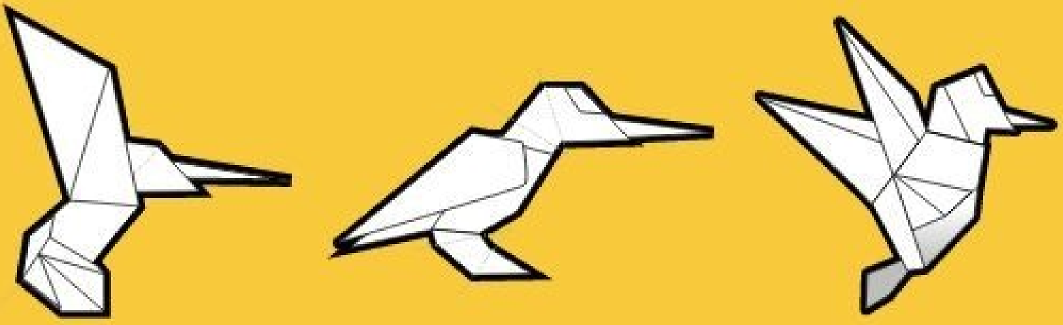
sitepoint

MODERN JAVASCRIPT TOOLS & SKILLS



10.

11.

12.

A COMPLETE ECOSYSTEM

Modern JavaScript Tools & Skills

Cover Design: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

Preface

There’s no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days. We're aiming to minimize that confusion with this set of books on modern JavaScript.

This book outlines essential tools and skills that every modern JavaScript developer should know.

Who Should Read This Book?

This book is for all front-end developers who wish to improve their JavaScript skills. You’ll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so: `<h1>A Perfect Summer's Day</h1> <p>It was a lovely day for a walk in the park. The birds were singing and the kids were all back at school.</p>`

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed: `function animate() { ⋮` **`new_variable = "Hello";`** `}`

Some lines of code should be entered on one line, but we’ve had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:
`URL.open("http://www.sitepoint.com/responsive-web- ➥design-real-user-testing/?responsive1");`

You’ll notice that we’ve used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: A Beginner’s Guide to Babel

by James Kolce

This article introduces Babel, a JavaScript compiler that allows developers to use next-generation JavaScript today.

It can be frustrating to write JavaScript when building web applications. We have to think about the features available in the browsers we’re targeting and what happens when a feature isn’t implemented. Some people would recommend simply not using it, which is a painful experience most of the time if we’re building something complicated.

Thankfully, some tools allow us to stop worrying about what’s supported and just write the best code we can. They’re called transpilers. A **transpiler** is a tool that takes source code as input and produces new source code as output, with a different syntax but semantically as close as possible — or ideally equivalent — to the original.

Babel is pretty much the standard transpiler to translate modern JavaScript (ES2015+) into compatible implementations that run in old browsers. It’s the perfect solution if you just want to concentrate on writing JavaScript.

And although the main goal of Babel is to translate the latest standards of ECMAScript (ES) for old — or sometimes current — browsers, it can do more. There’s an ecosystem of presets and plugins that make possible the addition of non-standard features as well. Each plugin makes a new feature/transformation available for your code, and presets are just a collection of plugins.

Getting Started

There are different ways to set up Babel depending on your project and the tools you use. In this article, we're going to explain how to set up Babel using the CLI, although if you're using a build system or framework, you can check out specific instructions on the official site. Most of the time the CLI is the fastest and easiest way to get started, so if you're a first-time user, feel free to continue.

The first step to set up Babel in a project is to install the package using npm and add it as a dev dependency. Assuming you have a working Node.js environment already in place, it's just a matter of running the following in your terminal:

```
mkdir babel-test
cd babel-test
npm init -y
npm install --save-dev babel-cli
```

This will create a directory (`babel-test`) change into the directory, initialize an npm project (thus creating a `package.json` file) and then install the babel-cli as a dev dependency.

If you need any help with the above, please consult our tutorials on installing Node and working with npm.

Next, we can open `package.json` and add a `build` command to our npm scripts:

```
"scripts": {
  "build": "babel src -d dist"
}
```

This will take the source files from the `src` directory and output the result in a `dist` directory. Then we can execute it as:

```
npm run build
```

But wait! Before running Babel we must install and set up the plugins that will transform our code. The easiest and quickest way to do this is to add the Env preset, which selects the appropriate plugins depending on the target browsers that you indicate. It can be installed using:

```
npm install babel-preset-env --save-dev
```

Then create a `.babelrc` file in the root of your project and add the preset:

```
{
  "presets": ["env"]
}
```

The `.babelrc` file is the place where you put all your settings for Babel. You'll be using this primarily for setting up presets and plugins, but a lot more options are available. You can check the complete list in the Babel API page.

Please note that, depending on your operating system, files beginning with a dot will be hidden by default. If this is problematic for you (or if you just prefer fewer files), you can put your Babel settings in the `package.json` file, under a `babel` key, like so:

```
{
  "name": "babel-test",
  "version": "1.0.0",
  "babel": {
    // config
  }
}
```

Finally, let's create the directories and files Babel is expecting to find:

```
mkdir src dist
```

And give it something to transform:

```
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(a);
console.log(b);
```

This example uses destructuring assignment to swap the values of two variables.

Running Babel

Now that you have a ready-to-use Babel installation, you can execute the `build` command to run the compilation process:

```
npm run build
```

This will take the code from `src/main.js`, transform it to ES5 code and output the transformed code to `dist/main.js`.

Here's what it produced:

```
"use strict";

var a = 1;
var b = 2;
var ref = [b, a];
a = ref[0];
b = _ref[1];

console.log(a);
console.log(b);
```

As you can see, `let` has been replaced by `var` and Babel has introduced a temporary variable (denoted by the underscore) to facilitate the swap.

And that's it. The code that you write in the `src` directory will be translated to previous versions of the language. By default, if you don't add any options to the preset, it will load all the transformations. You can also indicate the target browsers as follows:

```
{
  "presets": [
    ["env", {
      "targets": {
        "browsers": ["last 2 versions", "safari >= 7"]
      }
    }]
  ]
}
```

This will load the required transformations to support the latest two versions of each browser and Safari greater or equal to version 7. You can find the available options for the target browsers in the Browserlist repository.

Babel Ecosystem: A Quick Overview

As you noticed in the previous section, Babel won't do anything by itself when you install it. We have to install a set of plugins to obtain the desired behavior, or we can use presets, which are predefined sets of plugins.

Usually, each feature that you want to include will be in the form of a plugin. Some examples for ES2015 include:

- constants
- arrow functions
- block-scoped functions
- classes
- for-of
- spread
- template literals

See the Plugins page in the Babel Docs for a complete list.

But sometimes you don't want to include all the plugins one by one. So there are prebuilt presets that will facilitate the process of installing each plugin.

The three official presets currently available are:

- Env
- React
- Flow

Env is the most frequently used and the one we've used here. It automatically loads all the necessary transformations to make your code compatible depending on the targeted browsers.

The **React** preset transforms code typically found in React projects, mainly adding compatibility with Flow annotations and JSX.

And finally, the **Flow** preset is used to clean up the code from Flow type annotations (although it doesn't check whether the types are valid or not.)

Babel Polyfill

There are JavaScript features that can’t be transformed syntactically, usually because there’s no equivalent functionality implemented — for example, Promises and generator functions.

Those kinds of features have to be implemented in the browser by a library to be used in your code, and that’s the work of a polyfill.

The Babel polyfill is composed by [core-js]() and the [Regenerator]() runtime. Together, they cover all the features in ES2015+.

Advanced Use

As mentioned, Babel can also be used to transform features that haven’t yet been implemented in the language. A good example of this is the class fields proposal (currently at [TC39 stage 3: candidate]()). This is particularly popular among React devs, as it removes the necessity to explicitly bind methods to a particular component, and also means that a component’s `state` can be declared as a class field (potentially eliminating the need for a constructor).

For those of you wanting to use class fields today, you would need to add the [babel-plugin-transform-class-properties]() as a dev dependency: `npm install --save-dev babel-plugin-transform-class-properties`

You’d also update your `.babelrc` file as follows:

```
{
  "presets": ["env"],
  "plugins": ["transform-class-properties"]
}
```

Now you can write:

```
class App extends Component {
  state = { count: 0 };

  incCount = () => {
    this.setState(ps => ({ count: ps.count + 1 }));
  };

  render() {
    return (
      <div>
        <p>{ this.state.count }</p>
        <button onClick={this.incCount}>add one</button>
      </div>
    );
  }
}
```

And it doesn’t stop there. You can also use Babel to add new features of your own to the language, as our tutorial Understanding ASTs by Building Your Own Babel Plugin demonstrates.

Alternatives

Writing modern web applications sometimes requires more than the features available in JavaScript. Other languages can also be translated to compatible JavaScript but also implement other useful features.

The most popular option is TypeScript, which is regular JavaScript that implements modern ES features but also adds others, especially regarding type safety.

On the other extreme, there are entirely different languages across different categories, from the functional ones like PureScript to the object-oriented like Dart.

For a deeper overview of alternative languages, take a look at 10 Languages that Compile to JavaScript.

Conclusion

Babel is a great option for writing modern applications while still serving up JavaScript that can be understood by all developers and the wide range of browsers the code needs to run in.

Babel is not only useful for transforming ES2015+ to previous versions of the language — both in the browser and on platforms such as Node.js — but also for adding new features that aren't part of the standard. To see what I mean, just take a look at the npm directory to find all the available Babel plugins or presets.

As JavaScript is evolving at such a rapid pace, it's obvious that browser manufacturers will need a while to implement the latest features. Giving Babel a place in your toolkit means that you can write cutting-edge JavaScript today, safe in the knowledge that you're not abandoning any of your users. What's not to love?

Chapter 2: A Beginner’s Guide to Webpack 4 and Module Bundling

by Mark Brown

The Webpack 4 docs state that:

> Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset.

Webpack has become one of the most important tools for modern web development. It’s primarily a module bundler for your JavaScript, but it can be taught to transform all of your front-end assets like HTML, CSS, even images. It can give you more control over the number of HTTP requests your app is making and allows you to use other flavors of those assets (Pug, Sass, and ES8, for example). Webpack also allows you to easily consume packages from npm.

This article is aimed at those who are new to Webpack, and will cover initial setup and configuration, modules, loaders, plugins, code splitting and hot module replacement. If you find video tutorials helpful, I can highly recommend Glen Maddern’s Webpack from First Principles as a starting point to understand what it is that makes Webpack special. It’s a little old now, but the principles are still the same, and it’s a great introduction.

Example Code

To follow along at home, you’ll need to have Node.js installed. You can also download the demo app from GitHub.

Setup

Let’s initialize a new project with npm and install `webpack` and `webpack-cli`:

```
mkdir webpack-demo && cd webpack-demo
npm init -y
npm install --save-dev webpack webpack-cli
```

Next we’ll create the following directory structure and contents:

```
  webpack-demo
  |- package.json
+ |- webpack.config.js
+ |- src
+   |- index.js
+ |- dist
+   |- index.html
```

dist/index.html

```
<!doctype html>
<html>
  <head>
    <title>Hello Webpack</title>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

src/index.js

```
const root = document.createElement("div")
root.innerHTML = `<p>Hello Webpack.</p>`
document.body.appendChild(root)
```

webpack.config.js

```
const path = require('path')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
}
```

This tells Webpack to compile the code in our entry point `src/index.js` and output a bundle in *dist*`bundle.js`. Let’s add an npm script for running Webpack.

package.json

```
  {
    ...
    "scripts": {
-     "test": "echo \"Error: no test specified\" && exit 1"
+     "develop": "webpack --mode development --watch",
+     "build": "webpack --mode production"
    },
    ...
  }
```

Using the `npm run develop` command, we can create our first bundle!

```
Asset      Size      Chunks            Chunk Names
bundle.js  2.92 KiB  main  [emitted]  main
```

You should now be able to load `dist/index.html` in your browser and be greeted with “Hello Webpack”.

Open up `dist/bundle.js` to see what Webpack has done. At the top is Webpack’s module bootstrapping code, and right at the bottom is our module. You may not be colored impressed just yet, but if you’ve come this far you can now start using ES Modules, and Webpack will be able to produce a bundle for production that will work in all browsers.

Restart the build with `Ctrl` + `C` and run `npm run build` to compile our bundle in *production mode*.

```
Asset      Size       Chunks            Chunk Names
bundle.js  647 bytes  main  [emitted]  main
```

Notice that the bundle size has come down from **2.92 KiB** to **647 bytes**.

Take another look at `dist/bundle.js` and you’ll see an ugly mess of code. Our bundle has been minified with UglifyJS: the code will run exactly the same, but it’s done with the smallest file size possible.

- `--mode development` optimizes for build speed and debugging
- `--mode production` optimizes for execution speed at runtime and output file size.

Modules

Using ES Modules, you can split up your large programs into many small, self-contained programs.

Out of the box, Webpack knows how to consume ES Modules using `import` and `export` statements. As an example, let's try this out now by installing lodash-es and adding a second module:

```
npm install --save-dev lodash-es
```

src/index.js

```
import { groupBy } from "lodash-es"
import people from "./people"

const managerGroups = groupBy(people, "manager")

const root = document.createElement("div")
root.innerHTML = `<pre>${JSON.stringify(managerGroups, null, 2)}
</pre>`
document.body.appendChild(root)
```

src/people.js

```
const people = [
  {
    manager: "Jen",
    name: "Bob"
  },
  {
    manager: "Jen",
    name: "Sue"
  },
  {
    manager: "Bob",
    name: "Shirley"
  }
]

export default people
```

Run `npm run develop` to start Webpack and refresh `index.html`. You should see an array of people grouped by manager printed to the screen.

Imports Without a Relative Patch

Imports without a relative path like `'es-lodash'` are modules from npm installed to `/node_modules`. Your own modules will always need a relative path like `'./people'`, as this is how you can tell them apart.

Notice in the console that our bundle size has increased to **1.41 MiB**! This is worth keeping an eye on, though in this case there’s no cause for concern. Using `npm run build` to compile in *production* mode, all of the unused lodash modules from *lodash-es* are removed from bundle. This process of removing unused imports is known as **tree-shaking**, and is something you get for free with Webpack.

```
> npm run develop

Asset      Size      Chunks                  Chunk Names
bundle.js  1.41 MiB  main  [emitted]  [big]  main
```

```
> npm run build

Asset      Size      Chunks          Chunk Names
bundle.js  16.7 KiB  0  [emitted]  main
```

Loaders

Loaders let you run preprocessors on files as they're imported. This allows you to bundle static resources *beyond JavaScript*, but let's look at what can be done when loading `.js` modules first.

Let's keep our code modern by running all `.js` files through the next-generation JavaScript transpiler Babel:

```
npm install --save-dev "babel-loader@^8.0.0-beta" @babel/core
@babel/preset-env
```

webpack.config.js

```
  const path = require('path')

  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
+   module: {
+     rules: [
+       {
+         test: \.js$,
+         exclude: (node_modules|bower_components),
+         use: {
+           loader: 'babel-loader',
+         }
+       }
+     ]
+   }
  }
```

.babelrc

```
{
  "presets": [
    ["@babel/env", {
      "modules": false
    }]
  ],
  "plugins": ["syntax-dynamic-import"]
}
```

This config prevents Babel from transpiling `import` and `export` statements into ES5, and enables dynamic imports — which we'll look at later in the section on Code Splitting.

We're now free to use modern language features, and they'll be compiled down to ES5 that runs in all browsers.

Sass

Loaders can be chained together into a series of transforms. A good way to demonstrate how this works is by importing Sass from our JavaScript:

```
npm install --save-dev style-loader css-loader sass-loader node-
sass
```

webpack.config.js

```
  module.exports = {
    ...
    module: {
      rules: [
        ...
+       {
+         test: \.scss$,
+         use: [{
+           loader: 'style-loader'
+         }, {
+           loader: 'css-loader'
+         }, {
+           loader: 'sass-loader'
+         }]
+       }
      ]
    }
  }
```

These loaders are processed in reverse order:

- `sass-loader` transforms Sass into CSS.
- `css-loader` parses the CSS into JavaScript and resolves any dependencies.

- `style-loader` outputs our CSS into a `<style>` tag in the document.

You can think of these as function calls. The output of one loader feeds as input into the next:

```
styleLoader(cssLoader(sassLoader("source")))
```

Let’s add a Sass source file and import is a module.

src/style.scss

```
$bluegrey: #2b3a42;

pre {
  padding: 8px 16px;
  background: $bluegrey;
  color: #e1e6e9;
  font-family: Menlo, Courier, monospace;
  font-size: 13px;
  line-height: 1.5;
  text-shadow: 0 1px 0 rgba(23, 31, 35, 0.5);
  border-radius: 3px;
}
```

src/index.js

```
  import { groupBy } from 'lodash-es'
  import people from './people'

+ import './style.scss'

  ...
```

Restart the build with `Ctrl` + `C` and `npm run develop`. Refresh `index.html` in the browser and you should see some styling.

CSS in JS

We just imported a Sass file from our JavaScript, as a module.

Open up `dist/bundle.js` and search for “pre {”. Indeed, our Sass has been compiled to a string of CSS and saved as a module within our bundle. When we import this module into our JavaScript, `style-loader` outputs that string into an embedded `<style>` tag.

Why would you do such a thing?

I won’t delve too far into this topic here, but here are a few reasons to consider:

- A JavaScript component you may want to include in your project may *depend* on other assets to function properly (HTML, CSS, Images, SVG). If these can all be bundled together, it’s far easier to import and use.
- Dead code elimination: When a JS component is no longer imported by your code, the CSS will no longer be imported either. The bundle produced will only ever contain code that does something.
- CSS Modules: The global namespace of CSS makes it very difficult to be confident that a change to your CSS will not have any side effects. CSS modules change this by making CSS local by default and exposing unique class names that you can reference in your JavaScript.
- Bring down the number of HTTP requests by bundling/splitting code in clever ways.

Images

The last example of loaders we’ll look at is the handling of images with `file-loader`.

In a standard HTML document, images are fetched when the browser encounters an `img` tag or an element with a `background-image` property. With Webpack, you can optimize this in the case of small images by storing the source of the images as strings inside your JavaScript. By doing this, you preload them and the browser won’t have to fetch them with separate requests later:

```
npm install --save-dev file-loader
```

webpack.config.js

```
  module.exports = {
    ...
    module: {
      rules: [
        ...
+       {
+         test: \.(png|svg|jpg|gif)$,
+         use: [
+           {
+             loader: 'file-loader'
+           }
+         ]
+       }
      ]
    }
  }
```

Download a test image with this command:

```
curl https://raw.githubusercontent.com/sitepoint-editors/webpack-
demo/master/src/code.png --output src/code.png
```

Restart the build with `Ctrl` + `C` and `npm run develop` and you’ll now be able to import images as modules!

src/index.js

```
  import { groupBy } from 'lodash-es'
  import people from './people'

  import './style.scss'
+ import './image-example'

  ...
```

src/image-example.js

```
import codeURL from "./code.png"

const img = document.createElement("img")
img.src = codeURL
img.style = "background: #2B3A42; padding: 20px"
img.width = 32
document.body.appendChild(img)
```

This will include an image where the `src` attribute contains a data URI of the image itself:

```
<img src="data:image/png;base64,iVBO..." style="background:
#2B3A42; padding: 20px" width="32">
```

Background images in our CSS are also processed by `file-loader`.

src/style.scss

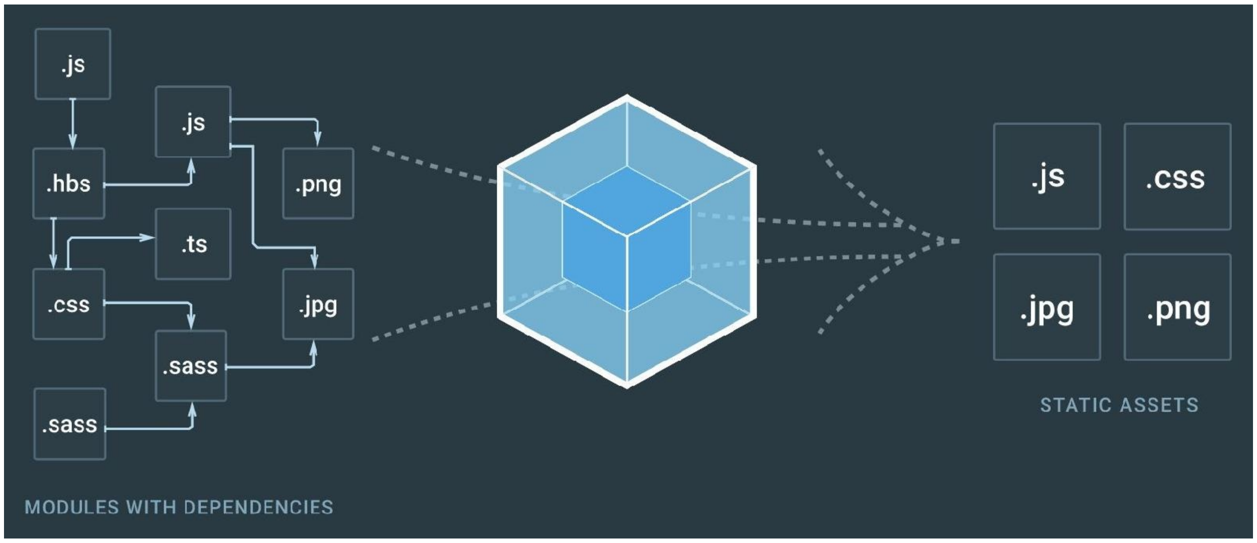
```
  $bluegrey: #2b3a42;

  pre {
    padding: 8px 16px;
-   background: $bluegrey;
+   background: $bluegrey url("code.png") no-repeat center center /
32px 32px;
    color: #e1e6e9;
    font-family: Menlo, Courier, monospace;
    font-size: 13px;
    line-height: 1.5;
    text-shadow: 0 1px 0 rgba(23, 31, 35, 0.5);
    border-radius: 3px;
  }
```

See more examples of Loaders in the docs:

- Loading Fonts
- Loading Data

Dependency Graph

You should now be able to see how loaders help to build up a tree of dependencies amongst your assets. This is what the image on the Webpack home page is demonstrating.



Though JavaScript is the entry point, Webpack appreciates that your other asset types — like HTML, CSS, and SVG — each have dependencies of their own, which should be considered as part of the build process.

Code Splitting

From the Webpack docs:

> Code splitting is one of the most compelling features of Webpack. This feature allows you to split your code into various bundles which can then be loaded on demand or in parallel. It can be used to achieve smaller bundles and control resource load prioritization which, if used correctly, can have a major impact on load time.

So far, we’ve only seen a single entry point — `src/index.js` — and a single output bundle — `dist/bundle.js`. When your app grows, you’ll need to split this up so that the entire codebase isn’t downloaded at the start. A good approach is to use Code Splitting and Lazy Loading to fetch things on demand as the code paths require them.

Let’s demonstrate this by adding a “chat” module, which is fetched and initialized when someone interacts with it. We’ll make a new entry point and give it a name, and we’ll also make the output’s filename dynamic so it’s different for each chunk.

webpack.config.js

```
   const path = require('path')

  module.exports = {
-   entry: './src/index.js',
+   entry: {
+     app: './src/app.js'
+   },
    output: {
-     filename: 'bundle.js',
+     filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
    ...
  }
```

src/app.js

```
import './app.scss'
```

```
const button = document.createElement("button")
button.textContent = 'Open chat'
document.body.appendChild(button)

button.onclick = () => {
  import(/* webpackChunkName: "chat" */ "./chat").then(chat => {
    chat.init()
  })
}
```

src/chat.js

```
import people from "./people"

export function init() {
  const root = document.createElement("div")
  root.innerHTML = `<p>There are ${people.length} people in the
room.</p>`
  document.body.appendChild(root)
}
```

src/app.scss

```
button {
  padding: 10px;
  background: #24b47e;
  border: 1px solid rgba(#000, .1);
  border-width: 1px 1px 3px;
  border-radius: 3px;
  font: inherit;
  color: #fff;
  cursor: pointer;
  text-shadow: 0 1px 0 rgba(#000, .3), 0 1px 1px rgba(#000, .2);
}
```

Not Webpack Specific Syntax

Note: Despite the `/* webpackChunkName */` comment for giving the bundle a name, this syntax is *not* Webpack specific. It’s the proposed syntax for dynamic imports intended to be supported directly in the browser.

Let’s run `npm run build` and see what this generates:

```
Asset             Size       Chunks         Chunk Names
chat.bundle.js    377 bytes  0   [emitted]  chat
app.bundle.js     7.65 KiB   1   [emitted]  app
```

As our entry bundle has changed, we’ll need to update our path to it as well.

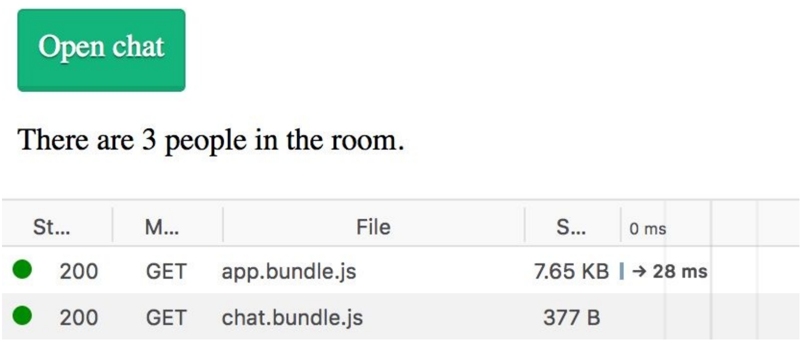
dist/index.html

```
  <!doctype html>
  <html>
    <head>
      <title>Hello Webpack</title>
    </head>
    <body>
-     <script src="bundle.js"></script>
+     <script src="app.bundle.js"></script>
    </body>
  </html>
```

Let’s start up a server from the dist directory to see this in action:

```
cd dist
npx serve
```

Open http://localhost:5000 in the browser and see what happens. Only `bundle.js` is fetched initially. When the button is clicked, the chat module is imported and initialized.



There are 3 people in the room.

St...	M...	File	S...	0 ms
● 200	GET	app.bundle.js	7.65 KB	→ 28 ms
● 200	GET	chat.bundle.js	377 B	

With very little effort, we’ve added dynamic code splitting and lazy loading of modules to our app. This is a great starting point for building a highly performant web app.

Plugins

While *loaders* operate transforms on single files, *plugins* operate across larger chunks of code.

Now that we're bundling our code, external modules *and* static assets, our bundle will grow — *quickly*. Plugins are here to help us split our code in clever ways and optimize things for production.

Without knowing it, we've actually already used many default Webpack plugins with "mode"

development

- Provides `process.env.NODE_ENV` with value "development"
- NamedModulesPlugin

production

- Provides `process.env.NODE_ENV` with value "production"
- UglifyJsPlugin
- ModuleConcatenationPlugin
- NoEmitOnErrorsPlugin

Production

Before adding additional plugins, we'll first split our config up so that we can apply plugins specific to each environment.

Rename `webpack.config.js` to `webpack.common.js` and add a config file for development and production.

```
- |- webpack.config.js
+ |- webpack.common.js
+ |- webpack.dev.js
+ |- webpack.prod.js
```

We'll use `webpack-merge` to combine our common config with the environment-specific config:

```
npm install --save-dev webpack-merge
```

webpack.dev.js

```
const merge = require('webpack-merge')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'development'
})
```

webpack.prod.js

```
const merge = require('webpack-merge')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'production'
})
```

package.json

```
   "scripts": {
-    "develop": "webpack --watch --mode development",
-    "build": "webpack --mode production"
+    "develop": "webpack --watch --config webpack.dev.js",
+    "build": "webpack --config webpack.prod.js"
   },
```

Now we can add plugins specific to development into `webpack.dev.js` and plugins specific to production in `webpack.prod.js`.

Split CSS

It's considered best practice to split your CSS from your JavaScript when bundling for production using ExtractTextWebpackPlugin.

The current `.scss` loaders are perfect for development, so we'll move those from `webpack.common.js` into `webpack.dev.js` and add `ExtractTextWebpackPlugin` to `webpack.prod.js` only.

```
npm install --save-dev extract-text-webpack-plugin@4.0.0-beta.0
```

webpack.common.js

```
 ...
 module.exports = {
  ...
  module: {
   rules: [
    ...
-   {
-    test: \.scss$,
-    use: [
-     {
-      loader: 'style-loader'
-     }, {
-      loader: 'css-loader'
-     }, {
-      loader: 'sass-loader'
-     }
-    ]
-   },
    ...
   ]
  }
 }
```

webpack.dev.js

```
 const merge = require('webpack-merge')
 const common = require('./webpack.common.js')

 module.exports = merge(common, {
  mode: 'development',
+ module: {
+  rules: [
+   {
+    test: \.scss$,
+    use: [
+     {
+      loader: 'style-loader'
+     }, {
+      loader: 'css-loader'
+     }, {
+      loader: 'sass-loader'
+     }
+    ]
+   }
+     ]
+   }
  })
```

webpack.prod.js

```
  const merge = require('webpack-merge')
+ const ExtractTextPlugin = require('extract-text-webpack-plugin')
  const common = require('./webpack.common.js')

  module.exports = merge(common, {
    mode: 'production',
+   module: {
+     rules: [
+       {
+         test: \.scss$,
+         use: ExtractTextPlugin.extract({
+           fallback: 'style-loader',
+           use: ['css-loader', 'sass-loader']
+         })
+       }
+     ]
+   },
+   plugins: [
+     new ExtractTextPlugin('style.css')
+   ]
  })
```

Let’s compare the output of our two build scripts:

```
> npm run develop

Asset            Size      Chunks            Chunk Names
app.bundle.js    28.5 KiB  app   [emitted]   app
chat.bundle.js   1.4 KiB   chat  [emitted]   chat
```

```
> npm run build

Asset            Size       Chunks         Chunk Names
chat.bundle.js   375 bytes  0  [emitted]   chat
app.bundle.js    1.82 KiB   1  [emitted]   app
style.css        424 bytes  1  [emitted]   app
```

Now that our CSS is extracted from our JavaScript bundle for production, we need to `<link>` to it from our HTML.

dist/index.html

```
  <!DOCTYPE html>
  <html>
    <head>
      <meta charset="UTF-8">
      <title>Code Splitting</title>
+     <link href="style.css" rel="stylesheet">
    </head>
    <body>
      <script type="text/javascript" src="app.bundle.js"></script>
    </body>
  </html>
```

This allows for parallel download of the CSS and JavaScript in the browser, so will be faster-loading than a single bundle. It also allows the styles to be displayed before the JavaScript finishes downloading.

Generating HTML

Whenever our outputs have changed, we've had to keep updating `index.html` to reference the new file paths. This is precisely what `html-webpack-plugin` was created to do for us automatically.

We may as well add `clean-webpack-plugin` at the same time to clear out our `/dist` directory before each build.

```
npm install --save-dev html-webpack-plugin clean-webpack-plugin
```

webpack.common.js

```
  const path = require('path')
+ const CleanWebpackPlugin = require('clean-webpack-plugin');
+ const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    ...
+   plugins: [
+     new CleanWebpackPlugin(['dist']),
+     new HtmlWebpackPlugin({
+       title: 'My killer app'
+     })
+   ]
  }
```

Now every time we build, dist will be cleared out. We’ll now see `index.html` output too, with the correct paths to our entry bundles.

Running `npm run develop` produces this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My killer app</title>
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script>
  </body>
</html>
```

And `npm run build` produces this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My killer app</title>
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script>
  </body>
</html>
```

Development

The webpack-dev-server provides you with a simple web server and gives you *live reloading*, so you don't need to manually refresh the page to see changes.

```
npm install --save-dev webpack-dev-server
```

package.json

```
  {
    ...
    "scripts": {
-     "develop": "webpack --watch --config webpack.dev.js",
+     "develop": "webpack-dev-server --config webpack.dev.js",
    }
    ...
  }
```

```
> npm run develop

ℹ ｢wds｣: Project is running at http://localhost:8080/
ℹ ｢wds｣: webpack output is served from /
```

Open up http://localhost:8080/ in the browser and make a change to one of the JavaScript or CSS files. You should see it build and refresh automatically.

HotModuleReplacement

The `HotModuleReplacement` plugin goes one step further than Live Reloading and swaps out modules at runtime *without the refresh*. When configured correctly, this saves a huge amount of time during development of single page apps. Where you have a lot of state in the page, you can make incremental changes to components, and only the changed modules are replaced and updated.

webpack.dev.js

```
+ const webpack = require('webpack')
  const merge = require('webpack-merge')
  const common = require('./webpack.common.js')

  module.exports = merge(common, {
    mode: 'development',
+   devServer: {
+     hot: true
+   },
+   plugins: [
+     new webpack.HotModuleReplacementPlugin()
+   ],
    ...
  }
```

Now we need to *accept* changed modules from our code to re-initialize things.

src/app.js

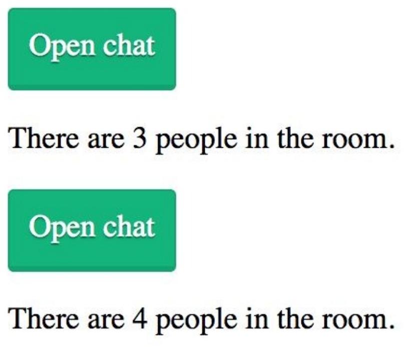
```
+ if (module.hot) {
+   module.hot.accept()
+ }

  ...
```

Hot Module Replacement

When Hot Module Replacement is enabled, `module.hot` is set to `true` for development and `false` for production, so these are stripped out of the bundle.

Restart the build and see what happens when we do the following:

- Click *Open chat*
- Add a new person to the `people.js` module
- Click *Open chat* again.



There are 3 people in the room.

There are 4 people in the room.

Here’s what’s happening:

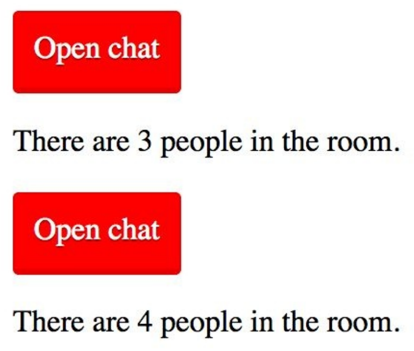
1. When *Open chat* is clicked, the `chat.js` module is fetched and initialized
2. HMR detects when `people.js` is modified
3. `module.hot.accept()` in `index.js` causes all modules loaded by this entry chunk to be replaced
4. When *Open chat* is clicked again, `chat.init()` is run with the code from the updated module.

CSS Replacement

Let’s change the button color to red and see what happens:

src/app.scss

```
  button {
    ...
-   background: #24b47e;
+   background: red;
    ...
  }
```



There are 3 people in the room.

There are 4 people in the room.

Now we get to see instant updates to our styles without losing any state. This is a much-improved developer experience! And it feels like the future.

HTTP/2

One of the primary benefits of using a module bundler like Webpack is that it can help you improve performance by giving you control over how the assets are *built* and then *fetched* on the client. It has been considered best practice for years to concatenate files to reduce the number of requests that need to be made on the client. This is still valid, but HTTP/2 now allows multiple files to be delivered in a *single* request, so concatenation isn’t a silver bullet anymore. Your app may actually benefit from having many small files individually cached. The client could then fetch a single changed module rather than having to fetch an entire bundle again with *mostly* the same contents.

The creator of Webpack, Tobias Koppers, has written an informative post explaining why bundling is still important, even in the HTTP/2 era.

Read more about this over at Webpack & HTTP/2.

Over to You

I hope you’ve found this introduction to Webpack helpful and are able to start using it to great effect. It can take a little time to wrap your head around Webpack’s configuration, loaders and plugins, but learning how this tool works will pay off.

The documentation for Webpack 4 is currently being worked on, but is really well put together. I highly recommend reading through the Concepts and Guides for more information. Here’s a few other topics you may be interested in:

- Source Maps for development
- Source Maps for production
- Cache busting with hashed filenames
- Splitting a vendor bundle

Chapter 3: An Introduction to Gulp.js

by Craig Buckler

Developers spend precious little time coding. Even if we ignore irritating meetings, much of the job involves basic tasks which can sap your working day:

- generating HTML from templates and content files
- compressing new and modified images
- compiling Sass to CSS code
- removing `console` and `debugger` statements from scripts
- transpiling ES6 to cross-browser–compatible ES5 code
- code linting and validation
- concatenating and minifying CSS and JavaScript files
- deploying files to development, staging and production servers.

Tasks must be repeated every time you make a change. You may start with good intentions, but the most infallible developer will forget to compress an image or two. Over time, pre-production tasks become increasingly arduous and time-consuming; you'll dread the inevitable content and template changes. It's mind-numbing, repetitive work. Wouldn’t it be better to spend your time on more profitable jobs?

If so, you need a *task runner* or *build process*.

That Sounds Scarily Complicated!

Creating a build process will take time. It's more complex than performing each task manually, but over the long term, you’ll save hours of effort, reduce human error and save your sanity. Adopt a pragmatic approach:

- Automate the most frustrating tasks first.
- Try not to over-complicate your build process. An hour or two is more than enough for the initial setup.
- Choose task runner software and stick with it for a while. Don't switch to another option on a whim.

Some of the tools and concepts may be new to you, but take a deep breath and concentrate on one thing at a time.

Task Runners: the Options

Build tools such as GNU Make have been available for decades, but web-specific task runners are a relatively new phenomenon. The first to achieve critical mass was Grunt — a Node.js task runner which used plugins controlled (originally) by a JSON configuration file. Grunt was hugely successful, but there were a number of issues:

1. Grunt required plugins for basic functionality such as file watching.
2. Grunt plugins often performed multiple tasks, which made customisation more awkward.
3. JSON configuration could become unwieldy for all but the most basic tasks.
4. Tasks could run slowly because Grunt saved files between every processing step.

Many issues were addressed in later editions, but Gulp had already arrived and offered a number of improvements:

1. Features such as file watching were built in.
2. Gulp plugins were *(mostly)* designed to do a single job.
3. Gulp used JavaScript configuration code that was less verbose, easier to read, simpler to modify, and provided better flexibility.
4. Gulp was faster because it uses Node.js streams to pass data through a series of piped plugins. Files were only written at the end of the task.

Of course, Gulp itself isn't perfect, and new task runners such as Broccoli.js, Brunch and webpack have also been competing for developer attention. More recently, npm itself has been touted as a simpler option. All have their pros and cons, but Gulp remains the favorite and is currently used by more than 40% of web developers.

Gulp requires Node.js, but while some JavaScript knowledge is beneficial, developers from all web programming faiths will find it useful.

What About Gulp 4?

This tutorial describes how to use Gulp 3 — the most recent release version at the time of writing. Gulp 4 has been in development for some time but remains a beta product. It's possible to use or switch to Gulp 4, but I recommend sticking with version 3 until the final release.

Step 1: Install Node.js

Node.js can be downloaded for Windows, macOS and Linux from nodejs.org/download/. There are various options for installing from binaries, package managers and docker images, and full instructions are available.

Windows Users

Node.js and Gulp run on Windows, but some plugins may not install or run if they depend on native Linux binaries such as image compression libraries. One option for Windows 10 users is the new bash commandline, which solves many issues.

Once installed, open a command prompt and enter:

```
node -v
```

This reveals the version number. You're about to make heavy use of `npm` — the Node.js package manager which is used to install modules. Examine its version number:

```
npm -v
```

For Linux Users

Node.js modules can be installed globally so they're available throughout your system. However, most users will not have permission to write to the global directories unless `npm` commands are prefixed with `sudo`. There are a number of options to fix npm permissions and tools such as nvm can help, but I often change the default directory. For example, on Ubuntu/Debian-based platforms:

```
cd ~
        mkdir .node_modules_global
        npm config set prefix=$HOME/.node_modules_global
        npm install npm -g
```

Then add the following line to the end of `~/.bashrc`:

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

Finally, update with this:

```
source ~/.bashrc
```

Step 2: Install Gulp Globally

Install Gulp command-line interface globally so the `gulp` command can be run from any project folder:

```
npm install gulp-cli -g
```

Verify Gulp has installed with this:

```
gulp -v
```

Step 3: Configure Your Project

For Node.js Projects

You can skip this step if you already have a `package.json` configuration file.

Presume you have a new or pre-existing project in the folder `project1`. Navigate to this folder and initialize it with npm: `cd project1 npm init`

You’ll be asked a series of questions. Enter a value or hit **Return** to accept defaults. A `package.json` file will be created on completion which stores your `npm` configuration settings.

For Git Users

Node.js installs modules to a `node_modules` folder. You should add this to your `.gitignore` file to ensure they’re not committed to your repository. When deploying the project to another PC, you can run `npm install` to restore them.

For the remainder of this article, we'll presume your project folder contains the following sub-folders: **`src` folder: preprocessed source files**

This contains further sub-folders:

- `html` - HTML source files and templates
- `images` — the original uncompressed images
- `js` — multiple preprocessed script files
- `scss` — multiple preprocessed Sass `.scss` files

`build` folder: compiled/processed files

Gulp will create files and create sub-folders as necessary:

- `html` — compiled static HTML files
- `images` — compressed images
- `js` — a single concatenated and minified JavaScript file
- `css` — a single compiled and minified CSS file

Your project will almost certainly be different but this structure is used for the examples below.

Following Along on Unix

If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,css}
```

Step 4: Install Gulp Locally

You can now install Gulp in your project folder using the command:

```
npm install gulp --save-dev
```

This installs Gulp as a development dependency and the `"devDependencies"` section of `package.json` is updated accordingly. We'll presume Gulp and all plugins are development dependencies for the remainder of this tutorial.

Alternative Deployment Options

Development dependencies are not installed when the `NODE_ENV` environment variable is set to `production` on your operating system. You would normally do this on your live server with the Mac/Linux command:

```
export NODE_ENV=production
```

Or on Windows:

```
set NODE_ENV=production
```

This tutorial presumes your assets will be compiled to the `build` folder and committed to your Git repository or uploaded directly to the server. However, it may be preferable to build assets on the live server if you want to change the way they are created. For example, HTML, CSS and JavaScript files are minified on production but not development environments. In that case, use the `--save` option for Gulp and all plugins, *i.e.*

```
npm install gulp --save
```

This sets Gulp as an application dependency in the `"dependencies"` section of `package.json`. It will be installed when you enter `npm install` and can be run wherever the project is deployed. You can remove the `build` folder from your repository since the files can be created on any platform when required.

Step 4: Create a Gulp Configuration File

Create a new `gulpfile.js` configuration file in the root of your project folder. Add some basic code to get started: `// Gulp.js configuration var // modules gulp = require('gulp'), // development mode? devBuild = (process.env.NODE_ENV !== 'production'), // folders folder = { src: 'src/', build: 'build/' } ;`

This references the Gulp module, sets a `devBuild` variable to `true` when running in development (or non-production mode) and defines the source and build folder locations.

ES6

ES6 note: ES5-compatible JavaScript code is provided in this tutorial. This will work for all versions of Gulp and Node.js with or without the `--harmony` flag. Most ES6 features are supported in Node 6 and above so feel free to use arrow functions, `let`, `const`, *etc.* if you're using a recent version.

`gulpfile.js` won't do anything yet because you need to …

Step 5: Create Gulp Tasks

On its own, Gulp does nothing. You must:

1. install Gulp plugins, and
2. write tasks which utilize those plugins to do something useful.

It's possible to write your own plugins but, since almost 3,000 are available, it's unlikely you'll ever need to. You can search using Gulp's own directory at gulpjs.com/plugins/, on npmjs.com, or search "gulp *something*" to harness the mighty power of Google.

Gulp provides three primary task methods:

- `gulp.task` — defines a new task with a name, optional array of dependencies and a function.
- `gulp.src` — sets the folder where source files are located.
- `gulp.dest` — sets the destination folder where build files will be placed.

Any number of plugin calls are set with `pipe` between the `.src` and `.dest`.

Image Task

This is best demonstrated with an example, so let's create a basic task which compresses images and copies them to the appropriate `build` folder. Since this process could take time, we'll only compress new and modified files. Two plugins can help us: gulp-newer and gulp-imagemin. Install them from the command-line:

```
npm install gulp-newer gulp-imagemin --save-dev
```

We can now reference both modules the top of `gulpfile.js`:

```
// Gulp.js configuration

var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
```

We can now define the image processing task itself as a function at the end of `gulpfile.js`:

```
// image processing
gulp.task('images', function() {
  var out = folder.build + 'images/';
  return gulp.src(folder.src + 'images/**/*')
    .pipe(newer(out))
    .pipe(imagemin({ optimizationLevel: 5 }))
    .pipe(gulp.dest(out));
});
```

All tasks are syntactically similar. This code:

1. Creates a new task named `images`.
2. Defines a function with a return value which …
3. Defines an `out` folder where build files will be located.
4. Sets the Gulp `src` source folder. The `/**/*` ensures that images in sub-folders are also processed.
5. Pipes all files to the `gulp-newer` module. Source files that are newer than corresponding destination files are passed through. Everything else is removed.
6. The remaining new or changed files are piped through `gulp-imagemin` which sets an optional `optimizationLevel` argument.
7. The compressed images are output to the Gulp `dest` folder set by `out`.

Save `gulpfile.js` and place a few images in your project's `src/images` folder before running the task from the command line:

```
gulp images
```

All images are compressed accordingly and you'll see output such as:

```
Using file gulpfile.js
Running 'imagemin'...
Finished 'imagemin' in 5.71 ms
gulp-imagemin: image1.png (saved 48.7 kB)
gulp-imagemin: image2.jpg (saved 36.2 kB)
gulp-imagemin: image3.svg (saved 12.8 kB)
```

Try running `gulp images` again and nothing should happen because no newer images exist.

HTML Task

We can now create a similar task which copies files from the source HTML folder. We can safely minify our HTML code to remove unnecessary whitespace and attributes using the gulp-htmlclean plugin:

```
npm install gulp-htmlclean --save-dev
```

This is then referenced at the top of `gulpfile.js`:

```
var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
  htmlclean = require('gulp-htmlclean'),
```

We can now create an `html` task at the end of `gulpfile.js`:

```
// HTML processing
gulp.task('html', ['images'], function() {
  var
    out = folder.build + 'html/',
    page = gulp.src(folder.src + 'html/**/*')
      .pipe(newer(out));

  // minify production code
  if (!devBuild) {
    page = page.pipe(htmlclean());
  }

  return page.pipe(gulp.dest(out));
});
```

This reuses `gulp-newer` and introduces a couple of concepts:

1. The `[images]` argument states that our `images` task must be run before processing the HTML (the HTML is likely to reference images). Any number of dependent tasks can be listed in this array and all will complete before the task function runs.
2. We only pipe the HTML through `gulp-htmlclean` if `NODE_ENV` is set to

`production`. Therefore, the HTML remains uncompressed during development which may be useful for debugging.

Save `gulpfile.js` and run `gulp html` from the command line. Both the `html` and `images` tasks will run.

JavaScript Task

Too easy for you? Let's process all our JavaScript files by building a basic module bundler. It will:

1. Ensure dependencies are loaded first using the gulp-deporder plugin. This analyses comments at the top of each script to ensure correct ordering. For example, `// requires: defaults.js lib.js`.
2. Concatenate all script files into a single `main.js` file using gulp-concat.
3. Remove all `console` and `debugging` statements with gulp-strip-debug and minimize code with gulp-uglify. This step will only occur when running in production mode.

Install the plugins:

```
npm install gulp-deporder gulp-concat gulp-strip-debug gulp-uglify
--save-dev
```

Reference them at the top of `gulpfile.js`:

```
var
  ...
  concat = require('gulp-concat'),
  deporder = require('gulp-deporder'),
  stripdebug = require('gulp-strip-debug'),
  uglify = require('gulp-uglify'),
```

Then add a new `js` task:

```
// JavaScript processing
gulp.task('js', function() {

  var jsbuild = gulp.src(folder.src + 'js/**/*')
    .pipe(deporder())
    .pipe(concat('main.js'));
  if (!devBuild) {
    jsbuild = jsbuild
      .pipe(stripdebug())
      .pipe(uglify());
  }

  return jsbuild.pipe(gulp.dest(folder.build + 'js/'));

});
```

Save then run `gulp js` to watch the magic happen!

CSS Task

Finally, let’s create a CSS task which compiles Sass `.scss` files to a single `.css` file using gulp-sass. This is a Gulp plugin for node-sass which binds to the superfast LibSass C/C++ port of the Sass engine *(you won't need to install Ruby)*. We'll presume your primary Sass file `scss/main.scss` is responsible for loading all partials.

Our task will also utilize the fabulous PostCSS via the gulp-postcss plugin. PostCSS requires its own set of plugins and we'll install these:

- postcss-assets to manage assets. This allows us to use properties such as `background: resolve('image.png');` to resolve file paths or `background: inline('image.png');` to inline data-encoded images.
- autoprefixer to automatically add vendor prefixes to CSS properties.
- css-mqpacker to pack multiple references to the same CSS media query into a single rule.
- cssnano to minify the CSS code when running in production mode.

First, install all the modules:

```
npm install gulp-sass gulp-postcss postcss-assets autoprefixer css-
mqpacker cssnano --save-dev
```

Then reference them at the top of `gulpfile.js`:

```
var
  ...
  sass = require('gulp-sass'),
  postcss = require('gulp-postcss'),
  assets = require('postcss-assets'),
  autoprefixer = require('autoprefixer'),
  mqpacker = require('css-mqpacker'),
  cssnano = require('cssnano'),
```

We can now create a new `css` task at the end of `gulpfile.js`. Note the `images` task is set as a dependency because the `postcss-assets` plugin can reference images during the build process. In addition, most plugins can be passed arguments (refer to their documentation for more information):

```
// CSS processing
gulp.task('css', ['images'], function() {

  var postCssOpts = [
  assets({ loadPaths: ['images/'] }),
  autoprefixer({ browsers: ['last 2 versions', '> 2%'] }),
  mqpacker
  ];

  if (!devBuild) {
    postCssOpts.push(cssnano);
  }

  return gulp.src(folder.src + 'scss/main.scss')
    .pipe(sass({
      outputStyle: 'nested',
      imagePath: 'images/',
      precision: 3,
      errLogToConsole: true
    }))
    .pipe(postcss(postCssOpts))
    .pipe(gulp.dest(folder.build + 'css/'));

});
```

Save the file and run the task from the command line:

```
gulp css
```

Step 6: Automate Tasks

We've been running one task at a time. We can run them all in one command by adding a new `run` task to `gulpfile.js`:

```
// run all tasks
gulp.task('run', ['html', 'css', 'js']);
```

Save and enter `gulp run` at the command line to execute all tasks. Note that I omitted the `images` task because it's already set as a dependency for the `html` and `css` tasks.

Is this still too much hard work? Gulp offers another method — `gulp.watch` — which can monitor your source files and run an appropriate task whenever a file is changed. The method is passed a folder and a list of tasks to execute when a change occurs. Let's create a new `watch` task at the end of `gulpfile.js`:

```
// watch for changes
gulp.task('watch', function() {

  // image changes
  gulp.watch(folder.src + 'images/**/*', ['images']);

  // html changes
  gulp.watch(folder.src + 'html/**/*', ['html']);

  // javascript changes
  gulp.watch(folder.src + 'js/**/*', ['js']);

  // css changes
  gulp.watch(folder.src + 'scss/**/*', ['css']);

});
```

Rather than running `gulp watch` immediately, let's add a default task:

```
// default task
gulp.task('default', ['run', 'watch']);
```

Save `gulpfile.js` and enter `gulp` at the command line. Your images, HTML, CSS and JavaScript will all be processed, then Gulp will remain active watching for updates and re-running tasks as necessary. Hit `Ctrl/Cmd` + `C` to abort monitoring and return to the command line.

Step 7: Profit!

Other plugins you may find useful:

- gulp-load-plugins ± load all Gulp plugin modules without `require` declarations
- gulp-preprocess — a simple HTML and JavaScript preprocessor
- gulp-less — the Less CSS preprocessor plugin
- gulp-stylus — the Stylus CSS preprocessor plugin
- gulp-sequence — run a series of gulp tasks in a specific order
- gulp-plumber — error handling which prevents Gulp stopping on failures
- gulp-size — displays file sizes and savings
- gulp-nodemon — uses nodemon to automatically restart Node.js applications when changes occur.
- gulp-util — utility functions including logging and color coding.

One useful method in `gulp-util` is `.noop()` which passes data straight through without performing any action. This could be used for cleaner development/production processing code. For example:

```
var gutil = require('gulp-util');



// HTML processing


gulp.task('html', ['images'], function() {
```

```
var out = folder.src + 'html/**/*';



return gulp.src(folder.src + 'html/**/*')

  .pipe(newer(out))

  .pipe(devBuild ? gutil.noop() : htmlclean())

  .pipe(gulp.dest(out));

                              });
```

Gulp can also call other Node.js modules, and they don't necessarily need to be plugins. For example:

- browser-sync — automatically reload assets or refresh your browser when changes occur
- del — delete files and folders (perhaps clean your `build` folder at the start of every run).

Invest a little time and Gulp could save many hours of development frustration. The advantages:

- plugins are plentiful

- configuration using pipes is readable and easy to follow
- `gulpfile.js` can be adapted and reused in other projects
- your total page weight can be reduced to improve performance
- you can simplify your deployment.

Useful links:

- Gulp home page
- Gulp plugins
- npm home page

Applying the processes above to a simple website reduced the total weight by more than 50%. You can test your own results using page weight analysis tools or a service such as New Relic, which provides a range of sophisticated application performance monitoring tools.

Gulp can revolutionize your workflow. I hope you found this tutorial useful and consider Gulp for your production process.

Chapter 4: 10 Languages That Compile to JavaScript

by James Kolce

This chapter includes a list of ten interesting languages that can compile to JavaScript to be executed in the browser or on a platform like Node.js.

Modern applications have different requirements from simple websites. But the browser is a platform with a (mostly) fixed set of technologies available, and JavaScript remains as the core language for web applications. Any application that needs to run in the browser has to be implemented in that language.

We all know that JavaScript isn’t the best language for every task, and when it comes to complex applications, it might fall short. To avoid this problem, several new languages and transpilers of existing ones have been created, all of them producing code that can work in the browser without any lines of JavaScript having to be written, and without you having to think about the limitations of the language.

Dart

Dart is a classical, object-oriented language where everything is an object and any object is an instance of a class (objects can act as functions too.) It's specially made to build applications for browsers, servers, and mobile devices. It's maintained by Google and is the language that powers the next generation AdWords UI, the most important product of Google regarding revenue, which is in itself a proof of its power at scale.

The language can be translated to JavaScript to be used in a browser, or be directly interpreted by the Dart VM, which allows you to build server applications too. Mobile applications can be made using the Flutter SDK.

Complex applications also require a mature set of libraries and language features specially designed for the task, and Dart includes all of this. An example of a popular library is AngularDart, a version of Angular for Dart.

It allows you to write type-safe code without being too intrusive. You can write types, but you aren't required to do so,* since they can be inferred. This allows for rapid prototyping without having to overthink the details, but once you have something working, you can add types to make it more robust.

Regarding concurrent programming in the VM, instead of shared-memory threads (Dart is single-threaded), Dart uses what they call **Isolates**, with their own memory heap, where communication is achieved using messages. In the browser, the story is a little different: instead of creating new isolates, you create new **Workers**.

```
// Example extracted from dartlang.org

import 'dart:async';
import 'dart:math' show Random;

main() async {
  print('Compute π using the Monte Carlo method.');
  await for (var estimate in computePi()) {
    print('π ≅ $estimate');
  }
}
```

```
/// Generates a stream of increasingly accurate estimates of π.
Stream<double> computePi({int batch: 1000000}) async* {
  var total = 0;
  var count = 0;
  while (true) {
    var points = generateRandom().take(batch);
    var inside = points.where((p) => p.isInsideUnitCircle);
    total += batch;
    count += inside.length;
    var ratio = count  total;
    / Area of a circle is A = π·r², therefore π = A/r².
    // So, when given random points with x ∈ <0,1>,
    // y ∈ <0,1>, the ratio of those inside a unit circle
    // should approach π  4. Therefore, the value of π
    / should be:
    yield ratio  4;
  }
}

Iterable<Point> generateRandom([int seed]) sync {
  final random = new Random(seed);
  while (true) {
    yield new Point(random.nextDouble(), random.nextDouble());
  }
}

class Point {
  final double x, y;
  const Point(this.x, this.y);
  bool get isInsideUnitCircle => x  x + y  y <= 1;
}
```

For more reading, I recommend Dart’s Get started with Dart resource.

TypeScript

TypeScript is a superset of JavaScript. A valid JavaScript program is also valid TypeScript, but with static typing added. The compiler can also work as a transpiler from ES2015+ to current implementations, so you always get the latest features.

Unlike many other languages, TypeScript keeps the spirit of JavaScript intact, only adding features to improve the soundness of the code. These are type annotations and other type-related functionality that makes writing JavaScript more pleasant, thanks to the enabling of specialized tools like static analyzers and other tools to aid in the refactoring process. Also, the addition of types improve the interfaces between the different components of your applications.

Type inference is supported, so you don’t have to write all the types from the beginning. You can write quick solutions, and then add all the types to get confident about your code.

TypeScript also has support for advanced types, like intersection types, union types, type aliases, discriminated unions and type guards. You can check out all these in the Advanced Types page in the TypeScript Documentation site.

JSX is also supported by adding the React typings if you use React:

```
class Person {
  private name: string;
  private age: number;
  private salary: number;

  constructor(name: string, age: number, salary: number) {
    this.name = name;
    this.age = age;
    this.salary = salary;
  }

  toString(): string {
    return `${this.name} (${this.age}) (${this.salary})`;
  }
}
```

For more on typeScript, check out SitePoint’s getting started with TypeScript article.

Elm

Elm is a purely functional programming language that can compile to JavaScript, HTML, and CSS. You can build a complete site with just Elm, making it a great alternative to JavaScript frameworks like React. The applications that you build with it automatically use a virtual DOM library, making it very fast. One big plus is the built-in architecture that makes you forget about data-flow and focus on data declaration and logic instead.

In Elm, all functions are pure, which means they'll always return the same output for a given input. They can't do anything else unless you specify it. For example, to access a remote API you'd create *command* functions to communicate with the external world, and *subscriptions* to listen for responses. Another point for purity is that values are immutable: when you need something, you create new values, instead of modifying them.

The adoption of Elm can be gradual. It's possible to communicate with JavaScript and other libraries using *ports*. Although Elm hasn't reached version 1 yet, it's being used for complex and large applications, making it a feasible solution for complex applications.

One of the most attractive features of Elm is the beginner-friendly compiler, which, instead of producing hard-to-read messages, generates code that helps you to fix your code. If you're learning the language, the compiler itself can be of big help.

```
module Main exposing (..)


import Html exposing (..)






-- MAIN

```

```
main : Program Never Model Msg

main =

    Html.program

        { init = init

        , update = update

        , view = view

        , subscriptions = subscriptions

                                 }
-- INIT



type alias Model = String



init : ( Model, Cmd Msg )
init = ( "Hello World!", Cmd.none )



-- UPDATE



type Msg

    = DoNothing



update : Msg -> Model -> ( Model, Cmd Msg )

update msg model =

    case msg of

        DoNothing ->

            ( model, Cmd.none )
```

```
-- VIEW



view : Model -> Html Msg

view model =

    div [] [text model]



-- SUBSCRIPTIONS



subscriptions : Model -> Sub Msg

subscriptions model =

    Sub.none
```

SitePoint has a handy getting started with Elm article if you want to find out more.

PureScript

PureScript is a purely functional and strongly typed programming language, created by Phil Freeman. It aims to provide strong compatibility with available JavaScript libraries, similar to Haskell in spirit, but keeping JavaScript at its core.

A strong point for PureScript is its minimalism. It doesn’t include any libraries for functionality that would be considered essential in other languages. For example, instead of including generators and promises in the compiler itself, you can use specific libraries for the task. You can choose the implementation you want for the feature you need, which allows a highly efficient and personalized experience when using PureScript, while keeping the generated code as small as possible.

Another distinctive feature of its compiler is the ability to make clean and readable code while maintaining compatibility with JavaScript, both concerning libraries and tools.

Like other languages, PureScript has its own build tool called Pulp, which can be compared to Gulp, but for projects written in this language.

Regarding the type system — unlike Elm, which is the other ML-like language — PureScript has support for advanced type features like higher-kinded types and type classes, which are taken from Haskell, allowing the creation of sophisticated abstractions:

```
module Main where

import Prelude
import Data.Foldable (fold)
import TryPureScript

main =
    render $ fold
      [ h1 (text "Try PureScript!")
      , p (text "Try out the examples below, or create your own!")
      , h2 (text "Examples")
      , list (map fromExample examples)
      ]
```

```
  where
    fromExample { title, gist } =
      link ("?gist=" <> gist) (text title)

    examples =
      [ { title: "Algebraic Data Types"
        , gist: "37c3c97f47a43f20c548"
        }
      , { title: "Loops"
        , gist: "cfdabdcd085d4ac3dc46"
        }
      , { title: "Operators"
        , gist: "3044550f29a7c5d3d0d0"
        }
      ]
```

To take your next step with PureScript, check out the getting started with PureScript guide on GitHub.

CoffeeScript

CoffeeScript is a language that aims to expose the good parts of JavaScript while providing a cleaner syntax and keeping the semantics in place. Although the popularity of the language has been waning in recent years, it's changing direction and recently received a new major version providing support for ES2015+ features.

The code you write in CoffeeScript is directly translated to readable JavaScript code and maintains compatibility with existing libraries. From version 2, the compiler produces code compatible with the latest versions of ECMAScript. For example, every time you use a `class`, you get a `class` in JavaScript. Also, if you use React, there's good news: JSX is compatible with CoffeeScript.

A very distinctive feature of the compiler is the ability to process code written in the literate style, where instead of making emphasis in the code and having comments as an extra, you write comments in the first place, and the code only occasionally appears. This style of programming was introduced by Donald Knuth, making a code file very similar to a technical article.

Unlike the other languages, CoffeeScript code can be interpreted directly in the browser using a library. So if you want to create a quick test, you can write your code in `text/coffeescript` script tags, and include the compiler, which will translate the code to JavaScript on the fly:

```
# Assignment:
number   = 42
opposite = true

# Conditions:
number = -42 if opposite

# Functions:
square = (x) -> x  x

# Arrays:
list = [1, 2, 3, 4, 5]

# Objects:
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x  square x

# Splats:
race = (winner, runners...) ->
  print winner, runners

# Existence:
alert "I knew it!" if elvis?

# Array comprehensions:
cubes = (math.cube num for num in list)
```

The CoffeeScript site has a handy getting started with CoffeeScript 2 resource.

ClojureScript

ClojureScript is a compiler that translates the Clojure programming language to JavaScript. It’s a general-purpose, functional language with dynamic typing and support for immutable data structures.

It’s the only one from this list that belongs to the Lisp family of programming languages and, naturally, it shares a lot of the features. For example, the code can be treated as data, and a macro system is available, making metaprogramming techniques possible. Unlike other Lisps, Clojure has support for immutable data structures, making the management of side effects easier.

The syntax can look intimidating for newcomers because of its use of parentheses, but it has profound reasons to be that way, and you’ll certainly appreciate it in the long run. That minimalism in the syntax and its syntactic abstraction capabilities make Lisp a powerful tool for solving problems that require high levels of abstraction.

Although Clojure is mainly a functional language, it isn’t pure like PureScript or Elm. Side effects can still happen, but other functional features are still present.

ClojureScript uses Google Closure for code optimization and also has compatibility with existing JavaScript libraries:

```
; Extracted from
https://github.com/clojure/clojurescript/blob/master/samples/dom/src


(ns dom.test
  (:require [clojure.browser.event :as event]
            [clojure.browser.dom   :as dom]))

(defn log [& args]
  (.log js/console (apply pr-str args)))

(defn log-obj [obj]
  (.log js/console obj))

(defn log-listener-count []
  (log "listener count: " (event/total-listener-count)))
```

```
(def source      (dom/get-element "source"))
(def destination (dom/get-element "destination"))

(dom/append source
            (dom/element "Testing me ")
            (dom/element "out!"))

(def success-count (atom 0))

(log-listener-count)

(event/listen source
              :click
              (fn [e]
                (let [i (swap! success-count inc)
                      e (dom/element :li
                                     {:id "testing"
                                     :class "test me out please"}
                                     "It worked!")]
                  (log-obj e)
                  (log i)
                  (dom/append destination
                              e))))

(log-obj (dom/element "Text node"))
(log-obj (dom/element :li))
(log-obj (dom/element :li {:class "foo"}))
(log-obj (dom/element :li {:class "bar"} "text node"))
(log-obj (dom/element [:ul [:li :li :li]]))
(log-obj (dom/element :ul [:li :li :li]))
(log-obj (dom/element :li {} [:ul {} [:li :li :li]]))
(log-obj (dom/element [:li {:class "baz"} [:li {:class "quux"}]]))

(log-obj source)
(log-listener-count)
```

To lean more, head over to the ClojureScript site’s getting started with ClojureScript resource.

Scala.js

Scala.js is a compiler that translates the Scala programming language to JavaScript. Scala is a language that aims to merge the ideas from object-oriented and functional programming into one language to create a powerful tool that is also easy to adopt.

As a strongly typed language, you get the benefits of a flexible type system with partial type inference. Most values can be inferred, but function parameters still require explicit type annotations.

Although many common object-oriented patterns are supported (for example, every value is an object and operations are method calls), you also get functional features like support for first-class functions and immutable data structures.

One of the special advantages of Scala.js is that you can start with a familiar, object-oriented approach and move to a more functional one as you need and at your own speed, without having to do a lot of work. Also, existing JavaScript code and libraries are compatible with your Scala code.

Beginner Scala developers will find the language not very different from JavaScript. Compare the following equivalent code:

```
// JavaScript
var xhr = new XMLHttpRequest();

xhr.open("GET",
  "https://api.twitter.com/1.1/search/" +
  "tweets.json?q=%23scalajs"
);
xhr.onload = (e) => {
  if (xhr.status === 200) {
    var r = JSON.parse(xhr.responseText);
    $("#tweets").html(parseTweets(r));
  }
};
xhr.send();
```

```
// Scala.js
val xhr = new XMLHttpRequest()
```

```
xhr.open("GET",
  "https://api.twitter.com/1.1/search/" +
  "tweets.json?q=%23scalajs"
)
xhr.onload = { (e: Event) =>
  if (xhr.status == 200) {
    val r = JSON.parse(xhr.responseText)
    $("#tweets").html(parseTweets(r))
  }
}
xhr.send()
```

Check out the Scala.js getting started with Scala.js docs for more.

Reason

Reason is a language created and maintained by Facebook, which offers a new syntax for the OCaml compiler, and the code can be translated to both JavaScript and native code.

Being part of the ML family and a functional language itself, it naturally offers a powerful but flexible type system with inference, algebraic data types and pattern matching. It also has support for immutable data types and parametric polymorphism (also known as *generics* in other languages) but, as in OCaml, support for object-oriented programming is available as well.

The use of existing JavaScript libraries is possible with bucklescript bindings. You can also mix in JavaScript alongside your Reason code. The inserted JavaScript code won't be strictly checked, but it works fine for quick fixes or prototypes.

If you're a React developer, bindings are available, and the language also has support for JSX:

```
/* A type variant being pattern matched

let possiblyNullValue1 = None;
let possiblyNullValue2 = Some "Hello@";

switch possiblyNullValue2 {
| None => print_endline "Nothing to see here."
| Some message => print_endline message
};

 Parametrized types

type universityStudent = {gpa: float};
type response 'studentType = {status: int, student: 'studentType};
let result: response universityStudent = fetchDataFromServer ();

 A simple typed object */

type payload = Js.t {.
  name: string,
  age: int
};
let obj1: payload = {"name": "John", "age": 30};
```

Check out the Reason site’s getting started with Reason guide for more.

Haxe

Haxe is a multi-paradigm programming language, and its compiler can produce both binaries and source code in other languages.

Although Haxe provides a strict type system with support for type inference, it can also work as a dynamic language if the target language supports it. In the same way, it provides support for a variety of programming styles like object-oriented, generic, and functional.

When you write Haxe code, you can target several platforms and languages for compilation without having to make considerable changes. Target-specific code blocks are also available.

You can write both back ends and front ends in Haxe with the same code and achieve communication using Haxe Remoting, for both synchronous and asynchronous connections.

As expected, Haxe code is compatible with existing libraries, but it also provides a mature standard library:

```
// Example extracted from http://code.haxe.org

extern class Database {
  function new();
  function getProperty<T>(property:Property<T>):T;
  function setProperty<T>(property:Property<T>, value:T):Void;
}

abstract Property<T>(String) {
  public inline function new(name) {
    this = name;
  }
}

class Main {
  static inline var PLAYER_NAME = new Property<String>
("playerName");
  static inline var PLAYER_LEVEL = new Property<Int>
("playerLevel");
```

```
  static function main() {
    var db = new Database();

    var playerName = db.getProperty(PLAYER_NAME);
    trace(playerName.toUpperCase());

    db.setProperty(PLAYER_LEVEL, 1);
  }
}
```

Check out the Haxe site’s getting started with Haxe pages for more.

Nim

Nim is a statically typed, multi-paradigm programming language with minimalist and whitespace-sensitive syntax that can compile to JavaScript as well as C, C++.

The language itself is very small, but its metaprogramming capabilities make it attractive to implement features by yourself that you might find built-in to other languages. The building blocks for this are macros, templates, and generics, and with them you can implement things from simple features to different paradigms. This makes Nim an extremely versatile language that can be adapted to your needs, in the spirit of Lisp.

The syntactic abstraction features of Nim allow you to adapt the language to your problems, making true DSLs possible. If you have specialized tasks to solve, you can get a higher level of expressiveness:

```
# Reverse a string
proc reverse(s: string): string =
  result = ""
  for i in countdown(high(s), 0):
    result.add s[i]

var str1 = "Reverse This!"
echo "Reversed: ", reverse(str1)

# Using templates
template genType(name, fieldname: expr, fieldtype: typedesc) =
  type
    name = object
      fieldname: fieldtype

genType(Test, foo, int)

var x = Test(foo: 4566)
echo(x.foo) # 4566
```

The Nim site has some useful getting started docs for more information.

Conclusion

If JavaScript isn’t your favorite language, you can still create web applications without having to suffer the shortcomings of the technology. The options available to create those applications can fill a wide spectrum of taste, from purely functional languages like PureScript to object-oriented ones like Dart. And if you want something more than a one-to-one language translation, you have options like Elm that provide you with tools like a virtual DOM and a built-in architecture.

Chapter 5: 10 Must-have VS Code Extensions for JavaScript Developers

by Michael Wanyoike

In this article, I'll focus on a list of must-have VS Code extensions for JavaScript developers.

Visual Studio Code is undoubtedly the most popular lightweight code editor today. It does borrow heavily from other popular code editors, mostly Sublime Text and Atom. However, its success mainly comes from its ability to provide better performance and stability. In addition, it also provides much needed features like IntelliSense, which were only available in full-sized IDEs like Eclipse or Visual Studio 2017.

The power of VS Code no doubt comes from the marketplace. Thanks to the wonderful open-source community, the editor is now capable of supporting almost every programming language, framework and development technology. Support for a library or framework comes in various ways, which mainly includes snippets, syntax highlighting, Emmet and IntelliSense features for that specific technology.

VS Code Extensions by Category

For this article, I’ll focus on VS Code extensions specifically targeting JavaScript developers. Currently, there are many VS Code extensions that fit this criterion, which of course means I won’t be able to mention all of them. Instead, I’ll highlight VS Code extensions that have gained popularity and those that are indispensable for JavaScript developers. For simplicity, I’ll group them into ten specific categories.

Snippet Extensions

When you first install VS Code, it comes with a several snippets for JavaScript and Typescript. Before you start writing modern JavaScript, you'll need some additional snippets to help you quickly write repetitive ES6/ES7 code:

- VS Code JavaScript (ES6) snippets: currently the most popular, with over 1.2 million installs to date. This extension provides ES6 syntax for JavaScript, TypeScript, HTML, React and Vue extensions.
- JavaScript Snippet Pack: a collection of useful snippets for JavaScript.
- Atom JavaScript Snippet: JavaScript snippets ported from the atom/language-javascript extension.
- JavaScript Snippets: a collection of ES6 snippets. This extension has snippets for Mocha, Jasmine and other BDD testing frameworks.

Syntax Extensions

VS Code comes with pretty good syntax highlighting for JavaScript code. You can change the colors by installing themes. However, you need a syntax highlighter extension if you want a certain level of readability. Here are a couple of them:

- JavaScript Atom Grammar: this extension replaces the JavaScript grammar in Visual Studio Code with the JavaScript grammar from the Atom editor.

- Babel JavaScript: syntax highlighting for ES201x JavaScript, React, FlowType and GraphQL code.

- DotENV: syntax highlighting for `.env` files. Handy if you're working with Node.

Linter Extensions

Writing JavaScript code efficiently with minimum fuss requires a linter that enforces a specific standard style for all team members. ESLint is the most popular, since it supports many styles including Standard, Google and Airbnb. Here are the most popular linter plugins for Visual Studio Code:

- ESLint: this extension integrates ESLint into VS Code. It's the most popular linter extension, with over 6.7 million installs to date. Rules are configured in `.eslintrc.json`.
- JSHint: a code checker extension for JSHint. Uses `.jshintrc`file at the root of your project for configuration.
- JavaScript Standard Style: a linter with zero configuration and rigid rules. Enforces the StandardJS Rules.
- JSLint: linter extension for JSLint.

If you'd like an overview of available linters and their pros and cons, check out our comparison of JavaScript linting tools.

Node Extensions

Every JavaScript project needs to at least one Node package, unless you’re someone who likes doing things the hard way. Here are a few VS Code extensions that will help you work with Node modules more easily.

- npm: uses `package.json`to validate installed packages. Ensures that the installed packages have the correct version numbers, highlights installed packages missing from `package.json`, and packages that haven’t been installed.

- Node.js Modules IntelliSense: autocomplete for JavaScript and TypeScript modules in import statements.

- Path IntelliSense: it’s not really Node related, but you definitely need IntelliSense for local files and this extension will autocomplete filenames.

- Node exec: allows you to execute the current file or your selected code with Node.js.

- View Node Package: quickly view a Node package source with this extension, which allows you to open a Node package repository/documentation straight from VS Code.

- Node Readme: quickly open npm package documentation.

- Search node_modules: this extension allows you to search the `node_modules` folder, which is usually excluded from standard search.

- Import Cost: displays size of an imported package.

Formatting Extensions

Once in a while, you find yourself formatting code that wasn't written in a preferred style. To save time, you can install any of these VS Code extensions to quickly format and refactor existing code:

- Beautify: a jsBeautifier extension that supports JavaScript, JSON, CSS and HTML. Can be customized via `.jsbeautifyrc` file. Most popular formatter with 2.3 million installs to date.

- Prettier Code Formatter: an extension that supports the formatting of JavaScript, TypeScript and CSS using Prettier (an opinionated code formatter). Has over 1.5 million installs to date.

- JS Refactor: provides a number of utilities and actions for refactoring JavaScript code, such as extracting variables/methods, converting existing code to use arrow functions or template literals, and exporting functions.

- JavaScript Booster: an amazing code refactoring tool. Features several coding actions such as converting `var` to `const` or `let`, removing redundant `else` statements, and merging declaration and initialization. Largely inspired by WebStorm.

Browser Extensions

Unless you're writing a console program in JavaScript, you'll most likely be executing your JavaScript code inside a browser. This means you'll need to frequently refresh the page to see the effect of each code update you make. Instead of doing this manually all the time, here are a few tools that can significantly reduce the development time of your iteration process:

- Debugger for Chrome: debug your JavaScript easily in Chrome (by setting breakpoints inside the editor.
- Live Server: local development server with live reload feature for static and dynamic pages.
- Preview on Web Server: provides web server and live preview features.
- PHP Server: useful for testing JavaScript code that needs to run client-side only.
- Rest Client: instead of using a browser or a CURL program to test your REST API endpoints, you can install this tool to interactively run HTTP requests right inside the editor.

Framework Extensions

For most projects, you'll need a suitable framework to structure your code and cut down your development time. VS Code has support for most major frameworks through extensions. However, there are still a number of established frameworks that don't have a fully developed extension yet. Here are some of the VS Code extensions that offer significant functionality.

- Angular 6: snippets for Angular 6. Supports Typescript, HTML, Angular Material ngRx, RxJS & Flex Layout. Has 2.2+ million installs and 172 Angular Snippets to date.
- Angular v5 snippets: provides Angular snippets for TypeScript, RxJS, HTML and Docker files. Has 2.7+ million installs to date.
- React Native/React/Redux snippets for es6/es7: provides snippets in ES6/ES7 syntax for all of these frameworks.
- React Native Tools: provides IntelliSense, commands and debugging features for the React Native framework.
- Vetur: provides syntax highlighting, snippets, Emmet, linting, formatting, IntelliSense and debugging support for the Vue framework. It comes with proper documentation published on GitBook
- Ember: provides command support and IntelliSense for Ember. After installation, all `ember cli` commands are available through Code's own command list.
- Cordova Tools: support for Cordova plugins and the Ionic framework. Provides IntelliSense, debugging and other support features for Cordova-based projects.
- jQuery Code Snippets: provides over 130 jQuery code snippets. Activated by the prefix `jq`.

Testing Extensions

Testing is a critical part of software development, especially for projects that are in production phase. You can get a broad overview of testing in JavaScript and read more about the different kind of tests you can run in our guide — JavaScript Testing: Unit vs Functional vs Integration Tests. Here are some popular VS Code extensions for testing:

- Mocha sidebar: provides support for testing using the Mocha library. This extension helps you run tests directly on the code and shows errors as decorators.

- ES6 Mocha Snippets: Mocha snippets in ES6 syntax. The focus of this extension is to keep the code dry, leveraging arrow functions and omitting curlies by where possible. Can be configured to allow semicolons.

- Jasmine Code Snippets: code snippets for Jasmine test framework.

- Protractor Snippets: end-to-end testing snippets for the Protractor framework. Supports both JavaScript and Typescript.

- Node TDD: provides support for test-driven development for Node and JavaScript projects. Can trigger an automatic test build whenever sources are updated.

Awesome Extensions

I’m just putting this next bunch of VS Code extensions into the “awesome” category, because that best describes them!

- Quokka.js: an awesome debugging tool that provides a rapid prototyping playground for JavaScript code. Comes with excellent documentation.
- Paste as JSON: quickly convert JSON data into JavaScript code.
- Code Metrics: this is another awesome extension that calculates complexity in JavaScript and TypeScript code.

Extension Packs

Now that we’ve come to our final category, I would just like to let you know that the VS Code marketplace has a category for extension packs. Essentially, these are collections of related VS Code extensions bundled into one package for easy installation. Here are some of the better ones:

- Nodejs Extension Pack: this pack contains ESLint, npm, JavaScript (ES6) snippets, Search node_modules, NPM IntelliSense and Path IntelliSense.
- VS Code for Node.js - Development Pack: this one has NPM IntelliSense, ESLint, Debugger for Chrome, Code Metrics, Docker and Import Cost.
- Vue.js Extension Pack: a collection of Vue and JavaScript extensions. It currently contains about 12 VS Code extensions, some of which haven’t been mentioned here such as auto-rename-tag and auto-close-tag.
- Ionic Extension Pack: this pack contains a number of VS Code extensions for Ionic, Angular, RxJS, Cordova and HTML development.

Summary

VS Code’s huge number of quality extensions makes it a popular choice for JavaScript developers. It’s never been easier to write JavaScript code this efficiently. Extensions such as ESLint help you avoid common mistakes, while others such as Debugger for Chrome help you debug your code more easily. Node.js extensions with their IntelliSense features help you import modules correctly, and the availability of tools such as Live Server and REST client reduces your reliance on external tools to complete your work. All of these tools make your iteration process far much easier.

I hope this list has been introduced you to new VS Code extensions that can help you in your workflow.

Chapter 6: Debugging JavaScript Projects with VS Code & Chrome Debugger

by Michael Wanyoike

Debugging JavaScript isn't the most fun aspect of JavaScript programming, but it's a vital skill. This article covers two tools that will help you debug JavaScript like a pro.

Imagine for a moment that the `console.log()` function did not exist in JavaScript. I'm pretty sure the first question you'd ask yourself would be “How am I ever going to confirm my code is working correctly?”

The answer lies in using debugging tools. For a long time, most developers, including myself, have been using `console.log` to debug broken code. It’s quick and easy to use. However, things can get finicky at times if you don't know where and what is causing the bug. Often you'll find yourself laying down `console.log` traps all over your code to see which one will reveal the culprit.

To remedy this, we need to change our habits and start using debugging tools. There are a number of tools available for debugging JavaScript code, such as the Chrome Dev Tools, Node Debugger, Node Inspect and others. In fact, every major browser provides its own tools.

In this this article, we'll look at how to use the debugging facilities provided by Visual Studio Code. We'll also look at how to use the Debugger for Chrome extension that allows VS Code to integrate with Chrome Dev Tools. Once we're finished, you'll never want to use a `console.log()` again.

Prerequisites

For this tutorial, you only need to have a solid foundation in modern JavaScript. We’ll also look at how we can debug a test written using Mocha and Chai. We'll be using a broken project, debug-example, to learn how to fix various bugs without using a single `console.log`. You'll need the following to follow along:

- Node.js
- Visual Studio Code
- Chrome Browser

Start by cloning the debug-example project to your workspace. Open the project in VS Code and install the dependencies via the integrated terminal: `# Install package dependencies npm install # Install global dependencies npm install -g mocha`

Now we're ready to learn how to debug a JavaScript project in VS Code.

Debugging JavaScript in VS Code

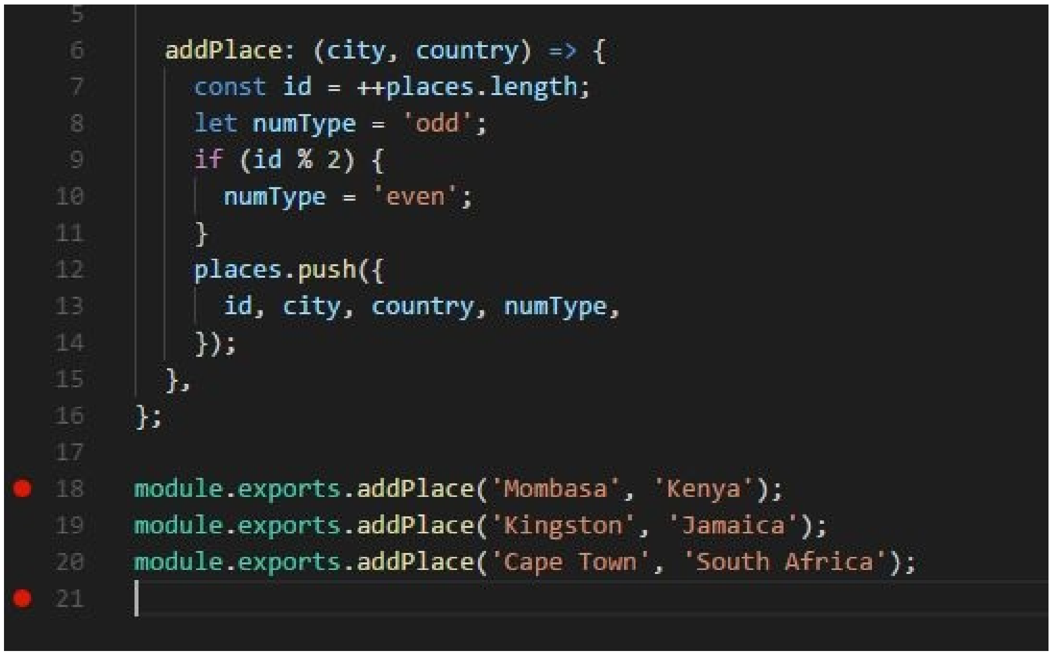
The first file I’d like you to look at is `src/places.js`. You'll need to open the `debug-project` folder in VS Code (*File* > *Open Folder*) and select the file from within the editor.

```
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};
```

The code is pretty simple, and if you have enough experience in coding you might notice it has a couple of bugs. If you do notice them, please ignore them. If not, perfect. Let's add a few of lines at the bottom to manually test the code:

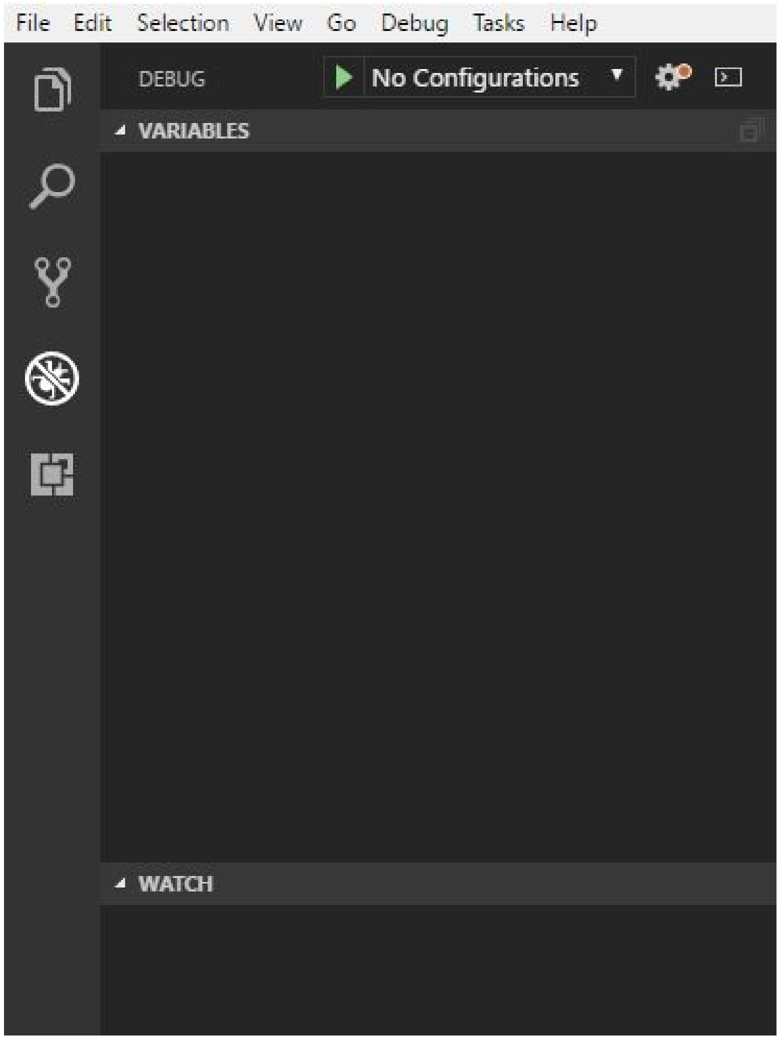
```
module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

Now, I’m sure you’re itching to do a `console.log` to output the value of `places`. But let's not do that. Instead, let's add **breakpoints**. Simply add them by left-clicking on the gutter — that is, the blank space next to the line numbers:



```
  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

See the red dots on the side? Those are the breakpoints. A breakpoint is simply a visual indication telling the debugger tool where to pause execution. Next, on the action bar, click the debug button (the icon that says “No Bugs Allowed”).



Look at the top section. You'll notice there's a gear icon with a red dot. Simply click on it. A debug configuration file, `launch.json`, will be created for you. Update the config like this so that you can run VS Code's debugger on `places.js`:

```
"configurations": [
  {
    "type": "node",
    "request": "launch",
    "name": "Launch Places",
    "program": "${workspaceFolder}\\src\\places.js"
  }
]
```

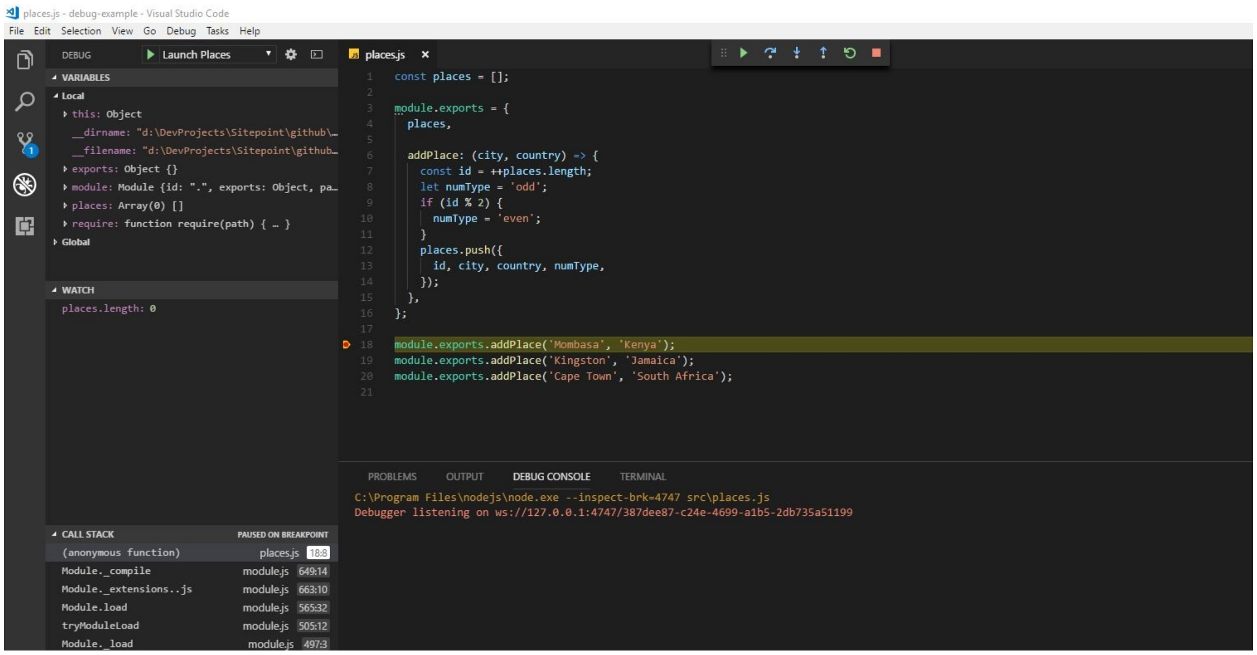
Backslashes

Depending on your operating system, you might have to replace the double backslash (`\\`) with a single forward slash (`/`).

After you've saved the file, you'll notice that the debug panel has a new dropdown, *Launch Places*. To run it, you can:

- hit the Green Play button on the debug panel
- press `F5`
- click *Debug* > *Start Debugging* on the menu bar.

Use whatever method you like and observe the debug process in action:



```
this: Object
  __dirname: "d:\DevProjects\Sitepoint\github\…
  __filename: "d:\DevProjects\Sitepoint\github…
exports: Object {}
module: Module {id: ".", exports: Object, pa…
places: Array(0) []
require: function require(path) { … }
```

```
places.length: 0
```

```
(anonymous function)     places.js  18:8
Module._compile          module.js  649:14
Module._extensions..js   module.js  663:10
Module.load              module.js  565:32
tryModuleLoad            module.js  505:12
Module._load             module.js  497:3
```

```js
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

```
C:\Program Files\nodejs\node.exe --inspect-brk=4747 src\places.js
Debugger listening on ws://127.0.0.1:4747/387dee87-c24e-4699-a1b5-2db735a51199
```

A number of things happen in quick succession once you hit the debug button. First, there's a toolbar that appears at the top of the editor. It has the following controls:

- **Drag Dots anchor**: for moving the toolbar to somewhere it's not blocking anything
- **Continue**: continue the debugging session
- **Step over**: execute code line by line, skipping functions
- **Step into**: execute code line by line, going inside functions
- **Step out**: if already inside a function, this command will take you out
- **Restart**: restarts the debugging session
- **Stop**: stops the debugging session.

Right now, you'll notice that the debug session has paused on your first breakpoint. To continue the session, just hit the *Continue* button, which will cause execution to continue until it reaches the second breakpoint and pause again. Hitting *Continue* again will complete the execution and the debugging session will complete.

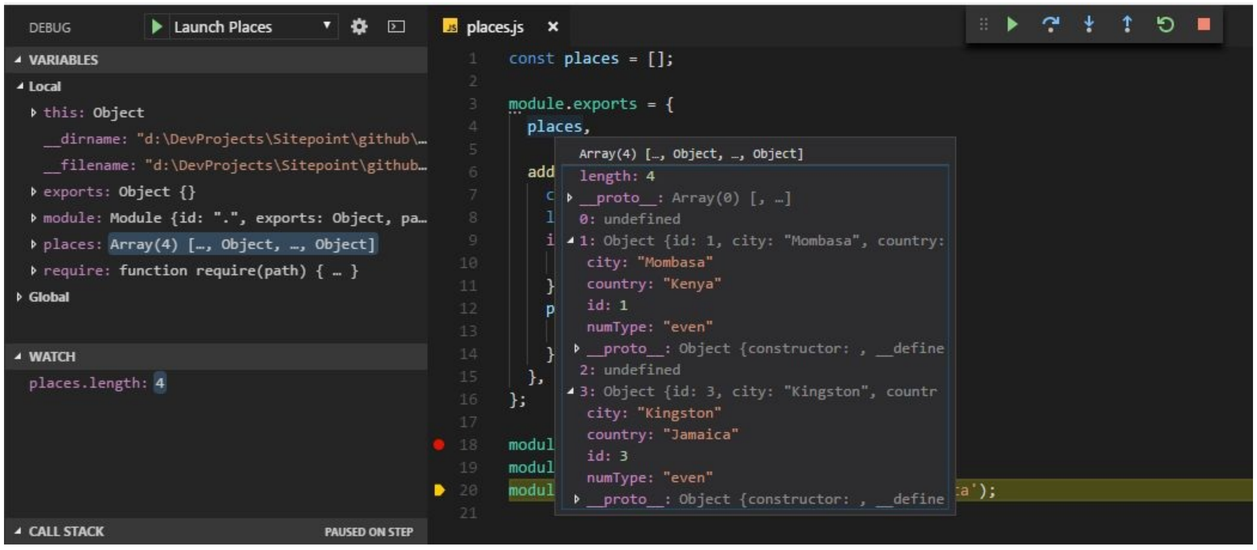
Let's start the debugging process again by hitting `F5`. Make sure the two breakpoints are still in place. When you place a breakpoint, the code pauses at the specified line. It doesn’t execute that line unless you hit *Continue* (`F5`) or *Step Over* (`F10`). Before you hit anything, let's take a look at the sections that make up the debug panel:

- **Variables**: displays local and global variables within the current scope (i.e. at the point of execution)
- **Watch**: you can manually add expressions of variables you want to monitor
- **Call Stack**: displays a call stack of the highlighted code
- **Breakpoints**: displays a list of files with breakpoints, along with their line numbers.

To add an expression to the *Watch* section, simply click the + sign and add any valid JavaScript expression — such as `places.length`. When the debugger pauses, if your expression is in scope, the value will be printed out. You can also hover over variables that are currently in scope. A popup will appear displaying their values.

Currently the `places` array is empty. Press any navigation control to see how debugging works. For example, *Step over* will jump into the next line, while *Step into* will navigate to the `addPlace` function. Take a bit of time to get familiar with the controls.

As soon as you’ve done some stepping, hover over the `places` variable. A popup will appear. Expand the values inside until you have a similar view:

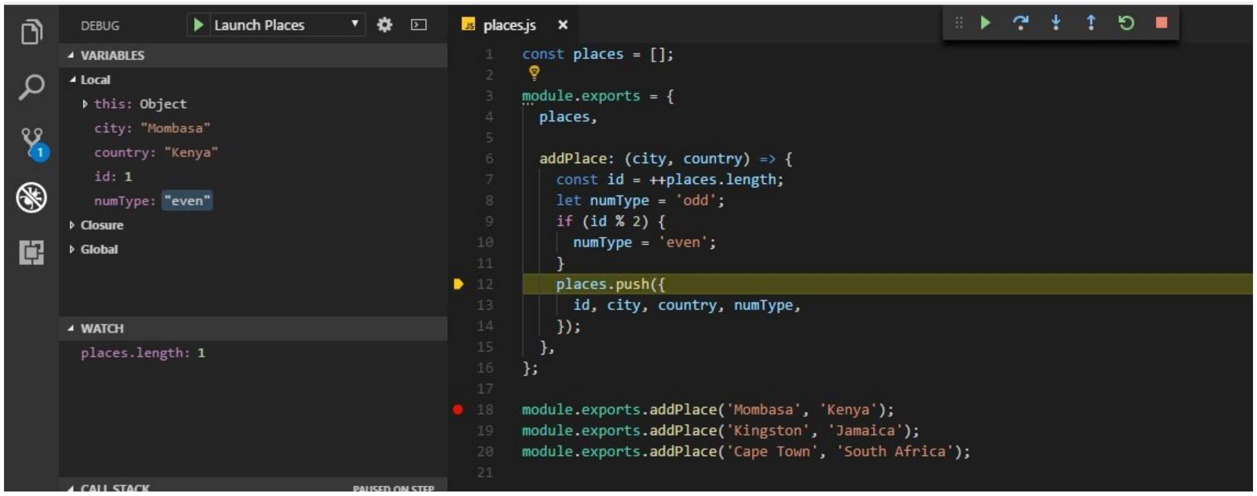


```
DEBUG  Launch Places
VARIABLES
 Local
  this: Object
   __dirname: "d:\DevProjects\Sitepoint\github\…
   __filename: "d:\DevProjects\Sitepoint\github…
  exports: Object {}
  module: Module {id: ".", exports: Object, pa…
  places: Array(4) […, Object, …, Object]
  require: function require(path) { … }
 Global
WATCH
 places.length: 4
CALL STACK                                PAUSED ON STEP
```

```
places.js
const places = [];

module.exports = {
  places,

  add
    c
    l
    i

    }
    p

    }
  },
};

modul
modul
modul                                   a');
```

```
Array(4) […, Object, …, Object]
length: 4
 __proto__: Array(0) [, …]
0: undefined
1: Object {id: 1, city: "Mombasa", country:
  city: "Mombasa"
  country: "Kenya"
  id: 1
  numType: "even"
  __proto__: Object {constructor: , __define
2: undefined
3: Object {id: 3, city: "Kingston", countr
  city: "Kingston"
  country: "Jamaica"
  id: 3
  numType: "even"
  __proto__: Object {constructor: , __define
```

You can also inspect all variables that are in scope in the *Variables* section.



```
DEBUG  Launch Places
VARIABLES
 Local
  this: Object
   city: "Mombasa"
   country: "Kenya"
   id: 1
   numType: "even"
 Closure
 Global
WATCH
 places.length: 1
CALL STACK                                PAUSED ON STEP
```

```
places.js
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

That's pretty awesome compared to what we normally do with `console.log`. The debugger allows us to inspect variables at a deeper level. You may have also noticed a couple of problems with the `places` array output:

1. there are multiple blanks in the array — that is, `places[0]` and `places[2]` are `undefined`
2. the `numType` property displays `even` for odd `id` values.

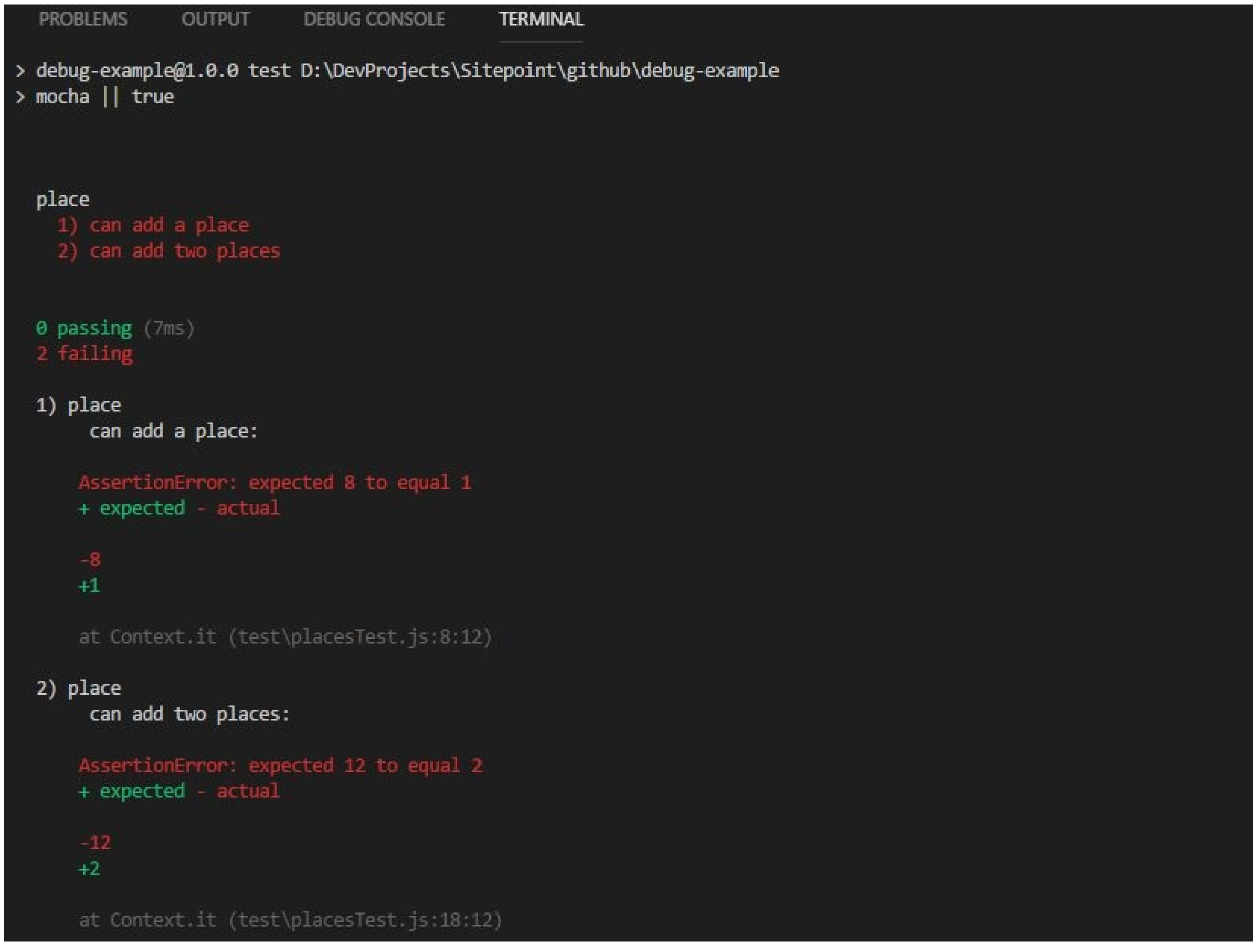
For now, just end the debugging session. We'll fix them in the next section.

Debugging Tests with Mocha

Open `test/placesTest.js` and review the code that's been written to test the code in `places.test`. If you've never used Mocha before, you need to install it globally first in order to run the tests.

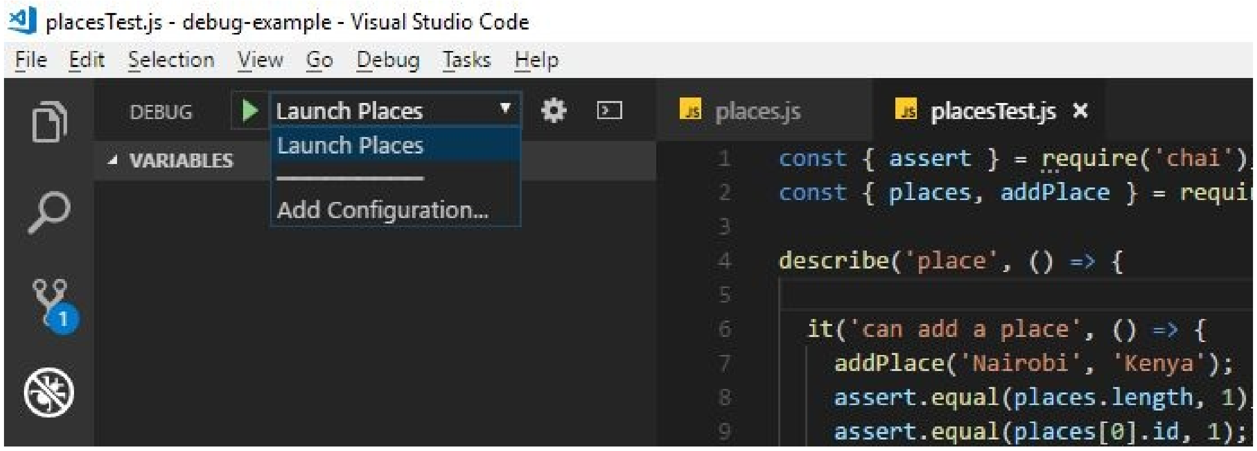
```
# Install mocha globally
npm install -g mocha

# Run mocha tests
mocha
```

You can also run `npm test` to execute the tests. You should get the following output:



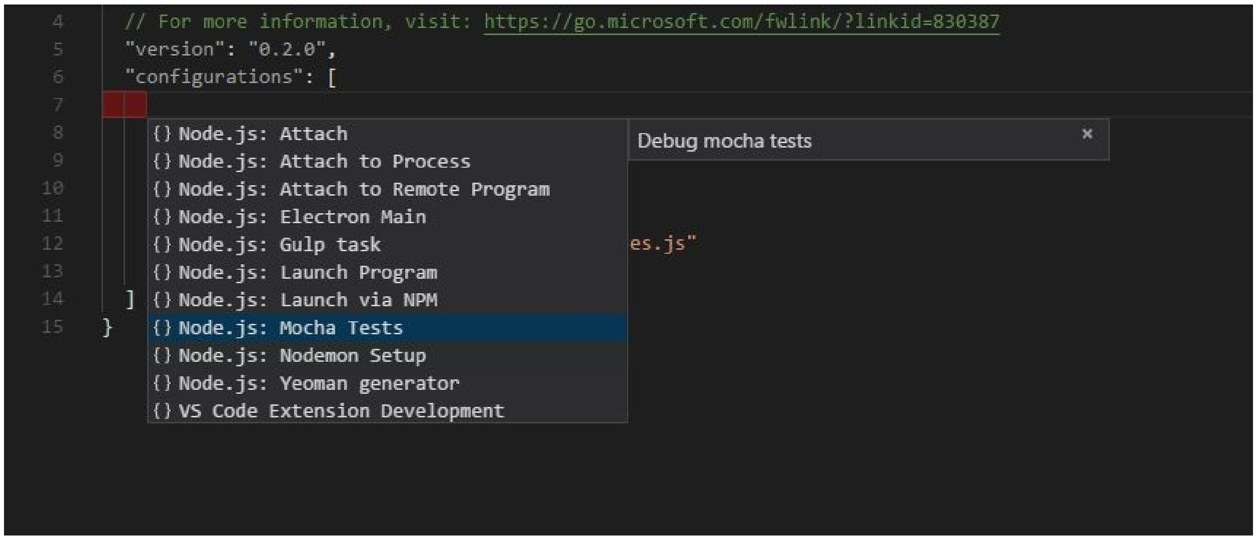
```
> debug-example@1.0.0 test D:\DevProjects\Sitepoint\github\debug-example
> mocha || true



  place
    1) can add a place
    2) can add two places


  0 passing (7ms)
  2 failing

  1) place
       can add a place:

      AssertionError: expected 8 to equal 1
      + expected - actual

      -8
      +1

      at Context.it (test\placesTest.js:8:12)

  2) place
       can add two places:

      AssertionError: expected 12 to equal 2
      + expected - actual

      -12
      +2

      at Context.it (test\placesTest.js:18:12)
```

All the tests are failing. To find out the problem, we're going to run the tests in debug mode. To do that, we need a new configuration. Go to the debug panel and click the dropdown to access the `Add Configuration` option:


placesTest.js - debug-example - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG
Launch Places
Launch Places
Add Configuration...
VARIABLES
places.js
placesTest.js

```
const { assert } = require('chai')
const { places, addPlace } = requi

describe('place', () => {

  it('can add a place', () => {
    addPlace('Nairobi', 'Kenya');
    assert.equal(places.length, 1)
    assert.equal(places[0].id, 1);
```

The `launch.json` file will open for you with a popup listing several configurations for you to choose from.



```
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [

    ]
}
```

{} Node.js: Attach
{} Node.js: Attach to Process
{} Node.js: Attach to Remote Program
{} Node.js: Electron Main
{} Node.js: Gulp task
{} Node.js: Launch Program
{} Node.js: Launch via NPM
{} Node.js: Mocha Tests
{} Node.js: Nodemon Setup
{} Node.js: Yeoman generator
{} VS Code Extension Development
Debug mocha tests
es.js"

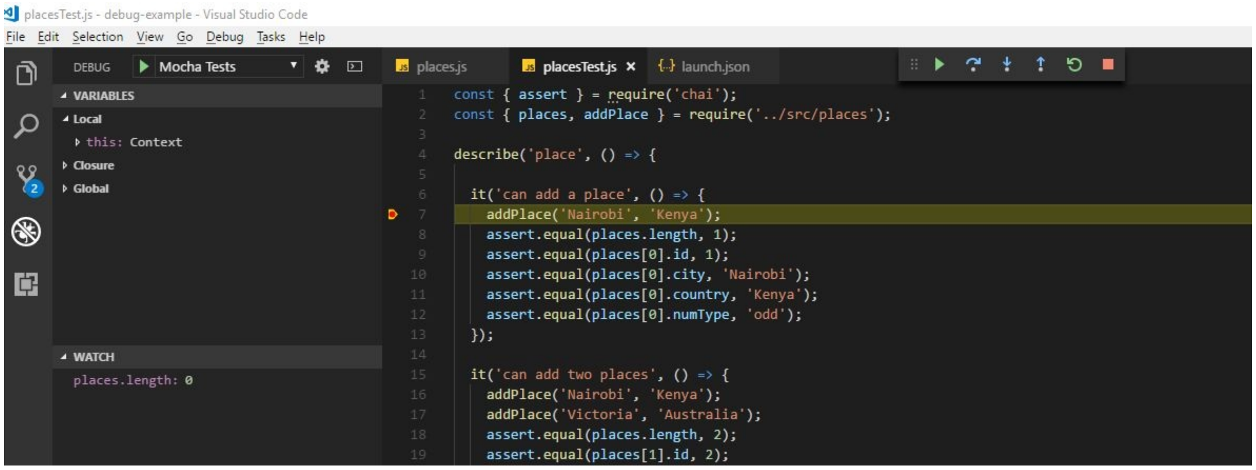
Simply select *Mocha Tests*. The following configuration will be inserted for you:

```
{
  "type": "node",
  "request": "launch",
  "name": "Mocha Tests",
  "program": "${workspaceFolder}/node_modules/mocha/bin/_mocha",
  "args": [
    "-u",
    "tdd",
    "--timeout",
    "999999",
    "--colors",
    "${workspaceFolder}/test"
  ],
  "internalConsoleOptions": "openOnSessionStart"
},
```

The default settings are fine. Go back to the dropdown and select *Mocha Tests*. You'll need to comment out the last three lines you added in `places.js`; otherwise the tests won't run as expected. Go back to `placesTest.js` and add a breakpoint on the line just before where the first test failure occurs. That should be line seven, where it says:

```
addPlace('Nairobi', 'Kenya');
```

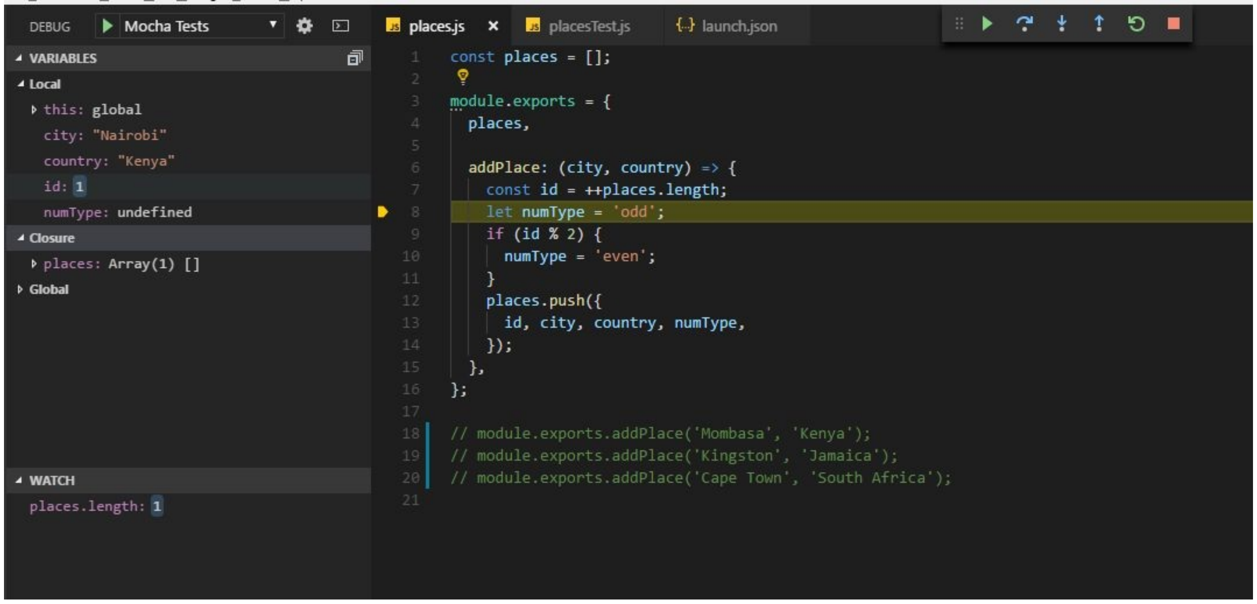
Make sure to add a `places.length` expression in the watch section. Hit the *Play* button to start the debugging session.



```
1   const { assert } = require('chai');
2   const { places, addPlace } = require('../src/places');
3
4   describe('place', () => {
5
6     it('can add a place', () => {
7       addPlace('Nairobi', 'Kenya');
8       assert.equal(places.length, 1);
9       assert.equal(places[0].id, 1);
10      assert.equal(places[0].city, 'Nairobi');
11      assert.equal(places[0].country, 'Kenya');
12      assert.equal(places[0].numType, 'odd');
13    });
14
15    it('can add two places', () => {
16      addPlace('Nairobi', 'Kenya');
17      addPlace('Victoria', 'Australia');
18      assert.equal(places.length, 2);
19      assert.equal(places[1].id, 2);
```

At the start of the test, `places.length` should read zero. If you hit *Step over*, `places.length` reads 2, yet only one place has been added. How can that be?

Restart the debugging session, and this time use *Step into* to navigate to the `addPlace` function. The debugger will navigate you to `places.js`. The value of `places.length` is still zero. Click *Step over* to execute the current line.



```
this: global
city: "Nairobi"
country: "Kenya"
id: 1
numType: undefined
places: Array(1) []
```

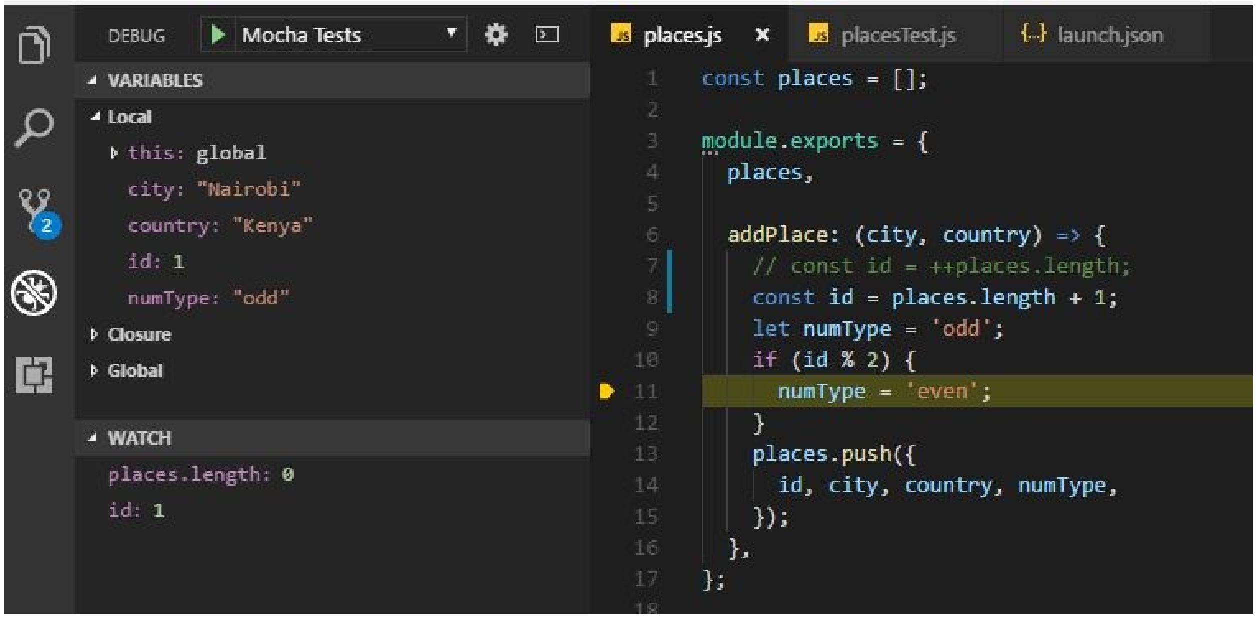
```
places.length: 1
```

```js
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

// module.exports.addPlace('Mombasa', 'Kenya');
// module.exports.addPlace('Kingston', 'Jamaica');
// module.exports.addPlace('Cape Town', 'South Africa');
```

Aha! The value of `places.length` just incremented by 1, yet we haven't added anything to the array. The problem is caused by the ++ operator which is mutating the array's length. To fix this, simply replace the line with:

```
const id = places.length + 1;
```

This way, we can safely get the value of `id` without changing the value of `places.length`. While we’re still in debug mode, let's try to fix another problem where the `numType` property is given the value `even` while `id` is 1. The problem seems to be the modulus expression inside the if statement:

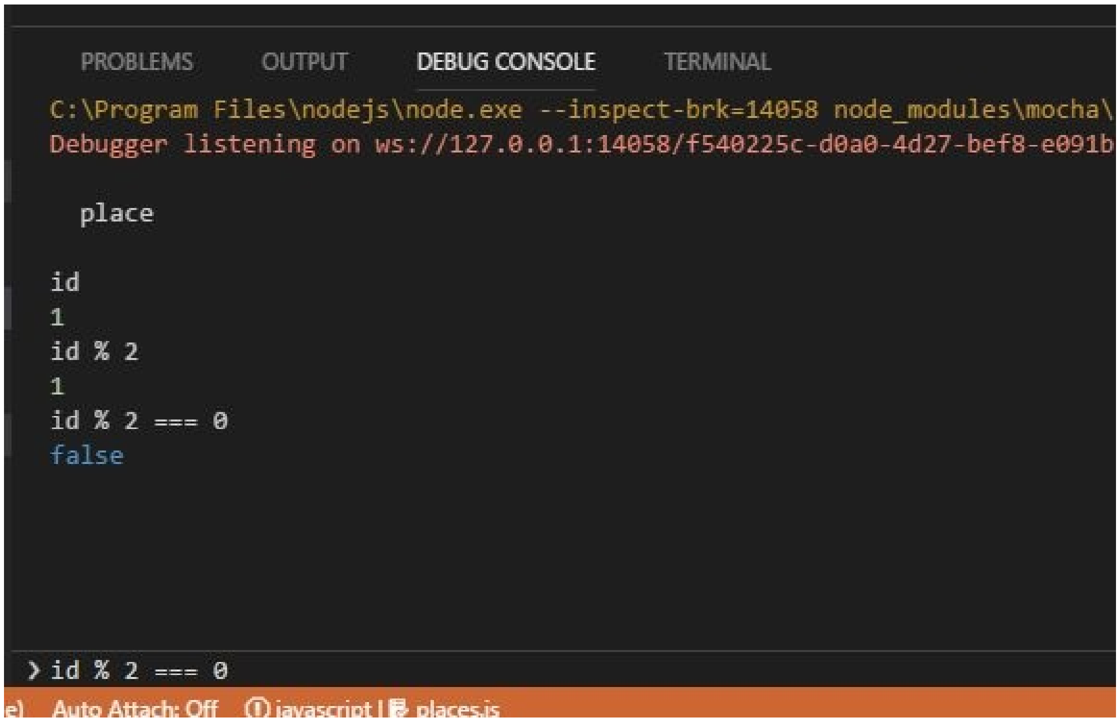


```
this: global
city: "Nairobi"
country: "Kenya"
id: 1
numType: "odd"
```

```
places.length: 0
id: 1
```

```js
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    // const id = ++places.length;
    const id = places.length + 1;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};
```

Let's do a quick experiment using the debug console. Start typing a proper expression for the `if` statement:



```
C:\Program Files\nodejs\node.exe --inspect-brk=14058 node_modules\mocha\
Debugger listening on ws://127.0.0.1:14058/f540225c-d0a0-4d27-bef8-e091b

  place

id
1
id % 2
1
id % 2 === 0
false

> id % 2 === 0
```

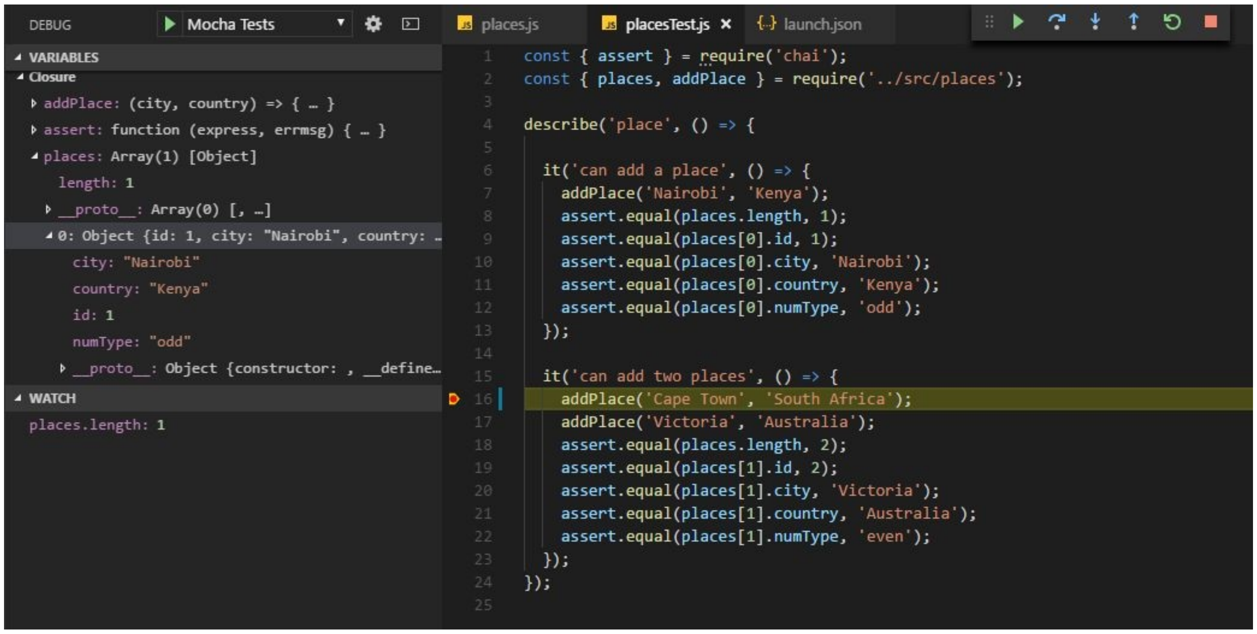
The debug console is similar to the browser console. It allows you to perform experiments using variables that are currently in scope. By trying out a few ideas in the console, you can easily find the solution without ever leaving the editor. Let's now fix the failing if statement:

```
if (id % 2 === 0) {
  numType = 'even';
}
```

Restart the debug session and hit *Continue* to skip the current breakpoint. The first test, “can add a place”, is now passing. But the second test isn't. To fix this, we need another breakpoint. Remove the current one and place a new breakpoint on line 16, where it says:

```
addPlace('Cape Town', 'South Africa');
```

Start a new debugging session:



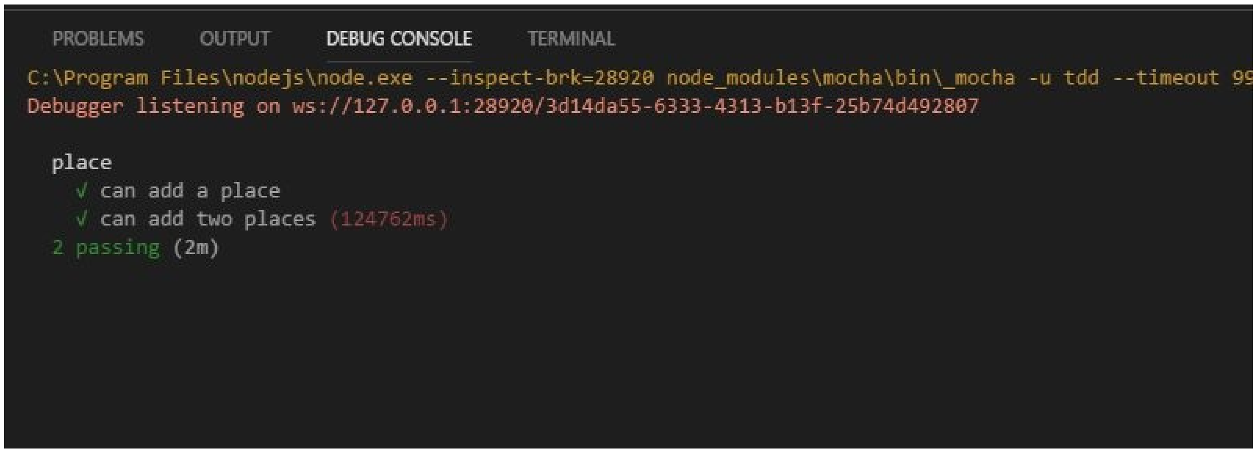
```
addPlace: (city, country) => { … }
assert: function (express, errmsg) { … }
places: Array(1) [Object]
  length: 1
  __proto__: Array(0) [, …]
  0: Object {id: 1, city: "Nairobi", country: …
    city: "Nairobi"
    country: "Kenya"
    id: 1
    numType: "odd"
    __proto__: Object {constructor: , __define…

places.length: 1
```

```
const { assert } = require('chai');
const { places, addPlace } = require('../src/places');

describe('place', () => {

  it('can add a place', () => {
    addPlace('Nairobi', 'Kenya');
    assert.equal(places.length, 1);
    assert.equal(places[0].id, 1);
    assert.equal(places[0].city, 'Nairobi');
    assert.equal(places[0].country, 'Kenya');
    assert.equal(places[0].numType, 'odd');
  });

  it('can add two places', () => {
    addPlace('Cape Town', 'South Africa');
    addPlace('Victoria', 'Australia');
    assert.equal(places.length, 2);
    assert.equal(places[1].id, 2);
    assert.equal(places[1].city, 'Victoria');
    assert.equal(places[1].country, 'Australia');
    assert.equal(places[1].numType, 'even');
  });
});
```

There! Look at the *Variables* section. Even before the second test begins we discover that the `places` array already has existing data created by the first test. This has obviously polluted our current test. To fix this, we need to implement some kind of `setup` function that resets the `places` array for each test. To do this in Mocha, just add the following code before the tests:

```
beforeEach(() => {
  places.length = 0;
});
```

Restart the debugger and let it pause on the breakpoint. Now the `places` array has a clean state. This should allow our test to run unpolluted. Just click *Continue* to let the rest of the test code execute.

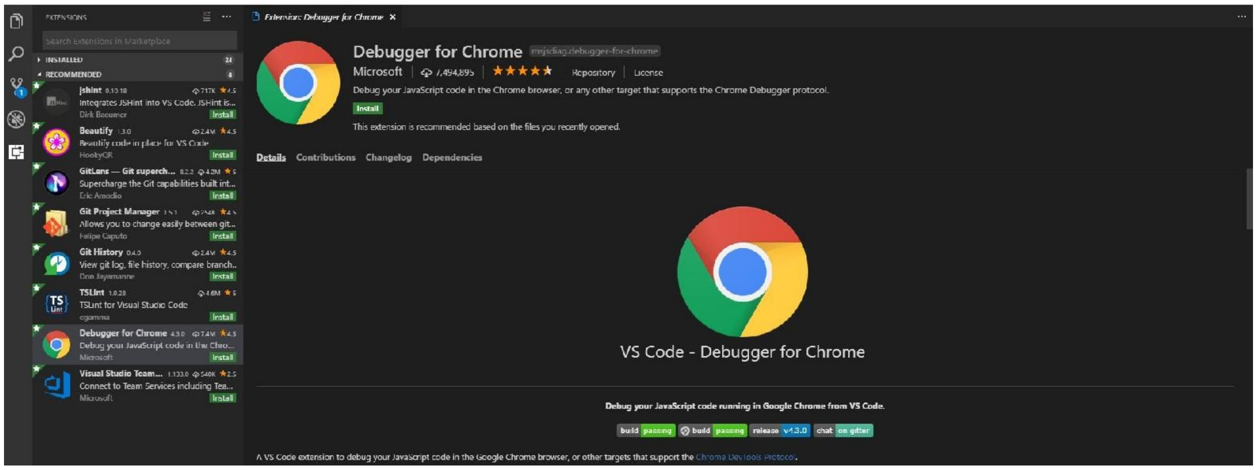


```
C:\Program Files\nodejs\node.exe --inspect-brk=28920 node_modules\mocha\bin\_mocha -u tdd --timeout 99
Debugger listening on ws://127.0.0.1:28920/3d14da55-6333-4313-b13f-25b74d492807

  place
    √ can add a place
    √ can add two places (124762ms)
  2 passing (2m)
```

All tests are now passing. You should feel pretty awesome, since you've learned how to debug code without writing a single line of `console.log`. Let's now look at how to debug client-side code using the browser.

Debugging JavaScript with Chrome Debugger

Now that you've become familiar with the basics of debugging JavaScript in VS Code, we're going to see how to debug a slightly more complex project using the Debugger for Chrome extension. Simply open the marketplace panel via the action bar. Search for the extension and install it.



Integrates JSHint into VS Code. JSHint is...

Beautify code in place for VS Code

Supercharge the Git capabilities built int...

Allows you to change easily between git...

View git log, file history, compare branch...

TSLint for Visual Studio Code

Debug your JavaScript code in the Chro...

Connect to Team Services including Tea...

Debug your JavaScript code in the Chrome browser, or any other target that supports the Chrome Debugger protocol.

This extension is recommended based on the files you recently opened.

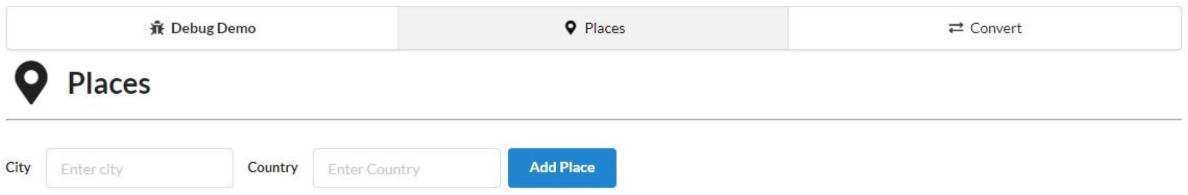
Debug your JavaScript code running in Google Chrome from VS Code.

A VS Code extension to debug your JavaScript code in the Google Chrome browser, or other targets that support the Chrome DevTools Protocol.

After installation, hit reload to activate the extension. Let's quickly review the code that we'll be debugging. The web application is mostly a client-side JavaScript project that's launched by running an Express server:

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

const app = express();
const port = 3000;

// Set public folder as root
app.use(express.static('public'));

// Provide access to node_modules folder
app.use('scripts', express.static(`${__dirname}node_modules/`));

// Redirect all traffic to index.html
app.use((req, res) =>
res.sendFile(`${__dirname}/public/index.html`));

app.listen(port, () => {
  console.info('listening on %d', port);
});
```

All the client-side code is in the `public` folder. The project's dependencies include Semantic-UI-CSS, jQuery, Vanilla Router, Axios and Handlebars. This is what the project looks like when you run it with `npm start`. You'll have to open the URL localhost:3000 in your browser to view the application.



Try to add a new place. When you do, you'll see that nothing seems to be happening. Clearly something’s going wrong, so it's time to look under the hood. We'll first review the code before we start our debugging session. Open `public/index.html`. Our focus currently is this section:

```
<!-- TEMPLATES -->
<!-- Places Form Template -->
<script id="places-form-template" type="text/x-handlebars-
template">
  <h1 class="ui header">
    <i class="map marker alternate icon"></i>
    <div class="content"> Places</div>
  </h1>
  <hr>
  <br>
  <form class="ui form">
    <div class="fields">
      <div class="inline field">
        <label>City</label>
        <input type="text" placeholder="Enter city" id="city"
name="city">
      </div>
      <div class="inline field">
        <label>Country</label>
        <input type="text" placeholder="Enter Country"
name="country">
      </div>
      <div class="ui submit primary button">Add Place</div>
    </div>
  </form>
  <br>
  <div id="places-table"></div>
</script>

<!-- Places Table Template -->
<script id="places-table-template" type="text/x-handlebars-
template">
  <table class="ui celled striped table">
    <thead>
      <tr>
        <th>Id</th>
        <th>City</th>
        <th>Country</th>
        <th>NumType</th>
      </tr>
    </thead>
    <tbody>
      {{#each places}}
      <tr>
        <td>{{id}}</td>
        <td>{{city}}</td>
        <td>{{country}}</td>
        <td>{{numType}}</td>
      </tr>
      {{/each}}
    </tbody>
  </table>
</script>
```

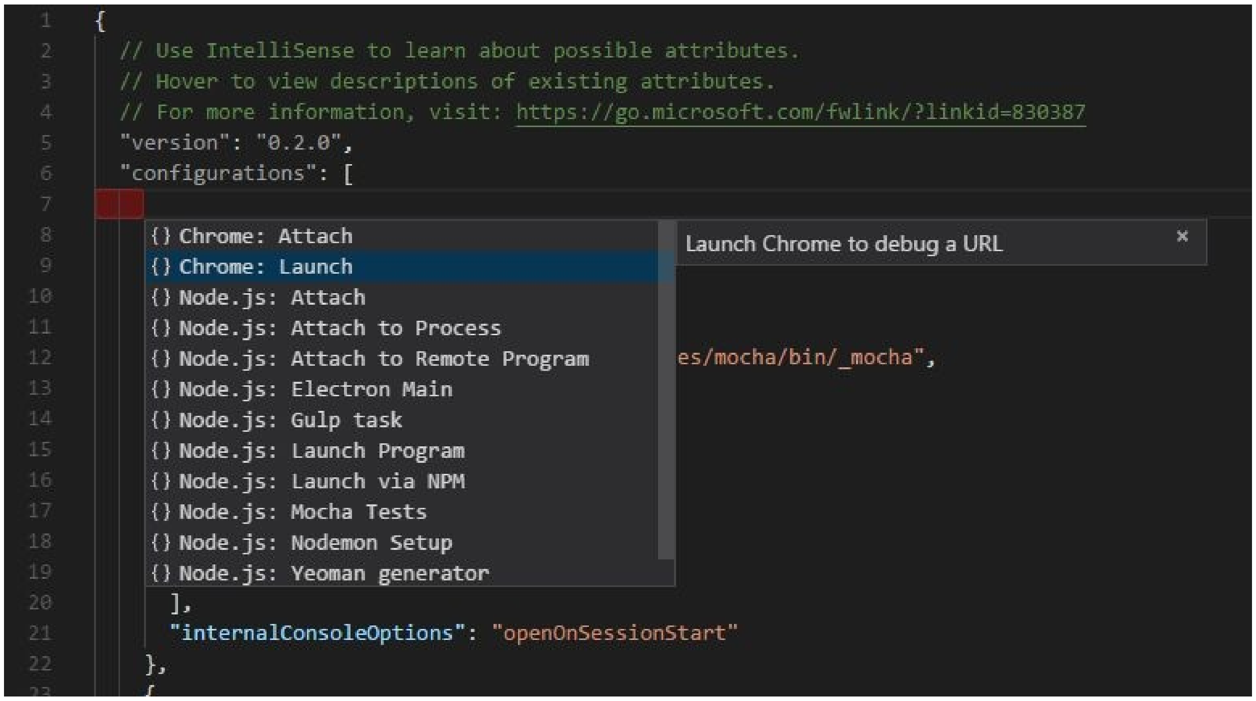
If you take a quick glance, the code will appear to be correct. So the problem must be in `app.js`. Open the file and analyze the code there. Below are the sections of code you should pay attention to. Take your time to read the comments in order to understand the code.

```
// Load DOM roots
const el = $('#app');
const placesTable = $('#places-table');

// Initialize empty places array
const places = [];

// Compile Templates
const placesFormTemplate = Handlebars.compile($('#places-form-
template').html());
const placesTableTemplate = Handlebars.compile($('#places-table-
template').html());

const addPlace = (city, country) => {
  const id = places.length + 1;
  const numType = (id % 2 === 0) ? 'even' : 'odd';
  places.push({
    id, city, country, numType,
  });
};

// Populate places array
addPlace('Nairobi', 'Kenya');

...

// Places View - ''
router.add('', () => {
  // Display Places Form
  const html = placesFormTemplate();
  el.html(html);
  // Form Validation Rules
  $('.ui.form').form({
    fields: {
      city: 'empty',
      country: 'empty',
    },
  });
  // Display Places Table
  const tableHtml = placesTableTemplate({ places });
  placesTable.html(tableHtml);
  $('.submit').on('click', () => {
    const city = $('#city').val();
    const country = $('#country').val();
    addPlace(city, country);
    placesTable.html(placesTableTemplate({ places }));
    $('form').form('clear');
    return false;
  });
});
```

Everything seems fine. But what could be the problem? Let's place a breakpoint on line 53 where it says:

```
placesTable.html(tableHtml);
```

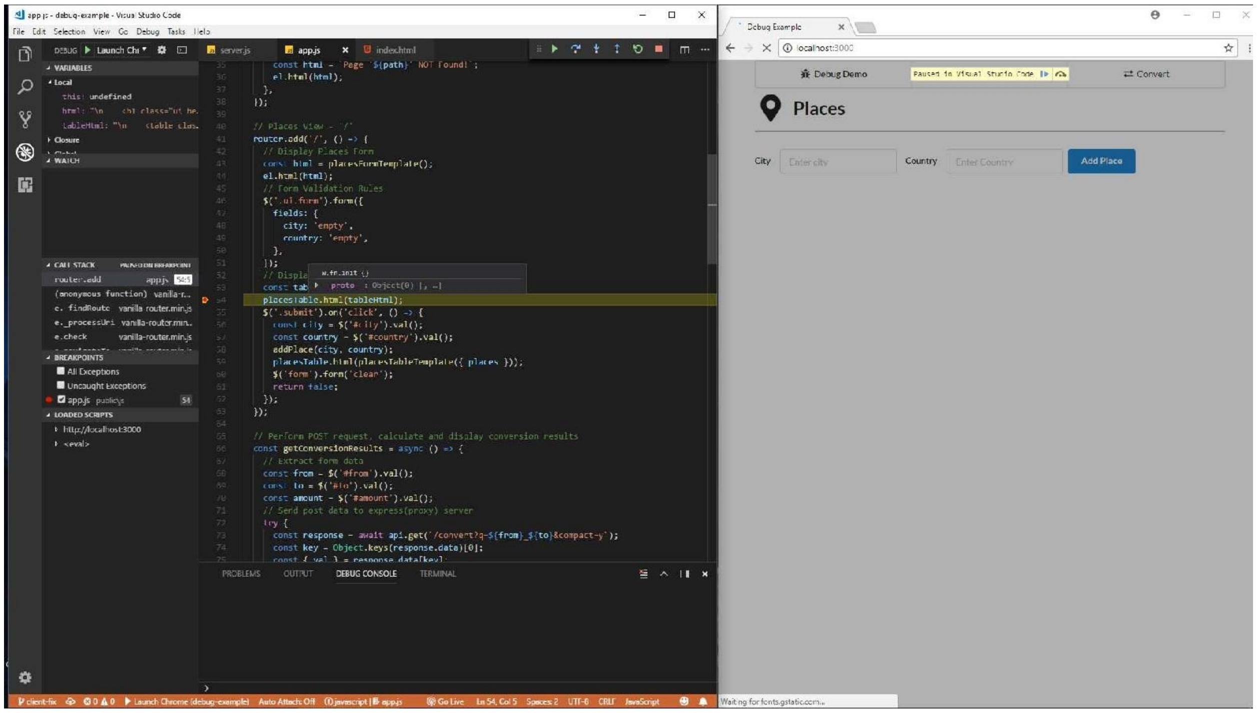
Next, create a *Chrome* configuration via the debug panel. Select the highlighted option:


```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [

      {} Chrome: Attach
      {} Chrome: Launch                         Launch Chrome to debug a URL
      {} Node.js: Attach
      {} Node.js: Attach to Process
      {} Node.js: Attach to Remote Program      es/mocha/bin/_mocha",
      {} Node.js: Electron Main
      {} Node.js: Gulp task
      {} Node.js: Launch Program
      {} Node.js: Launch via NPM
      {} Node.js: Mocha Tests
      {} Node.js: Nodemon Setup
      {} Node.js: Yeoman generator
      ],
      "internalConsoleOptions": "openOnSessionStart"
    },
    {
```

Then update the Chrome config as follows to match our environment:

```
{
  "type": "chrome",
  "request": "launch",
  "name": "Launch Chrome",
  "url": "http://localhost:3000",
  "webRoot": "${workspaceFolder}/public"
},
```

Next, start the server as normal using `npm start` or `node server`. Then select *Launch Chrome* and start the debugging session. A new instance of Chrome will be launched in debug mode and execution should pause where you set the breakpoint. Now’s a good time to position Visual Studio Code and the Chrome instance side by side so you can work efficiently.

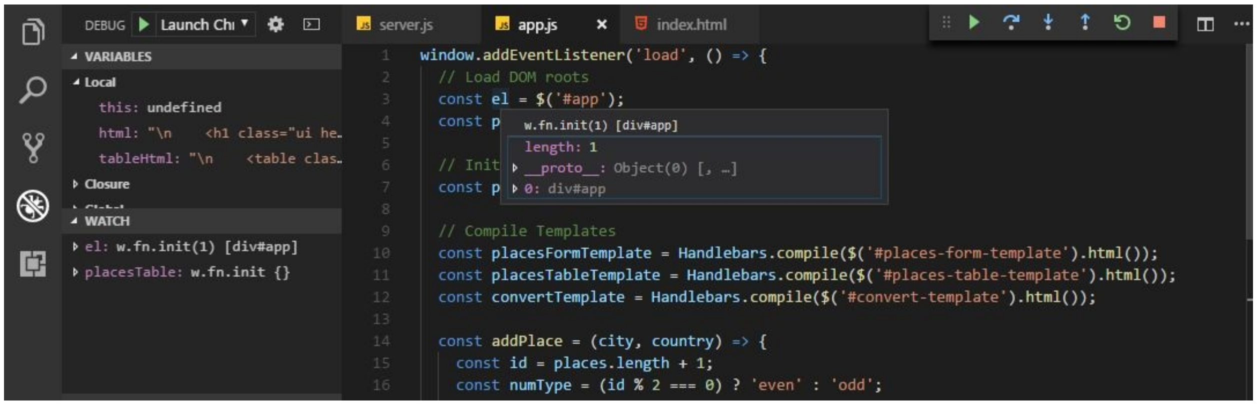


```
35      const html = `Page '${path}' NOT Found!`;
36      el.html(html);
37    },
38  });
39
40  // Places View - '/'
41  router.add('/', () => {
42    // Display Places Form
43    const html = placesFormTemplate();
44    el.html(html);
45    // Form Validation Rules
46    $('.ui.form').form({
47      fields: {
48        city: 'empty',
49        country: 'empty',
50      },
51    });
52    // Displa
53    const tab
54    placesTable.html(tableHtml);
55    $('.submit').on('click', () => {
56      const city = $('#city').val();
57      const country = $('#country').val();
58      addPlace(city, country);
59      placesTable.html(placesTableTemplate({ places }));
60      $('form').form('clear');
61      return false;
62    });
63  });
64
65  // Perform POST request, calculate and display conversion results
66  const getConversionResults = async () => {
67    // Extract form data
68    const from = $('#from').val();
69    const to = $('#to').val();
70    const amount = $('#amount').val();
71    // Send post data to express(proxy) server
72    try {
73      const response = await api.get(`/convert?q=${from}_${to}&compact=y`);
74      const key = Object.keys(response.data)[0];
```

```
w.fn.init {}
▸ __proto__: Object(0) [, …]
```

```
VARIABLES
  Local
    this: undefined
    html: "\n    <h1 class="ui he…
    tableHtml: "\n    <table clas…
  Closure
WATCH
CALL STACK                PAUSED ON BREAKPOINT
  router.add                  app.js 54:5
  (anonymous function)        vanilla-r…
  e._findRoute                vanilla-router.min.js
  e._processUri               vanilla-router.min…
  e.check                     vanilla-router.min.js
BREAKPOINTS
  All Exceptions
  Uncaught Exceptions
  app.js  public/js           54
LOADED SCRIPTS
  http://localhost:3000
  <eval>
```

Mouse over the `placesTable` constant. A popup appears, but it seems empty. In the watch panel, add the expressions `el` and `placesTable`. Or, alternatively, just scroll up to where the constants have been declared.



```
1   window.addEventListener('load', () => {
2     // Load DOM roots
3     const el = $('#app');
4     const p
5
6     // Init
7     const p
8
9     // Compile Templates
10    const placesFormTemplate = Handlebars.compile($('#places-form-template').html());
11    const placesTableTemplate = Handlebars.compile($('#places-table-template').html());
12    const convertTemplate = Handlebars.compile($('#convert-template').html());
13
14    const addPlace = (city, country) => {
15      const id = places.length + 1;
16      const numType = (id % 2 === 0) ? 'even' : 'odd';
```

```
w.fn.init(1) [div#app]
  length: 1
▸ __proto__: Object(0) [, …]
▸ 0: div#app
```

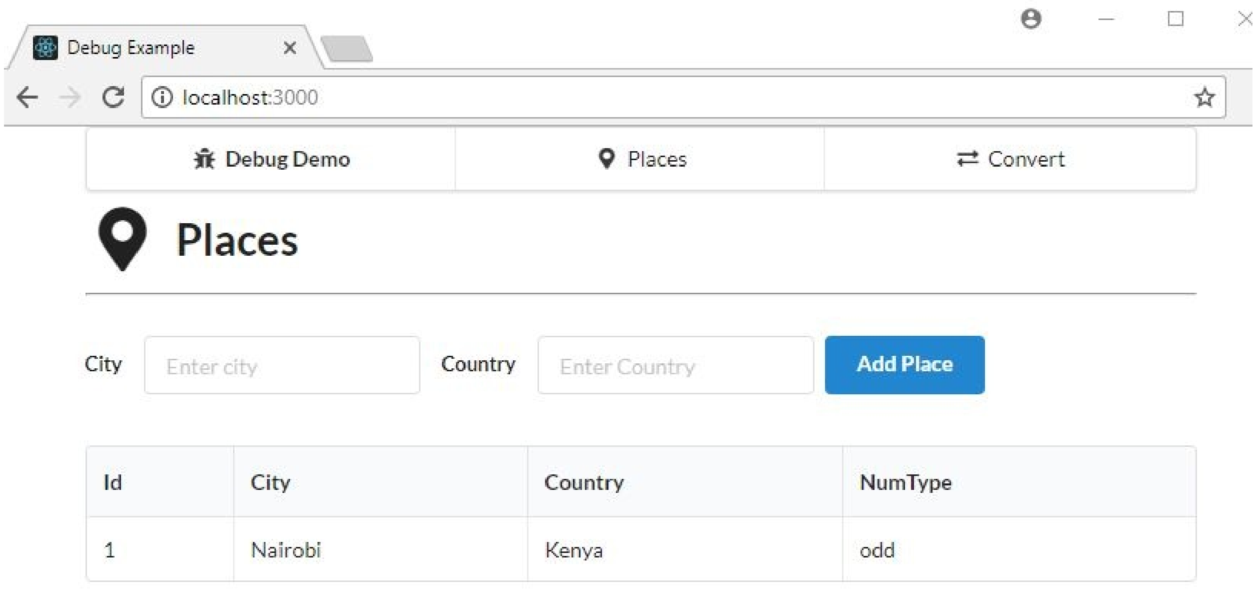
```
VARIABLES
  Local
    this: undefined
    html: "\n    <h1 class="ui he…
    tableHtml: "\n    <table clas…
  Closure
WATCH
  ▸ el: w.fn.init(1) [div#app]
  ▸ placesTable: w.fn.init {}
```

Notice that `el` is populated but `placesTable` is empty. This means that jQuery was unable to find the element `#places-table`. Let's go back to `public/index.html` and find where this `#places-table` is located.

Aha! The table div we want is located on line 55, right inside the `places-form-template`. This means the div `#places-table` can only be found after the template, `places-form-template`, has been loaded. To fix this, just go back to `app.js` and move the code to line 52, right after the “Display Places Table” comment:

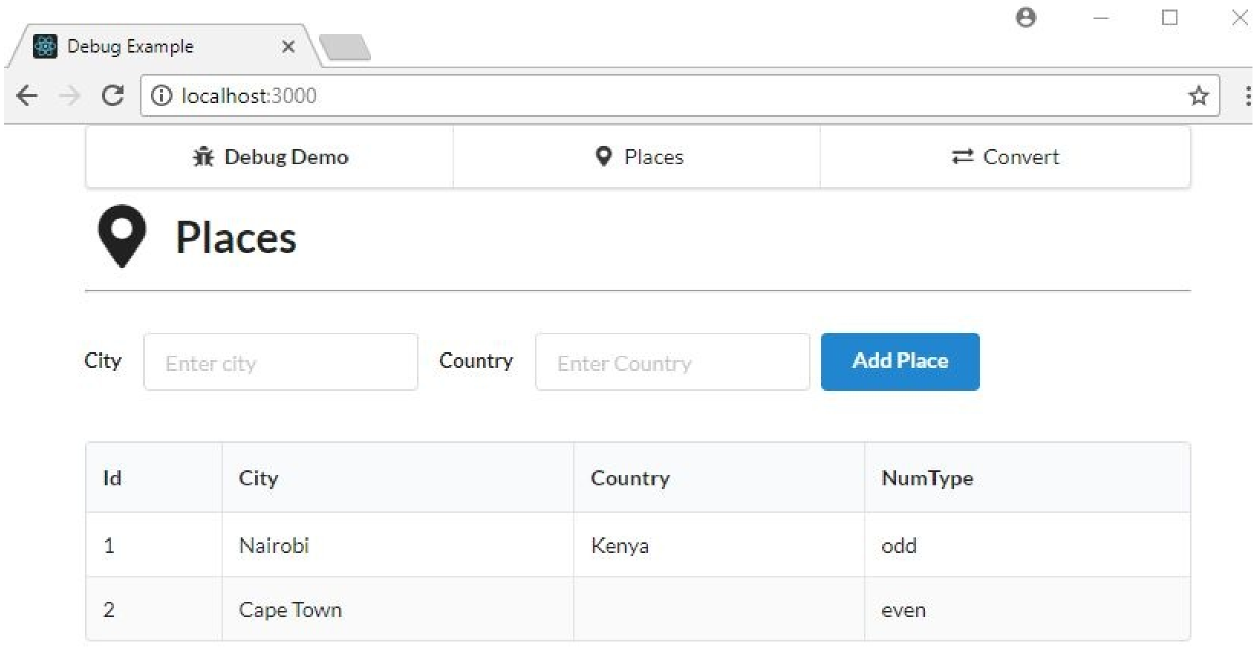
```
const placesTable = $('#places-table');
```

Save the file, and restart the debugging session. When it reaches the breakpoint, just hit *Continue* and let the code finish executing. The table should now be visible:



Id	City	Country	NumType
1	Nairobi	Kenya	odd

You can now remove the breakpoint. Let's try adding a new place — for example, Cape Town, South Africa

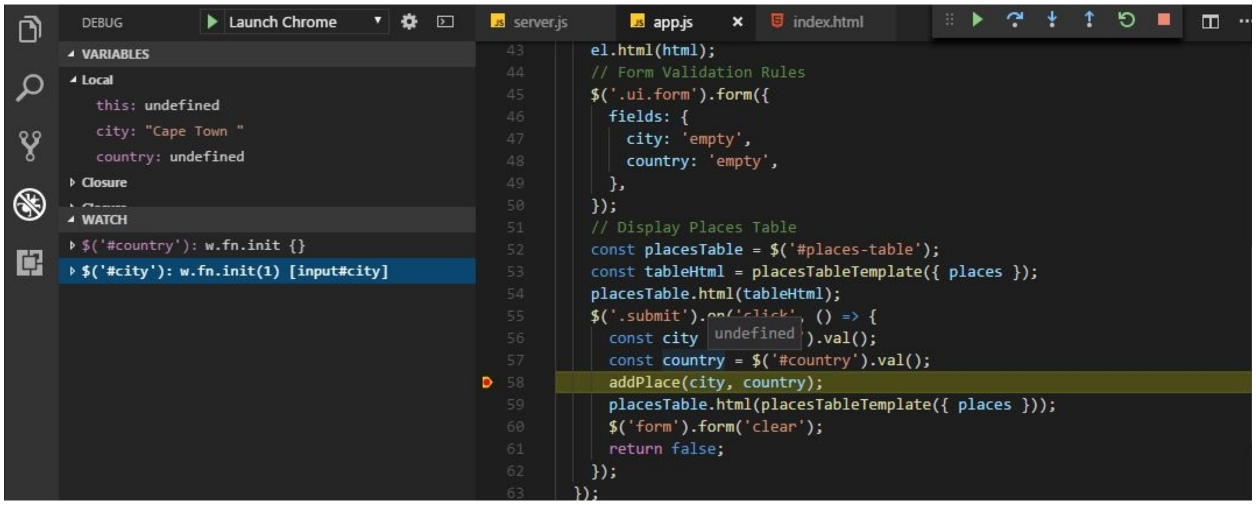


Id	City	Country	NumType
1	Nairobi	Kenya	odd
2	Cape Town		even

Hmm … that's not right. The place is added, but the country is not being displayed. The problem obviously isn't the HTML table code, since the first row has the country cell populated, so something must be happening on the JavaScript side. Open `app.js` and add a breakpoint on line 58 where it says:

```
addPlace(city, country);
```

Restart the debug session and try to add a new place again. The execution should pause at the breakpoint you just set. Start hovering over the relevant variables. You can also add expressions to the watch panel, as seen below:



```
this: undefined
city: "Cape Town "
country: undefined
```

```
$('#country'): w.fn.init {}
$('#city'): w.fn.init(1) [input#city]
```

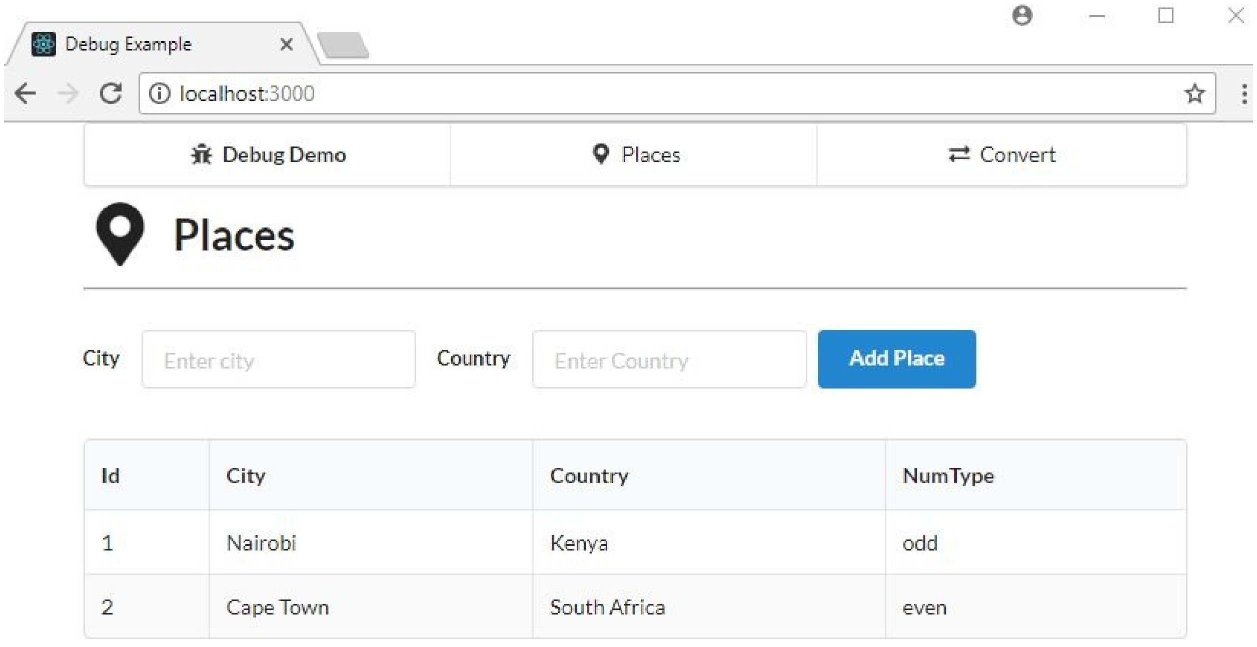
```
  el.html(html);
  // Form Validation Rules
  $('.ui.form').form({
    fields: {
      city: 'empty',
      country: 'empty',
    },
  });
  // Display Places Table
  const placesTable = $('#places-table');
  const tableHtml = placesTableTemplate({ places });
  placesTable.html(tableHtml);
  $('.submit').on('click', () => {
    const city undefined ).val();
    const country = $('#country').val();
    addPlace(city, country);
    placesTable.html(placesTableTemplate({ places }));
    $('form').form('clear');
    return false;
  });
});
```

As you can see, the `country` variable is undefined, but the `city` variable is. If you look at the jQuery selector expressions that have been set up in the watch panel, you'll notice that the `#country` selector returns nothing. This means it wasn't present in the DOM. Head over to `index.html` to verify.

Alas! If you look at line 59 where the country input has been defined, it's missing the ID attribute. You need to add one like this:

```
<input type="text" placeholder="Enter Country" name="country"
id="country">
```

Restart the debugging session and try to add a new place.

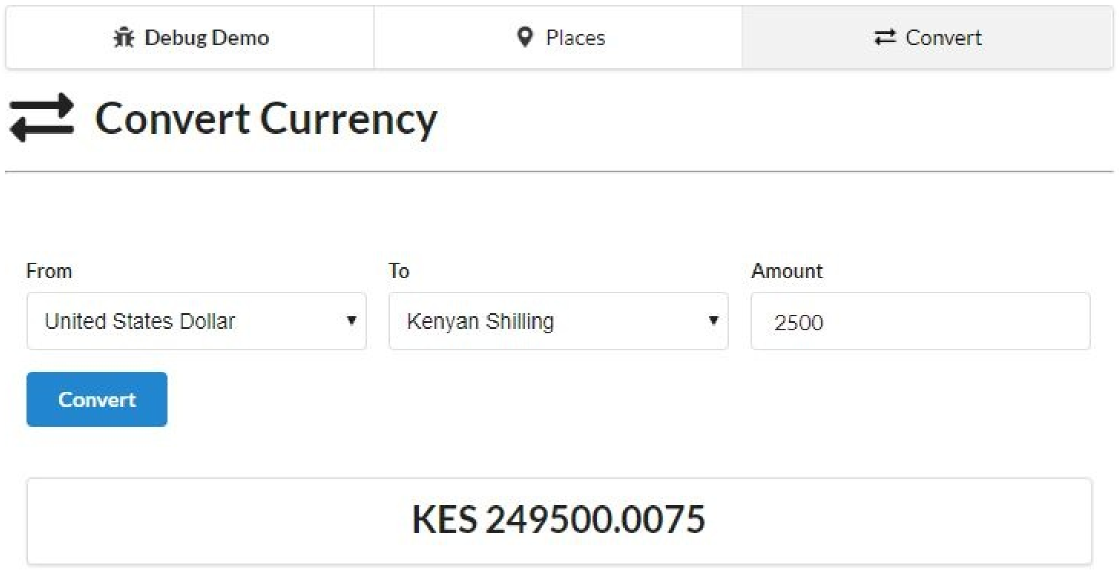


Id	City	Country	NumType
1	Nairobi	Kenya	odd
2	Cape Town	South Africa	even

It now works! Great job fixing another bug without `console.log`. Let's now move on to our final bug.

Debugging Clientside Routing

Click the *Convert* link in the navigation bar. You should be taken to this view to perform a quick conversion:



That runs fine. No bugs there.

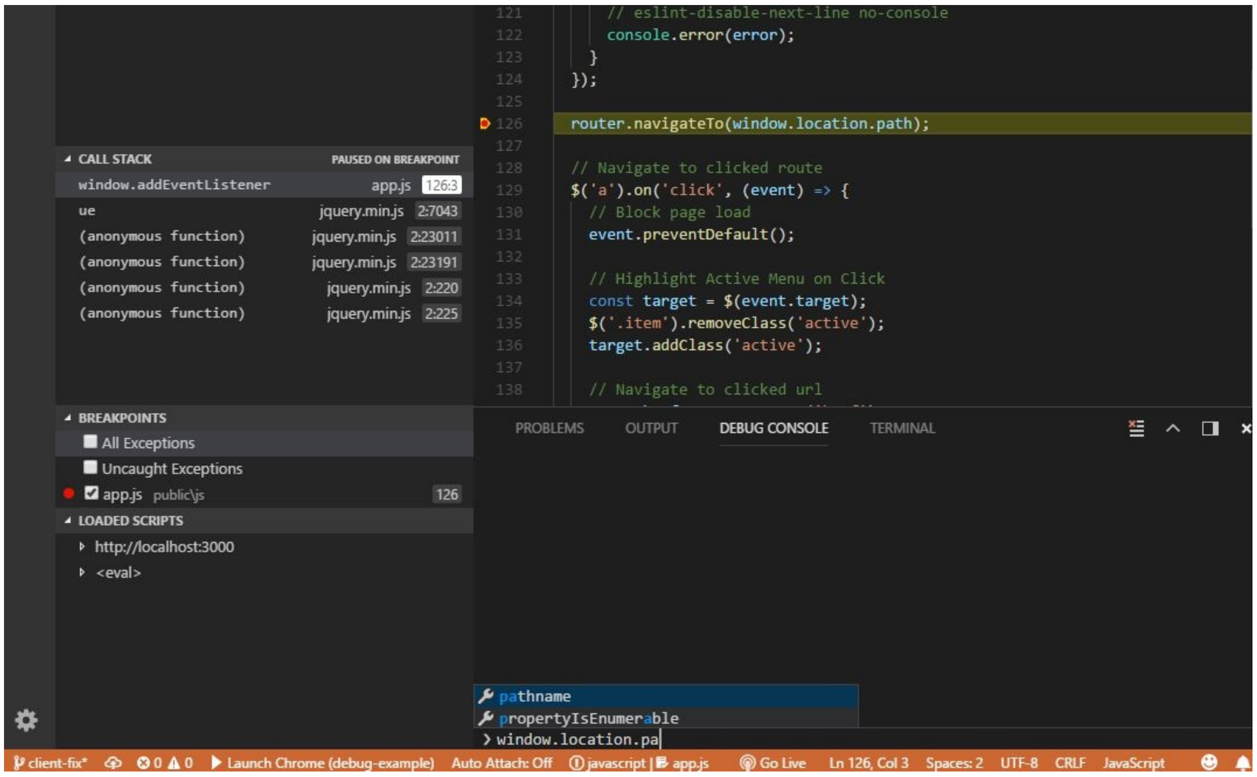
Actually there are, and they have nothing to do with the form. To spot them, refresh the page.

As soon as you hit reload, the user is navigated to back to `/`, the root of the app. This is clearly a routing problem which the Vanilla Router package is suppose to handle. Head back to `app.js` and look for this line:

```
router.navigateTo(window.location.path);
```

This piece of code is supposed to route users to the correct page based on the URL provided. But why isn't it working? Let's add a breakpoint here, then navigate back to the `/convert` URL and try refreshing the page again.

As soon as you refresh, the editor pauses at the breakpoint. Hover over the express `windows.location.path`. A popup appears which says the value is `undefined`. Let's go to the debug console and start typing the expression below:



```
CALL STACK                                   PAUSED ON BREAKPOINT
window.addEventListener          app.js      126:3
ue                               jquery.min.js  2:7043
(anonymous function)             jquery.min.js  2:23011
(anonymous function)             jquery.min.js  2:23191
(anonymous function)             jquery.min.js  2:220
(anonymous function)             jquery.min.js  2:225

BREAKPOINTS
☐ All Exceptions
☐ Uncaught Exceptions
● ☑ app.js  public\js                          126
LOADED SCRIPTS
  http://localhost:3000
  <eval>
```

```
      // eslint-disable-next-line no-console
      console.error(error);
    }
  });

router.navigateTo(window.location.path);

// Navigate to clicked route
$('a').on('click', (event) => {
  // Block page load
  event.preventDefault();

  // Highlight Active Menu on Click
  const target = $(event.target);
  $('.item').removeClass('active');
  target.addClass('active');

  // Navigate to clicked url
```

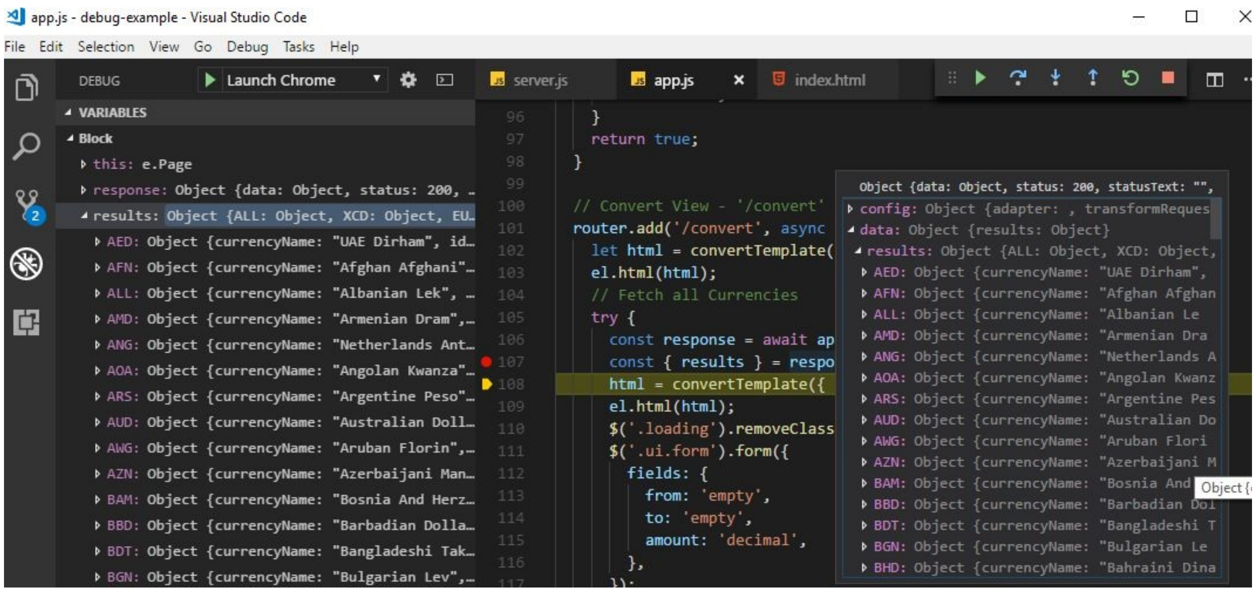
```
pathname
propertyIsEnumerable
> window.location.pa
```

```
client-fix*   ⊗ 0 ⚠ 0   Launch Chrome (debug-example)   Auto Attach: Off   javascript | app.js   Go Live   Ln 126, Col 3   Spaces: 2   UTF-8   CRLF   JavaScript
```

Hold up! The debug console just gave us the correct expression. It's supposed to read `window.location.pathname`. Correct the line of code, remove the breakpoint and restart the debugging session.

Navigate to the `/convert` URL and refresh. The page should reload the correct path. Awesome!

That's the last bug we're going to squash, but I do recommend you keep on experimenting within the debug session. Set up new breakpoints in order to inspect other variables. For example, check out the `response` object in the `router('/convert')` function. This demonstrates how you can use a debug session to figure out the data structure returned by an API request when dealing with new REST endpoints.


app.js - debug-example - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG
Launch Chrome
server.js
app.js
index.html

```
VARIABLES
 Block
   this: e.Page
   response: Object {data: Object, status: 200, …
   results: Object {ALL: Object, XCD: Object, EU…
     AED: Object {currencyName: "UAE Dirham", id…
     AFN: Object {currencyName: "Afghan Afghani"…
     ALL: Object {currencyName: "Albanian Lek", …
     AMD: Object {currencyName: "Armenian Dram",…
     ANG: Object {currencyName: "Netherlands Ant…
     AOA: Object {currencyName: "Angolan Kwanza"…
     ARS: Object {currencyName: "Argentine Peso"…
     AUD: Object {currencyName: "Australian Doll…
     AWG: Object {currencyName: "Aruban Florin",…
     AZN: Object {currencyName: "Azerbaijani Man…
     BAM: Object {currencyName: "Bosnia And Herz…
     BBD: Object {currencyName: "Barbadian Dolla…
     BDT: Object {currencyName: "Bangladeshi Tak…
     BGN: Object {currencyName: "Bulgarian Lev",…
```

```
    }
    return true;
  }

  // Convert View - '/convert'
  router.add('/convert', async
    let html = convertTemplate(
    el.html(html);
    // Fetch all Currencies
    try {
      const response = await ap
      const { results } = respo
      html = convertTemplate({
      el.html(html);
      $('.loading').removeClass
      $('.ui.form').form({
        fields: {
          from: 'empty',
          to: 'empty',
          amount: 'decimal',
        },
```

```
Object {data: Object, status: 200, statusText: "",
 config: Object {adapter: , transformReques
 data: Object {results: Object}
  results: Object {ALL: Object, XCD: Object,
   AED: Object {currencyName: "UAE Dirham",
   AFN: Object {currencyName: "Afghan Afghan
   ALL: Object {currencyName: "Albanian Le
   AMD: Object {currencyName: "Armenian Dra
   ANG: Object {currencyName: "Netherlands A
   AOA: Object {currencyName: "Angolan Kwanz
   ARS: Object {currencyName: "Argentine Pes
   AUD: Object {currencyName: "Australian Do
   AWG: Object {currencyName: "Aruban Flori
   AZN: Object {currencyName: "Azerbaijani M
   BAM: Object {currencyName: "Bosnia And
   BBD: Object {currencyName: "Barbadian Dol
   BDT: Object {currencyName: "Bangladeshi T
   BGN: Object {currencyName: "Bulgarian Le
   BHD: Object {currencyName: "Bahraini Dina
```

Summary

Now that we’ve come to the end of this tutorial, you should be proud of yourself for learning a vital skill in programming. Learning how to debug code properly will help you fix errors faster. You should be aware, however, that this article only scratches the surface of what’s possible, and you should take a look at the complete debugging documentation for VS Code. Here you'll find more details about specific commands and also types of breakpoint we haven't covered, such as Conditional Breakpoints.

I hope from now on you'll stop using `console.log` to debug and instead reach for VS Code to start debugging JavaScript like a pro!

Chapter 6: Introducing Axios, a Popular, Promise-based HTTP Client

by Nilson Jacques

Axios is a popular, promise-based HTTP client that sports an easy-to-use API and can be used in both the browser and Node.js.

Making HTTP requests to fetch or save data is one of the most common tasks a client-side JavaScript application will need to do. Third-party libraries — especially jQuery — have long been a popular way to interact with the more verbose browser APIs, and abstract away any cross-browser differences.

As people move away from jQuery in favor of improved native DOM APIs, or front-end UI libraries like React and Vue.js, including it purely for its `$.ajax` functionality makes less sense.

Let’s take a look at how to get started using Axios in your code, and see some of the features that contribute to its popularity among JavaScript developers.

Axios vs Fetch

As you’re probably aware, modern browsers ship with the newer Fetch API built in, so why not just use that? There are several differences between the two that many feel gives Axios the edge.

One such difference is in how the two libraries treat HTTP error codes. When using Fetch, if the server returns a 4xx or 5xx series error, your `catch()` callback won’t be triggered and it is down to the developer to check the response status code to determine if the request was successful. Axios, on the other hand, will reject the request promise if one of these status codes is returned.

Another small difference, which often trips up developers new to the API, is that Fetch doesn’t automatically send cookies back to the server when making a request. It’s necessary to explicitly pass an option for them to be included. Axios has your back here.

One difference that may end up being a show-stopper for some is progress updates on uploads/downloads. As Axios is built on top of the older XHR API, you’re able to register callback functions for `onUploadProgress` and `onDownloadProgress` to display the percentage complete in your app’s UI. Currently, Fetch has no support for doing this.

Lastly, Axios can be used in both the browser and Node.js. This facilitates sharing JavaScript code between the browser and the back end or doing server-side rendering of your front-end apps.

Fetch API for Node

there are versions of the Fetch API available for Node but, in my opinion, the other features Axios provides give it the edge.

Installing

As you might expect, the most common way to install Axios is via the npm package manager: `npm i axios`

and include it in your code where needed:

```
// ES2015 style import
import axios from 'axios';

// Node.js style require
const axios = require('axios');
```

If you’re not using some kind of module bundler (e.g. webpack), then you can always pull in the library from a CDN in the traditional way: `<script src="https://unpkg.com/axios/dist/axios.min.js"></script>`

Browser support

Axios works in all modern web browsers, and Internet Explorer 8+.

Making Requests

Similar to jQuery's `$.ajax` function, you can make any kind of HTTP request by passing an options object to Axios:

```
axios({
  method: 'post',
  url: '/login',
  data: {
    user: 'brunos',
    lastName: 'ilovenodejs'
  }
});
```

Here, we're telling Axios which HTTP method we'd like to use (e.g. GET/POST/DELETE etc.) and which URL the request should be made to.

We're also providing some data to be sent along with the request in the form of a simple JavaScript object of key/value pairs. By default, Axios will serialize this as JSON and send it as the request body.

Request Options

There are a whole bunch of additional options you can pass when making a request, but here are the most common ones:

- `baseUrl`: if you specify a base URL, it'll be prepended to any relative URL you use.
- `headers`: an object of key/value pairs to be sent as headers.
- `params`: an object of key/value pairs that will be serialized and appended to the URL as a query string.
- `responseType`: if you're expecting a response in a format other than JSON, you can set this property to `arraybuffer`, `blob`, `document`, `text`, or `stream`.
- `auth`: passing an object with `username` and `password` fields will use these credentials for HTTP Basic auth on the request.

Convenience methods

Also like jQuery, there are shortcut methods for performing different types of request.

The `get`, `delete`, `head` and `options` methods all take two arguments: a URL, and an optional config object.

```
axios.get('products5');
```

The `post`, `put`, and `patch` methods take a data object as their second argument, and an optional config object as the third:

```
axios.post(
  '/products',
  { name: 'Waffle Iron', price: 21.50 },
  { options }
);
```

Receiving a Response

Once you make a request, Axios returns a promise that will resolve to either a response object or an error object.

```
axios.get('product9')


  .then(response => console.log(response))


  .catch(error => console.log(error));
```

The response object

When the request is successful, your `then()` callback will receive a response object with the following properties:

- `data`: the payload returned from the server. By default, Axios expects JSON and will parse this back into a JavaScript object for you.
- `status`: the HTTP code returned from the server.
- `statusText`: the HTTP status message returned by the server.
- `headers`: all the headers sent back by the server.
- `config`: the original request configuration.
- `request`: the actual `XMLHttpRequest` object (when running in a browser).

The error object

If there’s a problem with the request, the promise will be rejected with an error object containing at least some of the following properties:

- `message`: the error message text.
- `response`: the response object (if received) as described in the previous section.
- `request`: the actual `XMLHttpRequest` object (when running in a browser).
- `config`: the original request configuration.

Transforms and Interceptors

Axios provides a couple of neat features inspired by Angular’s $http library. Although they appear similar, they have slightly different use cases.

Transforms

Axios allows you to provide functions to transform the outgoing or incoming data, in the form of two configuration options you can set when making a request: `transformRequest` and `transformResponse`. Both properties are arrays, allowing you to chain multiple functions that the data will be passed through.

Any functions passed to `transformRequest` are applied to PUT, POST and PATCH requests. They receive the request data, and the headers object as arguments and must return a modified data object

```
const options = {
  transformRequest: [
    (data, headers) => {
      // do something with data
      return data;
    }
  ]
}
```

Functions can be added to `transformResponse` in the same way, but are called only with the response data, and before the response is passed to any chained `then()` callbacks.

So what could we use transforms for? One potential use case is dealing with an API that expects data in a particular format — say XML or even CSV. You could set up a pair of transforms to convert outgoing and incoming data to and from the format the API requires.

It’s worth noting that Axios’ default `transformRequest` and `transformResponse` functions transform data to and from JSON, and specifying your own transforms will override these.

Interceptors

While transforms let you modify outgoing and incoming data, Axios also allows you to add functions called interceptors. Like transforms, these functions can be attached to fire when a request is made, or when a response is received.

```
// Add a request interceptor
axios.interceptors.request.use((config) => {
    // Do something before request is sent
    return config;
  }, (error) => {
    // Do something with request error
    return Promise.reject(error);
  });

// Add a response interceptor
axios.interceptors.response.use((response) => {
    // Do something with response data
    return response;
  }, (error) => {
    // Do something with response error
    return Promise.reject(error);
  });
```

As you might have noticed from the examples above, interceptors have some important differences from transforms. Instead of just receiving the data or headers, interceptors receive the full request config or response object.

When creating interceptors, you can also choose to provide an error handler function that allows you to catch any errors and deal with them appropriately.

Request interceptors can be used to do things such as retrieve a token from local storage and send with all requests, while a response interceptor could be used to catch 401 responses and redirect to a login page for authorization.

Thirdparty Add-ons

Being a popular library, Axios benefits from an ecosystem of thirdparty libraries that extend its functionality. From interceptors to testing adaptors to loggers, there’s quite a variety available. Here are a few that I think you may find useful:

- axios-mock-adaptor: allows you to easily mock requests to facilitate testing your code.
- axios-cache-plugin: a wrapper for selectively caching GET requests.
- redux-axios-middleware: if you’re a Redux user, this middleware provides a neat way to dispatch Ajax requests with plain old actions.

A list of more Axios add-ons and extentions is available on the Axios GitHub repo.

In summary, Axios has a lot to recommend it. It has a straightforward API, with helpful shortcut methods that will be familiar to anyone who’s used jQuery before. Its popularity, and the availability of a variety of thirdparty add-ons, make it a solid choice for including in your apps, whether front end, back end, or both.