Integrate with External Tools via Tasks

Lots of tools exist to automate tasks like building, packaging, testing or deploying software systems. Examples include Make (http://en.wikipedia.org/wiki/Make_software), Ant (http://ant.apache.org/), Gulp (http://gulpjs.com/), Jake (http://jakejs.com/), Rake (http://rake.rubyforge.org/) and MSBuild (https://github.com/Microsoft/msbuild).

These tools are mostly run from the command line and automate jobs outside the inner software development loop (edit, compile, test and debug). Given their importance in the development life-cycle, it is very helpful to be able run them and analyze their results from within VS Code.

Examples of Tasks in Action

The best way to highlight the power of Tasks is with a few examples of how VS Code can use Tasks to integrate external tools like linters and compilers.

Transpiling TypeScript to JavaScript

The TypeScript topic (/docs/languages/typescript#_transpiling-typescript-into-javascript) includes an example that creates a task to transpile TypeScript to JavaScript and observe any related errors from within VS Code.

Compiling Markdown to HTML

The Markdown topic provides two examples for compiling Markdown to HTML:

1. Manually compiling with a Build task (/docs/languages/markdown#_compiling-markdown-into-html)
2. Automation of the compile step with a file watcher (/docs/languages/markdown#_automating-markdown-compilation)

Transpiling Less and Sass into CSS

The CSS topic provides examples of how to use Tasks to generate CSS files.

1. Manually transpiling with a Build task (/docs/languages/css#_transpiling-sass-and-less-into-css)
2. Automation of the compile step with a file watcher (/docs/languages/css#_automating-sassless-compilation)

Processing Task Output with Problem Matchers

VS Code processes the output from a task with a problem matcher and we ship with a number of them 'in the box', we'll talk about how to make your own ones soon:

- **TypeScript**: `$tsc` assumes that file names in the output are relative to the opened folder.
- **JSHint**: `$jshint` assumes that file names are reported as an absolute path.
- **JSHint Stylish**: `$jshint-stylish` assumes that file names are reported as an absolute path.
- **ESLint Compact**: `$eslint-compact` assumes that file names in the output are relative to the opened folder.
- **ESLint Stylish**: `$eslint-stylish` assumes that file names in the output are relative to the opened folder.
- **CSharp and VB Compiler**: `$mscompile` assumes that file names are reported as an absolute path.
- **Less**: `$lessCompile` assumes that file names are reported as absolute path.

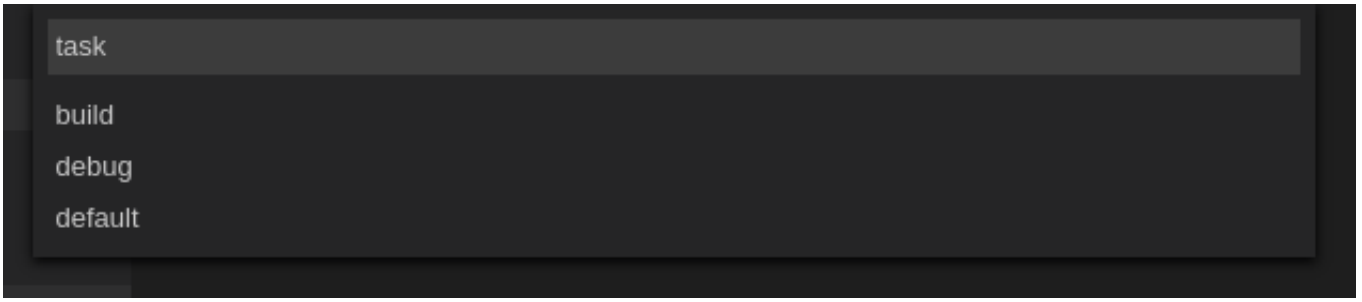
Autodetecting Gulp, Grunt and Jake Tasks

VS Code can autodetect tasks from within Gulp, Grunt and Jake files. This adds their tasks to the task list without requiring additional configuration (unless you need to use a problem matcher, more on that in a moment).

To help make this example more concrete, let's use this simple Gulp file. This defines two tasks: build and debug. The first compiles C# code using Mono (http://www.mono-project.com/)'s compiler. The second starts the MyApp under the Mono debugger.

```
var gulp = require("gulp");

var program = "MyApp";
var port = 55555;

gulp.task('default', ['debug']);

gulp.task('build', function() {
    return gulp
        .src('./**/*.cs')
        .pipe(msc(['-fullpaths', '-debug', '-target:exe', '-out:' + program]));
});

gulp.task('debug', ['build'], function(done) {
    return mono.debug({ port: port, program: program}, done);
});
```

Pressing `Ctrl+Shift+P` and then typing `Run Task` followed by `Enter` will list all available tasks. Selecting one and pressing `Enter` will execute the task.



> **Tip:** Gulp, Grunt and Jake are autodetected only if the corresponding files are present in the root of the opened folder.

Mapping Gulp, Grunt and Jake Output to Problem Matchers

You need to configure the tasks in `tasks.json` if you want to do more than simply run the task. For example, you might want to match reported problems and highlight them within VS Code, or to trigger a build using `Ctrl+Shift+B` .

To do this, we need to edit the `tasks.json` file to 'wrap' the build gulp task that was defined in the gulpfile. This is achieved with the following:

```
{
    "version": "0.1.0",
    "command": "gulp",
    "isShellCommand": true,
    "args": [
        "--no-color"
    ],
    "tasks": [
        {
            "taskName": "build",
            "args": [],
            "isBuildCommand": true,
            "problemMatcher": "$msCompile"
        }
    ]
}
```

In contrast to the `tasks.json` file in the TypeScript section, this file has:

1. We want to run the gulp command in a shell (VS Code directly executing it) so we used **isShellCommand**
2. We added an explicit **tasks** property which allowed us to *optionally* augment a task that was in the gulpfile.
3. We defined a problem matcher **$msCompile** to process the output - since we are compiling C# using the Mono compiler, the built-in one works as *msc* adheres to the Microsoft compiler pattern.

Syntax for the Tasks Property

The tasks property is defined as an array of object literals where each literal has the following properties:

- **taskName** this is the task's name in the Gulp or Jake file.
- **args** a string array of additional arguments to be passed to the task.
- **isBuildCommand** if this property is set to true, `Ctrl+Shift+B` will trigger this task.
- **problemMatcher** a string or array of strings based on the pre-defined problem matchers.

Output Window Behavior

Sometimes you will want to control how the output window behaves when running tasks. For instance, you may always want to show output for the debug command. The property **showOutput** controls this and the valid values are:

- **silent**: the output window is brought to front only if no problem matchers fire for the task. This is the default.
- **always**: the output window is always brought to front.
- **never**: The user must explicitly bring the output window to the using the View menu or the short cut `Ctrl+Shift+U` .

Operating System Specific Properties

The task system supports defining values (for example the command to be executed) specific to an operating system. To do so, simply put an operating system specific literal into the `tasks.json` file and specify the corresponding properties inside that literal.

Below is an example that uses the Node.js executable as a command and is treated differently on Windows and Linux:

```
{
    "version": "0.1.0",
    "windows": {
        "command": "C:\\Program Files\\nodejs\\node.exe"
    },
    "linux": {
        "command": "/usr/bin/node"
    }
}
```

Valid operating properties are **windows** for Windows, **linux** for Linux and **osx** for Mac OS X. Properties defined in an operating system specific scope override properties defined in the global scope. In the example below:

```
{
    "version": "0.1.0",
    "showOutput": "never",
    "windows": {
        "showOutput": "always"
    }
}
```

output from the executed task is never brought to front except for Windows where it is always shown.

Task Specific Properties

The global properties **showOutput** and **suppressTaskName** can be redefined on a task by task basis. The **args** property can be augmented resulting in the additional values being appended to the global arguments.

Here is a example where output for the "deploy" task is always brought to front:

```
{
    "version": "0.1.0",
    "showOutput": "silent",
    "tasks": [
        {
            "taskName": "deploy",
            "showOutput": "always",
        },
        {
        }
    ]
}
```

> **Tip:** If a property is redefined per OS and per task, the one from the task wins.

Defining a Problem Matcher

VS Code ships some of the most common problem matchers out of the box. However, there are lots of compilers and linting tools out there, all of which produce their own style of errors and warnings. So let's talk about how to make your own problem matcher.

We have a `helloWorld.c` program in which the developer mistyped **printf** as **prinft**. Compiling it with gcc (https://gcc.gnu.org/) will produce the following warning:

```
helloWorld.c:5:3: warning: implicit declaration of function ‘prinft’
```

We want to produce a problem matcher that can capture the message in the output and show a corresponding problem in VS Code. Problem matchers heavily rely on regular expressions (http://en.wikipedia.org/wiki/Regular_expression). The section below assumes you are familiar with regular expressions.

> **Tip:** We have found the RegEx101 playground (https://regex101.com/) a really good way to develop and test regular expressions.

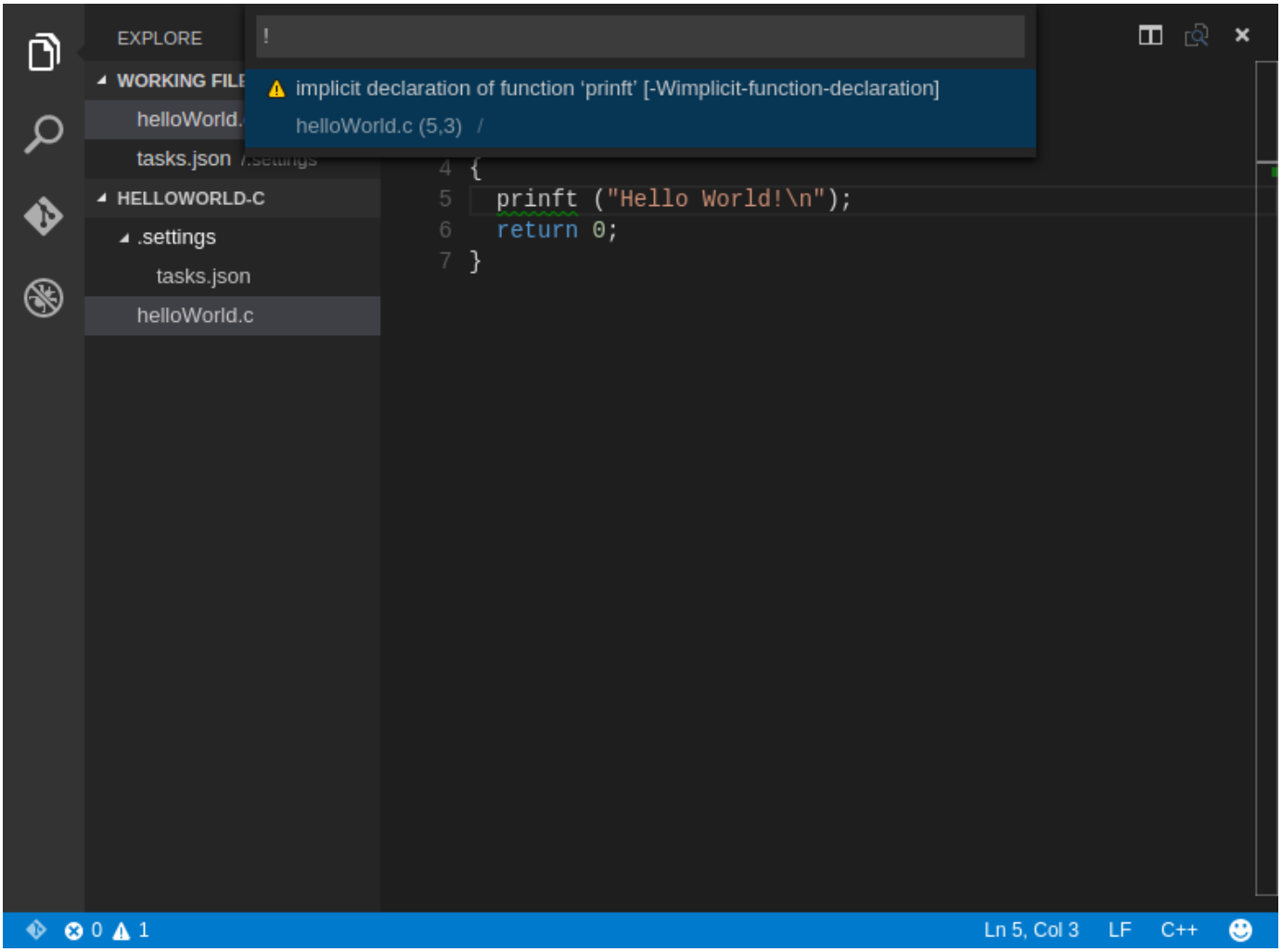
A matcher that captures the above warning (and errors) looks like:

```
{
    // The problem is owned by the cpp language service.
    "owner": "cpp",
    // The file name for reported problems is relative to the opened folder.
    "fileLocation": ["relative", "${workspaceRoot}"],
    // The actual pattern to match problems in the output.
    "pattern": {
        // The regular expression. Example to match: helloWorld.c:5:3: warning: implici
t declaration of function ‘prinft’ [-Wimplicit-function-declaration]
        "regexp": "^(.*):(\\d+):(\\d+):\\s+(warning|error):\\s+(.*)$",
        // The first match group matches the file name which is relative.
        "file": 1,
        // The second match group matches the line on which the problem occurred.
        "line": 2,
        // The third match group matches the column at which the problem occurred.
        "column": 3,
        // The fourth match group matches the problem's severity. Can be ignored. Then
all problems are captured as errors.
        "severity": 4,
        // The fifth match group matches the message.
        "message": 5
    }
}
```

Here is a finished `tasks.json` file with the code above (comments removed) wrapped with the actual task details:

```
{
    "version": "0.1.0",
    "command": "gcc",
    "args": ["-Wall", "helloWorld.c", "-o", "helloWorld"],
    "problemMatcher": {
        "owner": "cpp",
        "fileLocation": ["relative", "${workspaceRoot}"],
        "pattern": {
            "regexp": "^(.*):(\\d+):(\\d+):\\s+(warning|error):\\s+(.*)$",
            "file": 1,
            "line": 2,
            "column": 3,
            "severity": 4,
            "message": 5
        }
    }
}
```

Running it inside VS Code and pressing `Ctrl+Shift+M` to get the list of problems gives you the following output:



There are a couple more properties that can be used inside a pattern. These are:

- **location** if the problem location is line or line,column or startLine,startColumn,endLine,endColumn then our generic location match group can be used.
- **endLine** the match group index for the problem's end line. Can be omitted if no end line value is provided by the compiler.
- **endColumn** the match group index for the problem's end column. Can be omitted if no end column value is provided by the compiler.
- **code** the match group index for the problem's code. Can be omitted if no code value is provided by the compiler.

> **Note:** A functional pattern must at least provide a match group for file, message and line or location.

Defining a Multi-Line Problem Matcher

Some tools spread problems found in a source file over several lines, especially if stylish reporters are used. An example is ESLint (http://eslint.org/); in stylish mode it produces output like this:

```
test.js
  1:0   error  Missing "use strict" statement                strict
✖ 1 problems (1 errors, 0 warnings)
```

Our problem matcher is line-based so we need to capture the file name (test.js) with a different regular expression than the actual problem location and message (1:0 error Missing "use strict" statement).

To do this we use an array of problem patterns for the **pattern** property. This way you define a pattern per each line you want to match.

The following problem pattern matches the output from ESLint in stylish mode - but still has one small issue which we will resolve next. The code below has a first regular expression to capture the file name and the second to capture the line, column, severity, message and error code:

```
{
    "owner": "javascript",
    "fileLocation": ["relative", "${workspaceRoot}"],
    "pattern": [
        {
            "regexp": "^([^\\s].*)$",
            "file": 1
        },
        {
            "regexp": "^\\s+(\\d+):(\\d+)\\s+(error|warning|info)\\s+(.*)\\s\\s+(.*)$",
            "line": 1,
            "column": 2,
            "severity": 3,
            "message": 4,
            "code": 5
        }
    ]
}
```

Of course it's never quite that simple, and this pattern will not work if there is more than one problem on a resource. For instance, imagine the following output from ESLint:

```
test.js
  1:0   error  Missing "use strict" statement                strict
  1:9   error  foo is defined but never used                 no-unused-vars
  2:5   error  x is defined but never used                   no-unused-vars
  2:11  error  Missing semicolon                             semi
  3:1   error  "bar" is not defined                          no-undef
  4:1   error  Newline required at end of file but not found  eol-last
✖ 6 problems (6 errors, 0 warnings)
```

The pattern's first regular expression will match "test.js", the second "1:0 error ...". The next line "1:9 error ..." is processed but not matched by the first regular expression and so no problem is captured.

To make this work, the last regular expression of a multi-line pattern can specify the **loop** property. If set to **true**, it instructs the task system to apply the last pattern of a multi-line matcher to the lines in the ouput as long as the regular expression matches.

The information captured by all previous patterns is combined with the information captured by the last pattern and turned into a problem inside VS Code.

Here is a problem matcher to fully capture ESLint stylish problems:

```
{
    "owner": "javascript",
    "fileLocation": ["relative", "${workspaceRoot}"],
    "pattern": [
        {
            "regexp": "^([^\\s].*)$",
            "file": 1
        },
        {
            "regexp": "^\\s+(\\d+):(\\d+)\\s+(error|warning|info)\\s+(.*)\\s\\s+(.*)$",
            "line": 1,
            "column": 2,
            "severity": 3,
            "message": 4,
            "code": 5,
            "loop": true
        }
    ]
}
```

Variables in tasks.json

When authoring tasks it is often useful to have a set of predefined common variables. VS Code supports variable substitution inside strings in the task.json file and has the following predefined variables:

- **${workspaceRoot}** the path of the folder opened in VS Code
- **${file}** the current opened file
- **${fileBasename}** the current opened file's basename
- **${fileDirname}** the current opened file's dirname
- **${fileExtname}** the current opened file's extension
- **${cwd}** the task runner's current working directory on startup

Below is an example of a configuration that passes the current opened file to the TypeScript compiler.

```
{
    "command": "tsc",

    "args": ["${file}"],
}
```

Next Steps

That was tasks - let's keep going...

- tasks.json Schema (tasks_appendix) - Still want more on tasks dig into the schema to see what else is possible
- Editing Evolved (editingevolved) - Lint, IntelliSense, Lightbulbs, Peek and Goto Definition and more
- Language Support (/docs/languages/overview) - Our Good, Better, Best language grid to see

what you can expect
- Debugging (debugging) - This is where VS Code really shines

Common Questions

Q: Some task runners require node (https://nodejs.org/) for execution. Does VS Code require executing a task runner under a special node version?

A: We recommend that you use Node.js version 0.12.x. This is due to the fact that Node.js 0.10.x doesn't flush stdio on exit (see this issue (https://github.com/joyent/node/issues/8329) for details)

Last updated on 10/12/2015