) – Pause on exceptions mode.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

set_return_value(new_value) [source]

Changes return value in top frame. Available only at return break position.

EXPERIMENTAL

PARAMETERS:

new_value (`CallArgument`) – New return value.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

set_script_source(script_id, script_source, dry_run=None, allow_top_frame_editing=None) [source]

Edits JavaScript source live.

In general, functions that are currently on the stack can not be edited with a single exception: If the edited function is the top-most stack frame and that is the only

activation of that function on the stack. In this case the live edit will be successful and a `Debugger.restartFrame` for the top-most function is automatically triggered.

PARAMETERS:

- **script_id** (`ScriptId`) – Id of the script to edit.
- **script_source** (`str`) – New content of the script.
- **dry_run** (`Optional`[`bool`]) – *(Optional)* If true the change will not actually be applied. Dry run may be used to get result description without actually modifying the code.
- **allow_top_frame_editing** (`Optional`[`bool`]) – **(EXPERIMENTAL)** *(Optional)* If true, then `` `scriptSource` `` is allowed to change the function on top of the stack as long as the top-most stack frame is the only activation of that function.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `Tuple`[`Optional`[`List`[`CallFrame`]], `Optional`[`bool`], `Optional`[`StackTrace`], `Optional`[`StackTraceId`], `str`, `Optional`[`ExceptionDetails`]]]

RETURNS:

A tuple with the following items:

0. **callFrames** - *(Optional)* New stack trace in case editing has happened while VM was stopped.
1. **stackChanged** - *(Optional)* Whether current call stack was modified after applying the changes.
2. **asyncStackTrace** - *(Optional)* Async stack trace, if any.
3. **asyncStackTraceId** - *(Optional)* Async stack trace, if any.
4. **status** - Whether the operation was successful or not. Only `` Ok`` denotes a successful live edit while the other enum variants denote why the live edit failed.
5. **exceptionDetails** - *(Optional)* Exception details if any. Only present when `` status`` is `` CompileError`.

set_skip_all_pauses(skip) [source]

Makes page not interrupt on any pauses (breakpoint, exception, dom exception etc).

PARAMETERS:

skip (`bool`) – New value for skip pauses state.

RETURN TYPE:

`Generator`[`Dict`[`str`, `Any`], `Dict`[`str`, `Any`], `None`]

set_variable_value(scope_number, variable_name, new_value, call_frame_id)[source]

Changes value of variable in a callframe. Object-based scopes are not supported and must be mutated manually.

PARAMETERS:

- **scope_number** (`int`) – 0-based number of scope as was listed in scope chain. Only ‘local’, ‘closure’ and ‘catch’ scope types are allowed. Other scopes could be manipulated manually.
- **variable_name** (`str`) – Variable name.
- **new_value** (`CallArgument`) – New variable value.
- **call_frame_id** (`CallFrameId`) – Id of callframe that holds variable.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

step_into(break_on_async_call=None, skip_list=None) [source]

Steps into the function call.

PARAMETERS:

- **break_on_async_call** (`Optional[bool]`) – **(EXPERIMENTAL)** *(Optional)* Debugger will pause on the execution of the first async task which was scheduled before next pause.
- **skip_list** (`Optional[List[LocationRange]]`) – **(EXPERIMENTAL)** *(Optional)* The skipList specifies location ranges that should be skipped on step into.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

step_out() [source]

Steps out of the function call.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

step_over(skip_list=None) [source]

Steps over the statement.

PARAMETERS:

- **skip_list** (`Optional[List[LocationRange]]`) – **(EXPERIMENTAL)** *(Optional)* The skipList specifies location ranges that should be skipped on step over.

RETURN TYPE:

`Generator[Dict[str, Any], Dict[str, Any], None]`

Events

Generally, you do not need to instantiate CDP events yourself. Instead, the API creates events for you and then you use the event's attributes.

class **BreakpointResolved**(breakpoint_id, location) [source]

Fired when breakpoint is resolved to an actual script and location.

breakpoint_id: BreakpointId

Breakpoint unique identifier.

location: Location

Actual breakpoint location.

class **Paused**(call_frames, reason, data, hit_breakpoints, async_stack_trace, async_stack_trace_id, async_call_stack_trace_id) [source]

Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

call_frames: List [CallFrame]

Call stack the virtual machine stopped on.

reason: str

Pause reason.

data: Optional [dict]

Object containing break-specific auxiliary properties.

hit_breakpoints: Optional [List [str]]

Hit breakpoints IDs

async_stack_trace: Optional [StackTrace]

Async stack trace, if any.

async_stack_trace_id: Optional [StackTraceId]

Async stack trace, if any.

async_call_stack_trace_id: Optional [StackTraceId]

Never present, will be removed.

class **Resumed** [source]

Fired when the virtual machine resumed execution.

class **ScriptFailedToParse**(script_id, url, start_line, start_column, end_line, end_column, execution_context_id, hash_, execution_context_aux_data,

source_map_url, has_source_url, is_module, length, stack_trace, code_offset, script_language, embedder_name) [source]

Fired when virtual machine fails to parse the script.

script_id: ScriptId

Identifier of the script parsed.

url: str

URL or name of the script parsed (if any).

start_line: int

Line offset of the script within the resource with given URL (for script tags).

start_column: int

Column offset of the script within the resource with given URL.

end_line: int

Last line of the script.

end_column: int

Length of the last line of the script.

execution_context_id: ExecutionContextId

Specifies script creation context.

hash_: str

Content hash of the script, SHA-256.

execution_context_aux_data: Optional[dict]

‘default’``’isolated’``’worker’, frameId: string}

TYPE:

Embedder-specific auxiliary data likely matching {isDefault

TYPE:

boolean, type

source_map_url: Optional[str]

URL of source map associated with script (if any).

has_source_url: Optional[bool]

True, if this script has sourceURL.

is_module: Optional[bool]

True, if this script is ES6 module.

length: Optional[int]

This script length.

stack_trace: Optional [StackTrace]

JavaScript top stack frame of where the script parsed event was triggered if available.

code_offset: Optional [int]

If the scriptLanguage is WebAssembly, the code section offset in the module.

script_language: Optional [ScriptLanguage]

The language of the script.

embedder_name: Optional [str]

The name the embedder supplied for this script.

```
class ScriptParsed(script_id, url, start_line, start_column, end_line,
    end_column, execution_context_id, hash_, execution_context_aux_data,
    is_live_edit, source_map_url, has_source_url, is_module, length,
    stack_trace, code_offset, script_language, debug_symbols, embedder_name)
                                                                    [source]
```

Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

script_id: ScriptId

Identifier of the script parsed.

url: str

URL or name of the script parsed (if any).

start_line: int

Line offset of the script within the resource with given URL (for script tags).

start_column: int

Column offset of the script within the resource with given URL.

end_line: int

Last line of the script.

end_column: int

Length of the last line of the script.

execution_context_id: ExecutionContextId

Specifies script creation context.

hash_: str

Content hash of the script, SHA-256.

execution_context_aux_data: Optional[dict]

‘default’``’isolated’``’worker’, frameId: string}

TYPE:

Embedder-specific auxiliary data likely matching {isDefault

TYPE:

boolean, type

is_live_edit: Optional[bool]

True, if this script is generated as a result of the live edit operation.

source_map_url: Optional[str]

URL of source map associated with script (if any).

has_source_url: Optional[bool]

True, if this script has sourceURL.

is_module: Optional[bool]

True, if this script is ES6 module.

length: Optional [int]

This script length.

stack_trace: Optional [StackTrace]

JavaScript top stack frame of where the script parsed event was triggered if available.

code_offset: Optional [int]

If the scriptLanguage is WebAssembly, the code section offset in the module.

script_language: Optional [ScriptLanguage]

The language of the script.

debug_symbols: Optional [List [DebugSymbols]]

If the scriptLanguage is WebAssembly, the source of debug symbols for the module.

embedder_name: Optional [str]

The name the embedder supplied for this script.