# Inspector

Stability: 2 - Stable

The `inspector` module provides an API for interacting with the V8 inspector.

It can be accessed using:

```
const inspector = require('inspector');
```

## `inspector.close()`

Deactivate the inspector. Blocks until there are no active connections.

## `inspector.console`

- {Object} An object to send messages to the remote inspector console.

```
require('inspector').console.log('a message');
```

The inspector console does not have API parity with Node.js console.

## `inspector.open([port[, host[, wait]]])`

- `port` {number} Port to listen on for inspector connections. Optional. **Default:** what was specified on the CLI.
- `host` {string} Host to listen on for inspector connections. Optional. **Default:** what was specified on the CLI.
- `wait` {boolean} Block until a client has connected. Optional. **Default:** `false`.

Activate inspector on host and port. Equivalent to `node --inspect=[[host:]port]`, but can be done programmatically after node has started.

If wait is `true`, will block until a client has connected to the inspect port and flow control has been passed to the debugger client.

See the security warning regarding the `host` parameter usage.

## `inspector.url()`

- Returns: {string|undefined}

Return the URL of the active inspector, or `undefined` if there is none.

```
$ node --inspect -p 'inspector.url()'
Debugger listening on ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34
For help, see: https://nodejs.org/en/docs/inspector
ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34

$ node --inspect=localhost:3000 -p 'inspector.url()'
```

```
Debugger listening on ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a
For help, see: https://nodejs.org/en/docs/inspector
ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a

$ node -p 'inspector.url()'
undefined
```

### `inspector.waitForDebugger()`

Blocks until a client (existing or connected later) has sent `Runtime.runIfWaitingForDebugger` command.

An exception will be thrown if there is no active inspector.

## Class: `inspector.Session`

- Extends: {EventEmitter}

The `inspector.Session` is used for dispatching messages to the V8 inspector back-end and receiving message responses and notifications.

#### `new inspector.Session()`

Create a new instance of the `inspector.Session` class. The inspector session needs to be connected through `session.connect()` before the messages can be dispatched to the inspector backend.

#### Event: `'inspectorNotification'`

- {Object} The notification message object

Emitted when any notification from the V8 Inspector is received.

```
session.on('inspectorNotification', (message) => console.log(message.method));
// Debugger.paused
// Debugger.resumed
```

It is also possible to subscribe only to notifications with specific method:

#### Event: `<inspector-protocol-method>;`

- {Object} The notification message object

Emitted when an inspector notification is received that has its method field set to the `<inspector-protocol-method>` value.

The following snippet installs a listener on the `'Debugger.paused'` event, and prints the reason for program suspension whenever program execution is suspended (through breakpoints, for example):

```
session.on('Debugger.paused', ({ params }) => {
  console.log(params.hitBreakpoints);
});
// [ '/the/file/that/has/the/breakpoint.js:11:0' ]
```

#### `session.connect()`

Connects a session to the inspector back-end.

#### `session.connectToMainThread()`

Connects a session to the main thread inspector back-end. An exception will be thrown if this API was not called on a Worker thread.

#### `session.disconnect()`

Immediately close the session. All pending message callbacks will be called with an error. `session.connect()` will need to be called to be able to send messages again. Reconnected session will lose all inspector state, such as enabled agents or configured breakpoints.

#### `session.post(method[, params][, callback])`

- `method` {string}
- `params` {Object}
- `callback` {Function}

Posts a message to the inspector back-end. `callback` will be notified when a response is received. `callback` is a function that accepts two optional arguments: error and message-specific result.

```
session.post('Runtime.evaluate', { expression: '2 + 2' },
             (error, { result }) => console.log(result));
// Output: { type: 'number', value: 4, description: '4' }
```

The latest version of the V8 inspector protocol is published on the Chrome DevTools Protocol Viewer.

Node.js inspector supports all the Chrome DevTools Protocol domains declared by V8. Chrome DevTools Protocol domain provides an interface for interacting with one of the runtime agents used to inspect the application state and listen to the run-time events.

## Example usage

Apart from the debugger, various V8 Profilers are available through the DevTools protocol.

### CPU profiler

Here's an example showing how to use the CPU Profiler:

```
const inspector = require('inspector');
const fs = require('fs');
const session = new inspector.Session();
session.connect();

session.post('Profiler.enable', () => {
  session.post('Profiler.start', () => {
    // Invoke business logic under measurement here...

    // some time later...
    session.post('Profiler.stop', (err, { profile }) => {
      // Write profile to disk, upload, etc.
      if (!err) {
        fs.writeFileSync('./profile.cpuprofile', JSON.stringify(profile));
      }
    });
  });
});
```

### Heap profiler

Here's an example showing how to use the Heap Profiler:

```
const inspector = require('inspector');
const fs = require('fs');
const session = new inspector.Session();

const fd = fs.openSync('profile.heapsnapshot', 'w');

session.connect();

session.on('HeapProfiler.addHeapSnapshotChunk', (m) => {
  fs.writeSync(fd, m.params.chunk);
});

session.post('HeapProfiler.takeHeapSnapshot', null, (err, r) => {
  console.log('HeapProfiler.takeHeapSnapshot done:', err, r);
  session.disconnect();
  fs.closeSync(fd);
});
```