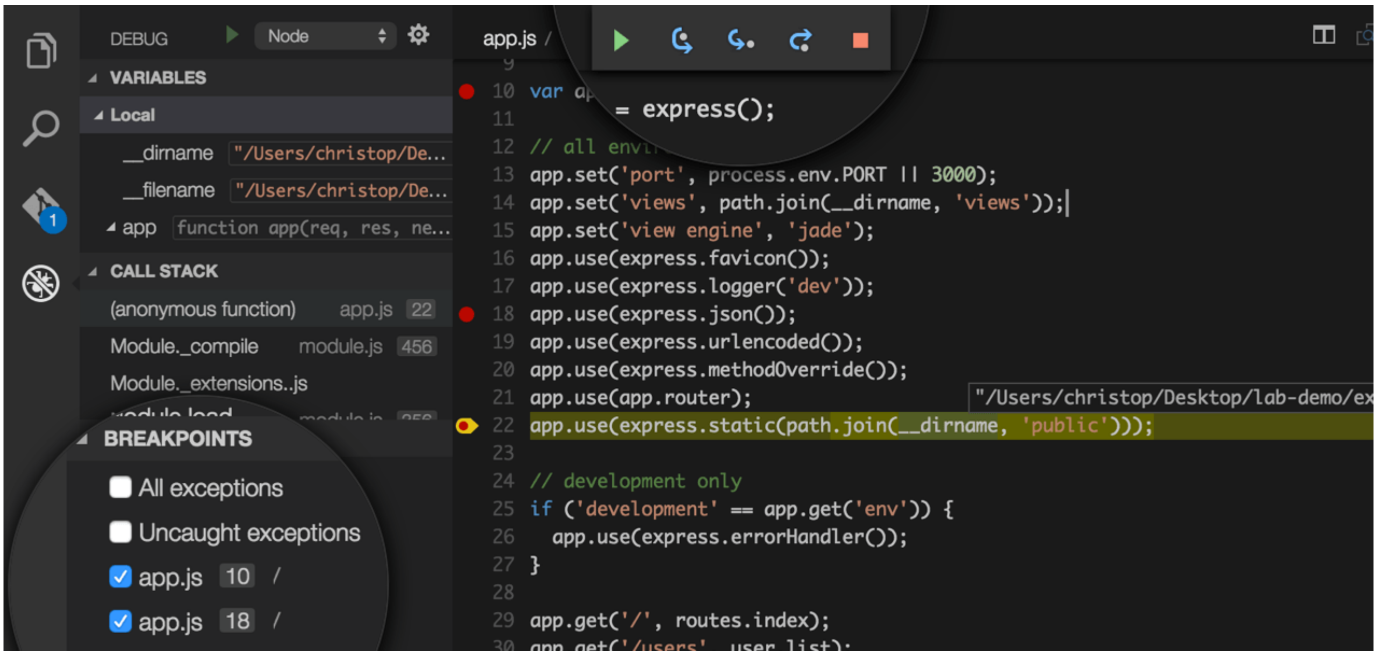
Debugging in Visual Studio Code

Debugging

One of the key features of Visual Studio Code is its great debugging support. VS Code's built-in debugger helps accelerate your edit, compile and debug loop.



```
__dirname   "/Users/christop/De…
__filename  "/Users/christop/De…
app   function app(req, res, ne…
```

```
(anonymous function)   app.js   22
Module._compile        module.js   456
Module._extensions..js
```

```
☐ All exceptions
☐ Uncaught exceptions
☑ app.js 10
☑ app.js 18
```

```
var a              = express();

// all envi
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(app.router);                         "/Users/christop/Desktop/lab-demo/ex
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
```

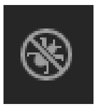
Today we have good debugging support for Node.js (JavaScript and TypeScript) on all platforms and experimental support for Mono (C# and F#) on OS X and Linux. For debugging other languages, please look for `Debuggers` extensions in our VS Code Marketplace.

It is helpful to first create a sample Node.js application before reading about debugging. Follow this guide to do a run-through with Node.js:

Once you are all set up this page will take you through the debugging scenarios we support.

Debug View

To bring up the Debug view, click on the Debugging icon in the View Bar on the side of VS Code.



The Debug view displays all information pertaining to debugging and has a top bar with debugging commands and configuration settings.

Launch Configurations

To debug your app in VS Code, you'll first need to setup your debugging launch configuration file -

`launch.json`. Click on the Configure gear icon on the Debug view top bar, choose your debug environment and VS Code will generate a launch.json. Here is the one generated for Node.js debugging:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch",
            "type": "node",
            "request": "launch",
            "program": "app.js",
            "stopOnEntry": false,
            "args": [],
            "cwd": ".",
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development"
            },
            "externalConsole": false,
            "preLaunchTask": "",
            "sourceMaps": false,
            "outDir": null
        },
        {
            "name": "Attach",
            "type": "node",
            "request": "attach",
            "port": 5858
        }
    ]
}
```

In VS Code we support launching your app in debug mode or attaching to an already running app. Depending on the request (attach or launch) different attributes are required and our launch.json validation and suggestions should help with that.

Review the generated values and make sure that they make sense for your project and debugging environment. You can add additional configurations to the launch.json (use hover and IntelliSense to help).

Select the configuration named `Launch` using the **Configuration dropdown** in the Debug view. Once you have your launch configuration set, start your debug session with F5.

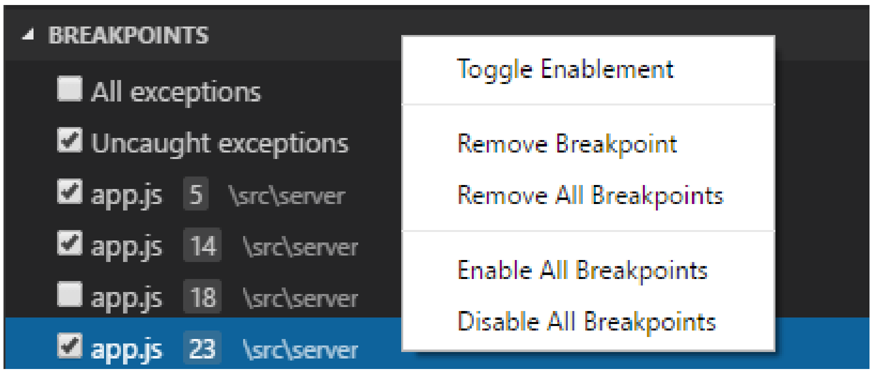
To launch a task before the start of each debug session, set the `preLaunchTask` to the name of one of the tasks specified in tasks.json (located under the workspace's `.vscode` folder).

VS Code supports variable substitution inside strings in launch.json the same way as for tasks.json.

Breakpoints

Breakpoints can be toggled by clicking on the **editor margin**. Finer breakpoint control (enable/disable/reapply) can be done in the Debug view **BREAKPOINTS** section.

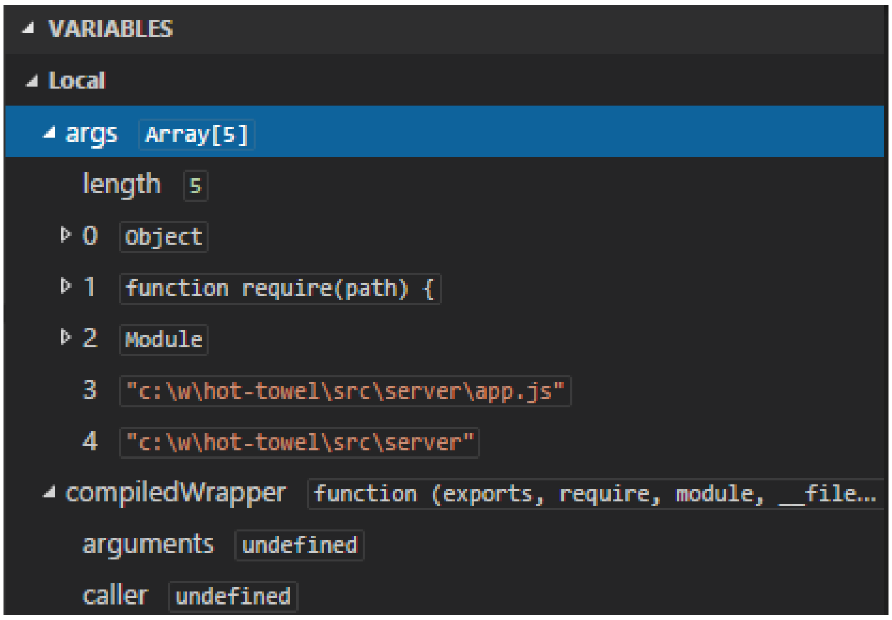
- Breakpoints in the editor margin are normally shown as red filled circles.
- Disabled breakpoints have a filled gray circle.
- When a debugging sessions starts, breakpoints that cannot be registered with the debugger change to a gray hollow circle.



The `Reapply All Breakpoints` command sets all breakpoints again to their original location. This is helpful if your debug environment is "lazy" and "misplaces" breakpoints in code that has not yet been executed. (For details see below under **Node Debugging: Breakpoint Validation**)

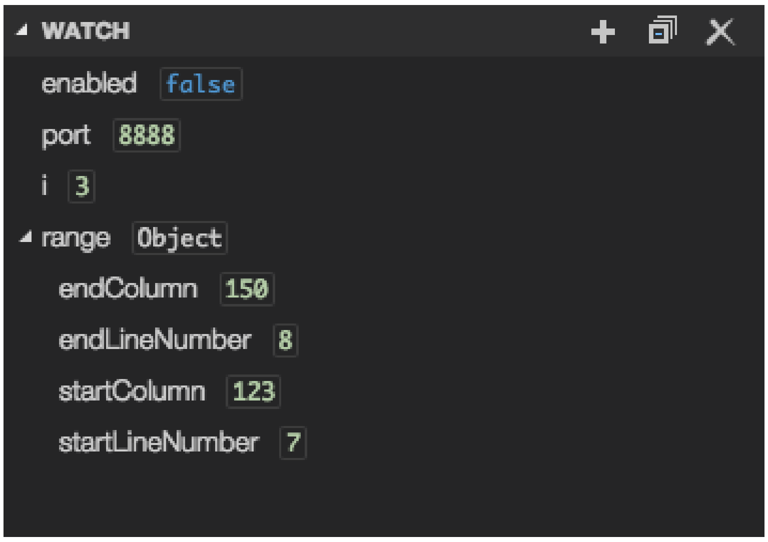
Data inspection

Variables can be inspected in the **VARIABLES** section of the Debug view or by hovering over their source in the editor. Variables and expression evaluation is relative to the selected stack frame in the **CALL STACK** section.



```
VARIABLES
  Local
    args  Array[5]
      length  5
      0  Object
      1  function require(path) {
      2  Module
      3  "c:\w\hot-towel\src\server\app.js"
      4  "c:\w\hot-towel\src\server"
    compiledWrapper  function (exports, require, module, __file…
      arguments  undefined
      caller  undefined
```

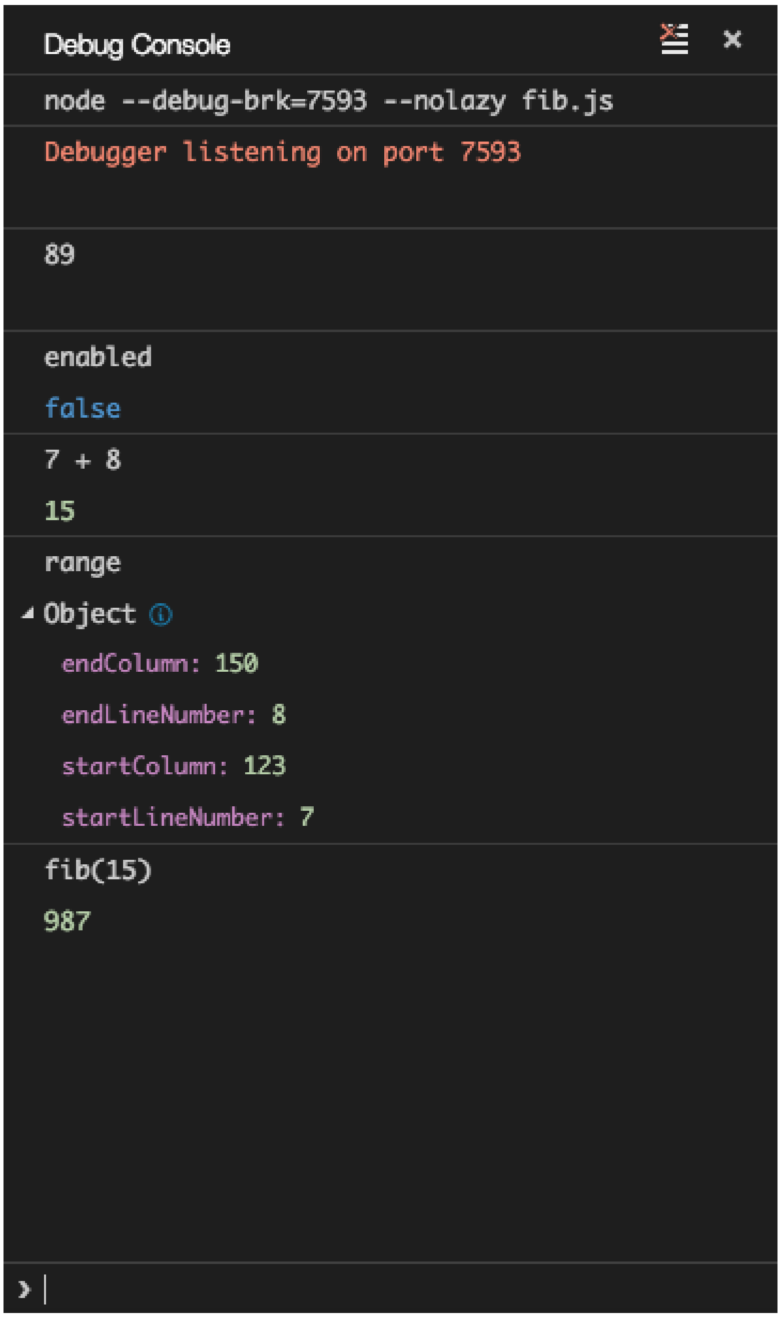
Variables and expressions can also be evaluated and watched in the Debug view **WATCH** section.



```
WATCH
  enabled  false
  port  8888
  i  3
  range  Object
    endColumn  150
    endLineNumber  8
    startColumn  123
    startLineNumber  7
```

Debug Console

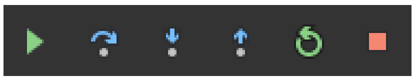
Expressions can be evaluated in the **Debug Console**. To open the Debug Console use the `Open Console` action at the top of the Debug pane or using the Command Palette.



```
node --debug-brk=7593 --nolazy fib.js
Debugger listening on port 7593

89

enabled
false
7 + 8
15
range
◢ Object ⓘ
   endColumn: 150
   endLineNumber: 8
   startColumn: 123
   startLineNumber: 7
fib(15)
987

> |
```

Debug actions

Once a debug session starts, the **Debug actions pane** will appear on the top of the editor.



- Continue / Pause F5
- Step Over F10
- Step Into F11
- Step Out Shift+F11
- Restart Ctrl+Shift+F5
- Stop Shift+F5

Node Debugging

Node Console

By default node debug sessions launch the target in the internal VS Code Debug Console. Since the Debug Console does not support programs that need to read input from the console, you can enable an external, native console by setting the attribute `externalConsole` to `true` in your launch configuration.

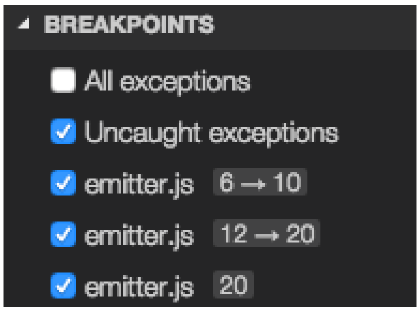
Breakpoint Validation

For performance reasons Node.js parses the functions inside JavaScript files lazily on first access. As a consequence, breakpoints don't work in source code areas that haven't been seen (parsed) by Node.js.

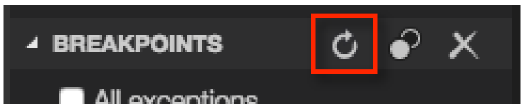
Since this behavior is not ideal for debugging, VS Code passes the `--nolazy` option to Node.js automatically. This prevents the delayed parsing and ensures that breakpoints can be validated before running the code (so they no longer "jump").

Since the `--nolazy` option might increase the start-up time of the debug target significantly, you can easily opt out by passing a `--lazy` as a `runtimeArgs` attribute.

When doing so you will find that some of your breakpoints don't "stick" to the line requested but instead "jump" for the next possible line in already-parsed code. To avoid confusion, VS Code always shows breakpoints at the location where Node.js thinks the breakpoint is. In the breakpoint section, these breakpoints are shown with an arrow between requested and actual line number:



This breakpoint validation occurs when a session starts and the breakpoints are registered with Node.js, or when a session is already running and a new breakpoint is set. In this case, the breakpoint may "jump" to a different location. After Node.js has parsed all the code (e.g. by running through it), breakpoints can be easily re-applied to the requested locations with the Reapply button in the breakpoint section header. This should make the breakpoints "jump back" to the requested location.



JavaScript Source Maps

The Node.js debugger of VS Code supports JavaScript Source Maps which help debugging of transpiled languages, e.g. TypeScript or minified/uglified JavaScript. With source maps, it is possible to single step through or set breakpoints in the original source. If no source map exists for the original source or if the source map is broken and cannot successfully map between the source and the generated JavaScript, the breakpoints are shown as gray hollow circles.

The source map feature is enabled by setting the `sourceMaps` attribute to `true` in the launch configuration. In addition, you can specify a source file (e.g. app.ts) with the `program` attribute. If the generated (transpiled) JavaScript files do not live next to their source but in a separate directory, you can help the VS Code debugger locate them by setting the `outDir` attribute. Whenever you set a breakpoint in the original source, VS Code tries to find the generated source, and the associated source map, in the `outDir` directory.

Since source maps are not automatically created, you must configure the TypeScript compiler to create them:

```
tsc --sourceMap --outDir bin app.ts
```

This is the corresponding launch configuration for a TypeScript program:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch TypeScript",
            "type": "node",
            "request": "attach",
            "program": "app.ts",
            "sourceMaps": true,
            "outDir": "bin"
        }
    ]
}
```

Source maps can be generated with two kinds of inlining:

- **Inlined source maps**: the generated JavaScript file contains the source map as a data URI at the end (instead of referencing the source map through a file URI).
- **Inlined source**: the source map contains the original source (instead of referencing the source through a path).

VS Code supports **inlined source maps** but not **inlined source**.

Attaching VS Code to Node

If you want to attach the VS Code debugger to a Node.js program, launch Node.js as follows:

```
node --debug program.js
node --debug-brk program.js
```

With the `--debug-brk` option Node.js stops on the first line of the program. The corresponding launch configuration looks like this:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Attach to Node",
            "type": "node",
            "request": "attach",
            "port": 5858
        }
    ]
}
```

Mono Debugging

On Linux or OS X, the Mono debugging support of VS Code requires Mono version 3.12 or later. If you intend to build ASP.NET Core applications with Visual Studio Code, we recommend you first follow the steps **Installing ASP.NET Core and DNX** in ASP.NET Core Applications which will install a version of Mono that supports debugging.

If you just want to try VS Code Mono debugging, you can either download the latest Mono version for Linux or OS X at Mono project or you can use your package manager.

- On OS X: `brew install mono`
- On Linux: `sudo apt-get install mono-complete`

Installing Mono debugging support

VS Code Mono debugging integration comes from the 'Mono Debug' extension on the Visual Studio Marketplace.

You can either install the **Mono Debug** extension with the VS Code **Extensions: Install Extension** command or if you already have a Mono based project with a `mono` launch configuration, simply by starting a debug session. VS Code will then prompt you to download and install **Mono Debug**.

Enable Mono debugging

To enable debugging of Mono based C# (and F#) programs, you have to pass the `-debug` option to the compiler:

```
mcs -debug Program.cs
```

If you want to attach the VS Code debugger to a Mono program, pass these additional arguments to the Mono runtime:

```
mono --debug --debugger-
agent=transport=dt_socket,server=y,address=127.0.0.1:55555 Program.exe
```

The corresponding launch configuration looks like this:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Attach to Mono",
            "request": "attach",
            "type": "mono",
            "address": "localhost",
            "port": 55555
        }
    ]
}
```

Next Steps

In case you didn't already read the Node.js section, take a look at:

- Node.js - End to end Node scenario with a sample application

To learn about VS Code's task running support, go to:

- Tasks - Running tasks with Gulp, Grunt and Jake. Showing Errors and Warnings

To write your own debugger extension, visit:

- Debuggers - Steps to create a VS Code debug extension starting from a mock sample

Common Questions

Q: What are the supported debugging scenarios?
A: Debugging of Node.js based applications is supported on Linux, OS X, and Windows. Debugging of C# applications running on Mono is supported on Linux and OS X. ASP.NET Core applications are compiled using the Roslyn compiler, not the Mono compiler. ASP.NET Core debugging will be available

through a VS Code extension.

Q: Why can’t I remote debug my app?
A: Currently we support local debugging only. This is a known limitation. If that’s something you care about, please let us know!

Q: I do not see any launch configurations in the debug view drop down, what is wrong?
A: The most common problem is that you did not set up launch.json yet or there is a syntax error in the launch.json file.

Q: What Node.js version is required for Node.js debugging?
A: Version 0.12.x is recommended, though most functionality is supported in 0.10.x as well (except break on unhandled exceptions).

Q: Is Mono debugging supported on Windows?
A: No. Currently Mono debugging is only supported on Mac and Linux.

Was this documentation helpful?

Yes No

Last updated on 3/7/2016