THE JAVASCRIPT DEVELOPER TOOLS

BY FLAVIO COPES

HTTPS://FLAVIOCOPES.COM

UPDATED MAY 19, 2018

Tooling

- Webpack
- Babel

Testing

- Jest

Package managers

- Yarn
- npm

Editor tools

- ESLint
- Prettier

Version control and collaboration

- Git
- A Git cheat sheet
- GitHub

Debugging

- Browser DevTools
- Console API

Developer-friendly hosting and platforms

- Glitch
- Airtable

- Netlify
- Firebase Hosting

WEBPACK

Webpack is a tool that has got a lot of attention in the last few years, and it is now seen used in almost every project. Learn about it.

- What is webpack?
- Installing webpack
    - Global install

    - Local install

- Webpack configuration
- The entry point
- The output
- Loaders
- Plugins
- The webpack mode
- Running webpack
- Watching changes
- Handling images
- Process your SASS code and transform it to CSS
- Generate Source Maps

What is webpack?

Webpack is a tool that lets you compile JavaScript modules, also known as **module bundler**.

Given a large number of files, it generates a single file (or a few files) that run your app.

It can perform many operations:

- helps you bundle your resources.
- watches for changes and re-runs the tasks.
- can run Babel transpilation to ES5, allowing you to use the latest JavaScript features without worrying about browser support.
- can transpile CoffeeScript to JavaScript

- can convert inline images to data URIs.
- allows you to use require() for CSS files.
- can run a development webserver.
- can handle hot module replacement.
- can split the output files into multiple files, to avoid having a huge js file to load in the first page hit.
- can perform tree shaking.

Webpack is not limited to be used on the frontend, but it's also useful in backend Node.js development as well.

Predecessors of webpack, and still widely used tools, include:

- Grunt
- Broccoli
- Gulp

There are lots of similarities in what those and Webpack can do, but the main difference is that those are known as **task runners**, while webpack was born as a module bundler.

It's a more focused tool: you specify an entry point to your app (it could even be an HTML file with script tags) and webpack analyzes the files and bundles in a single JavaScript output file all you need to run the app.

Installing webpack

Webpack can be installed globally or locally for each project.

Global install

Here's how to install it globally with Yarn:

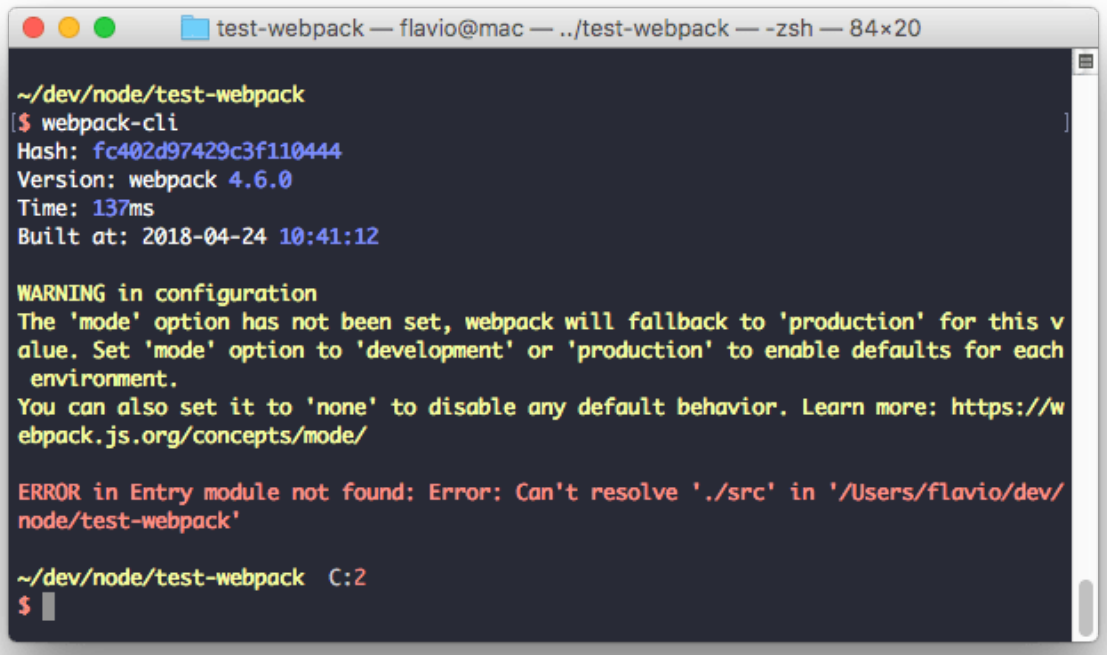
```
yarn global add webpack webpack-cli
```

with npm:

```
npm i -g webpack webpack-cli
```

once this is done, you should be able to run

```
webpack-cli
```



Local install

Webpack can be installed locally as well. It's the recommended setup, because webpack can be updated per-project, and you have less resistance in using the latest features just for a small project rather than updating all the projects you have the use webpack.

With Yarn:

```
yarn add webpack webpack-cli -D
```

with npm:

```
npm i webpack webpack-cli --save-dev
```

Once this is done, add this to your `package.json` file:

```
{
  //...
  "scripts": {
    "build": "webpack"
  }
}
```

once this is done, you can run webpack by typing

```
yarn build
```

in the project root.

Webpack configuration

By default, webpack (starting from version 4) does not require any config if you respect these conventions:

- the **entry point** of your app is `./src/index.js`
- the output is put in `./dist/main.js`.
- Webpack works in production mode

You can customize every little bit of webpack of course, when you need. The webpack configuration is stored in the `webpack.config.js` file, in the project root folder.

The entry point

By default the entry point is `./src/index.js` This simple example uses the `./index.js` file as a starting point:

```
module.exports = {
  /*...*/
  entry: './index.js'
  /*...*/
}
```

The output

By default the output is generated in `./dist/main.js`. This example puts the output bundle into `app.js`:

```
module.exports = {
  /*...*/
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'app.js'
  }
  /*...*/
}
```

Loaders

Using webpack allows you to use `import` or `require` statements in your JavaScript code to not just include other JavaScript, but any kind of file, for example CSS.

Webpack aims to handle all our dependencies, not just JavaScript, and loaders are one way to do that.

For example, in your code you can use:

```
import 'style.css'
```

by using this loader configuration:

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

The regular expression target any CSS file.

A loader can have options:

```
module.exports = {
  /*...*/
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: 'css-loader',
            options: {
              modules: true
            }
          }
        ]
      }
    ]
  }
  /*...*/
}
```

You can require multiple loaders per each rule:

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

```
      }
    ]
  }
  /*...*/
}
```

In this example, `css-loader` interprets the `import 'style.css'` directive in the CSS. `style-loader` is then responsible for injecting that CSS in the DOM, using a `<style>` tag.

The order matters, and it's reversed (the last is executed first).

What kind of loaders are there? Many!
 You can find the full list here

.

A commonly used loader is Babel, which is used to transpile modern JavaScript to ES5 code:

```
module.exports = {
  /*...*/
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  }
```

```
  /*...*/
}
```

This example makes Babel preprocess all our React/JSX files:

```
module.exports = {
  /*...*/
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: 'babel-loader'
      }
    ]
  },
  resolve: {
    extensions: [
      '.js',
      '.jsx'
    ]
  }
  /*...*/
}
```

See the `babel-loader` options here (https://webpack.js.org/loaders/babel-loader/) .

Plugins

Plugins are like loaders, but on steroids. They can do things that loaders can't do, and they are the main building block of webpack.

Take this example:

```
module.exports = {
  /*...*/
  plugins: [
    new HTMLWebpackPlugin()
  ]
  /*...*/
}
```

The `HTMLWebpackPlugin` plugin has the job of automatically creating an HTML file, add the output JS bundle path, so the JavaScript is ready to be served.

There are lots of plugins available (https://webpack.js.org/plugins/) .

One useful plugin, `CleanWebpackPlugin`, can be used to clear the `dist/` folder before creating any output, so you don't leave around files when you change the names of the output file:

```
module.exports = {
  /*...*/
  plugins: [
    new CleanWebpackPlugin(['dist']),
  ]
  /*...*/
}
```

The webpack mode

The mode (introduced in webpack 4) sets the environment on which webpack works. It can be set to `development` or `production` (defaults to production, so you only set it when moving to development)

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

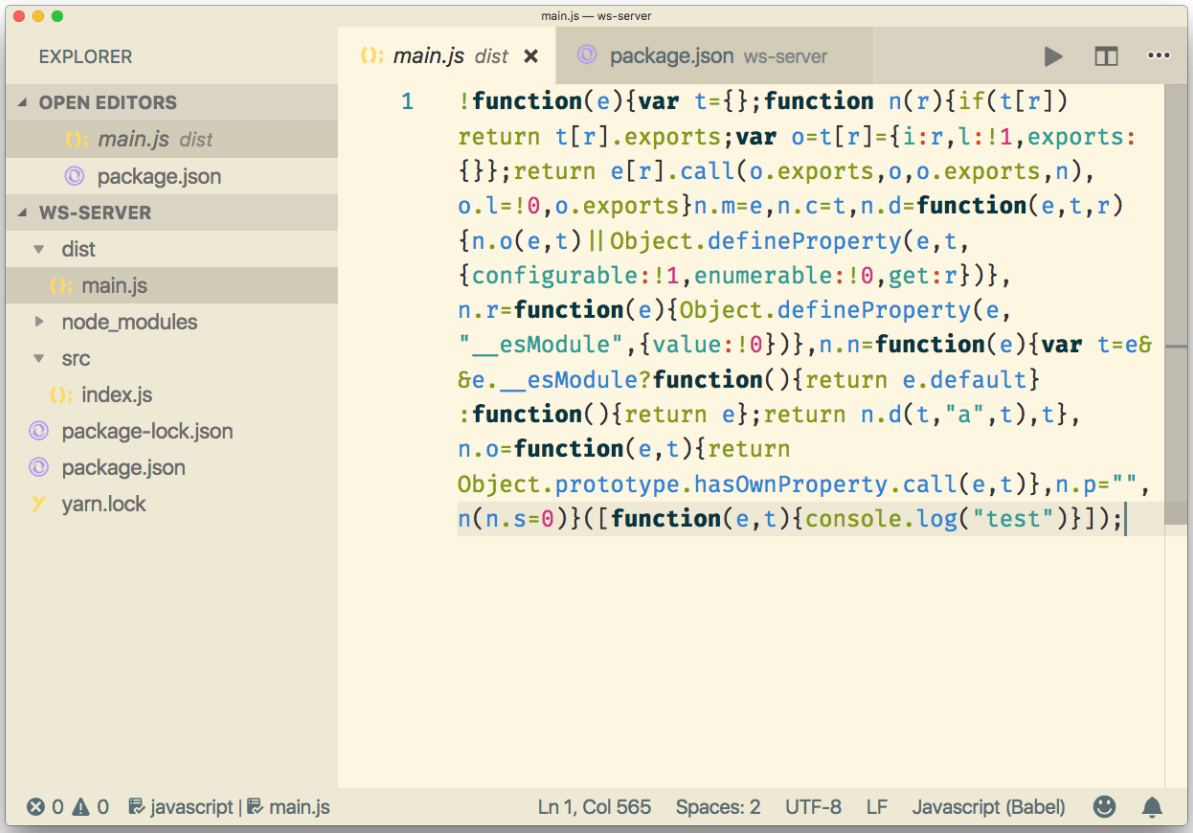
Development mode:

- builds very fast
- is less optimized than production
- does not remove comments
- provides more detailed error messages and suggestions
- provides a better debugging experience

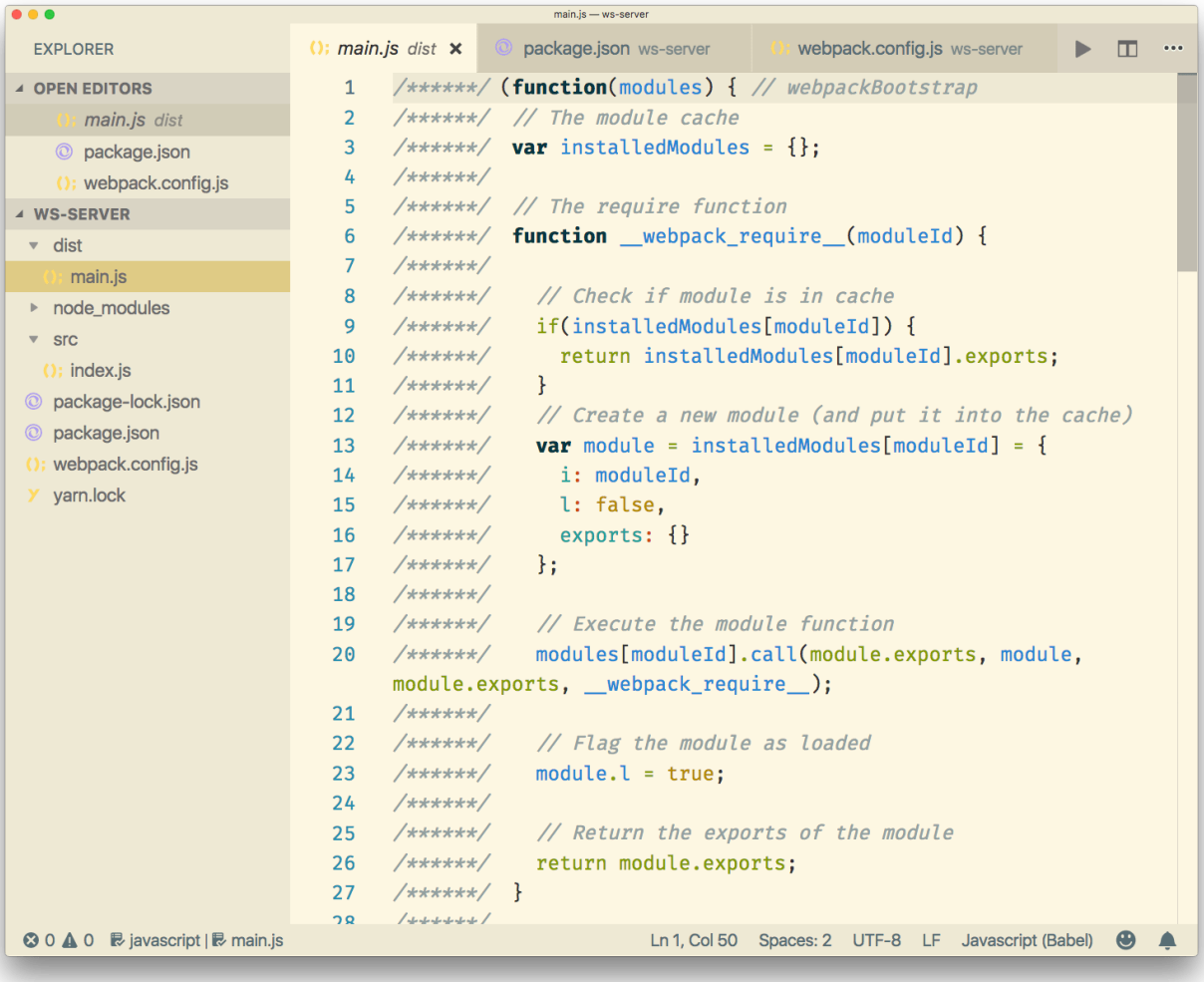
Production mode is slower to build, since it needs to generate a more optimized bundle. The resulting JavaScript file is smaller in size, as it removes many things that are not needed in production.

I made a sample app that just prints a `console.log` statement.

Here's the production bundle:



Here's the development bundle:



Running webpack

Webpack can be run from the command line manually if installed globally, but generally you write a script inside the `package.json` file, which is then run using `npm` or `yarn`.

For example this `package.json` scripts definition we used before:

```
"scripts": {
  "build": "webpack"
}
```

allows us to run `webpack` by running

```
npm run build
```

or

```
yarn run build
```

or simply

```
yarn build
```

Watching changes

Webpack can automatically rebuild the bundle when a change in your app happens, and keep listening for the next change.

Just add this script:

```
"scripts": {
  "watch": "webpack --watch"
}
```

and run

```
npm run watch
```

or

```
yarn run watch
```

or simply

```
yarn watch
```

One nice feature of the watch mode is that the bundle is only changed if the building has no errors. If there are errors, `watch` will keep listening for changes, and try to rebuild the bundle, but the current, working bundle is not affected by those problematic builds.

Handling images

Webpack allows to use images in a very convenient way, using the `file-loader` (https://webpack.js.org/loaders/file-loader/) loader.

This simple configuration:

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

Allows you to import images in your JavaScript:

```
import Icon from './icon.png'

const img = new Image()
img.src = Icon
element.appendChild(img)
```

(`img` is an HTMLImageElement. Check the Image docs (https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement/Image))

`file-loader` can handle other asset types as well, like fonts, CSV files, xml, and more.

Another nice tool to work with images is the `url-loader` loader.

This example loads any PNG file smaller than 8KB as a data URL.

```
module.exports = {
  /*...*/
  module: {
    rules: [
      {
        test: /\.png$/,
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192
            }
          }
        ]
      }
    ]
  }
```

```
  /*...*/
}
```

Process your SASS code and transform it to CSS

Using `sass-loader`, `css-loader` and `style-loader`:

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

Generate Source Maps

Since webpack bundles the code, Source Maps are mandatory to get a reference to the original file that raised an error, for example.

You tell webpack to generate source maps using the `devtool` property of the configuration:

```
module.exports = {
  /*...*/
  devtool: 'inline-source-map',
  /*...*/
}
```

`devtool` has many possible values (https://webpack.js.org/configuration/devtool/) , the most used probably are:

- `none` : adds no source maps
- `source-map` : ideal for production, provides a separate source map that can be minimized, and adds a reference into the bundle, so development tools know that the source map is available. Of course you should configure the server to avoid shipping this, and just use it for debugging purposes
- `inline-source-map` : ideal for development, inlines the source map as a Data URL

BABEL

Babel is an awesome entry in the Web Developer toolset. It's an awesome tool, and it’s been around for quite some time, but nowadays almost every JavaScript developer relies on it, and this will continue going on, because Babel is now indispensable and has solved a big problem for everyone.

- Introduction to Babel
- Installing Babel
- An example Babel configuration
- Babel presets
    - `es2015` preset
    - `env` preset
    - `react` preset
    - More info on presets
- Using Babel with webpack

> *This article covers Babel 6, the current stable release*

Introduction to Babel

Babel is an awesome tool, and it's been around for quite some time, but nowadays almost every JavaScript developer relies on it, and this will continue going on, because Babel is now indispensable and has solved a big problem for everyone.

Which problem?

The problem that every Web Developer has surely had: a feature of JavaScript is available in the latest release of a browser, but not in the older versions. Or maybe Chrome or Firefox implement it, but Safari iOS and Edge do not.

For example, ES2015 introduced the **arrow function**:

```
[1, 2, 3].map((n) => n + 1)
```

Which is now supported by all modern browsers. IE11 does not support it, nor Opera Mini (How do I know? By checking the ES6 Compatibility Table (http://kangax.github.io/compat-table/es6/#test-arrow_functions)).

So how should you deal with this problem? Should you move on and leave the customers with older/incompatible browsers behind, or should you write older JavaScript code to make all your users happy?

Enter Babel. Babel is a **compiler**: it takes code written in one standard, and it transpiles it to code written into another standard.

You can configure Babel to transpile modern ES2017 JavaScript into JavaScript ES5 syntax:

```
[1, 2, 3].map(function(n) {
  return n + 1
})
```

This must happen at build time, so you must setup a workflow that handles this for you. Webpack is a common solution.

(P.S. if all this *ES* thing sounds confusing to you, see more about ES versions in the ECMAScript guide)

Installing Babel

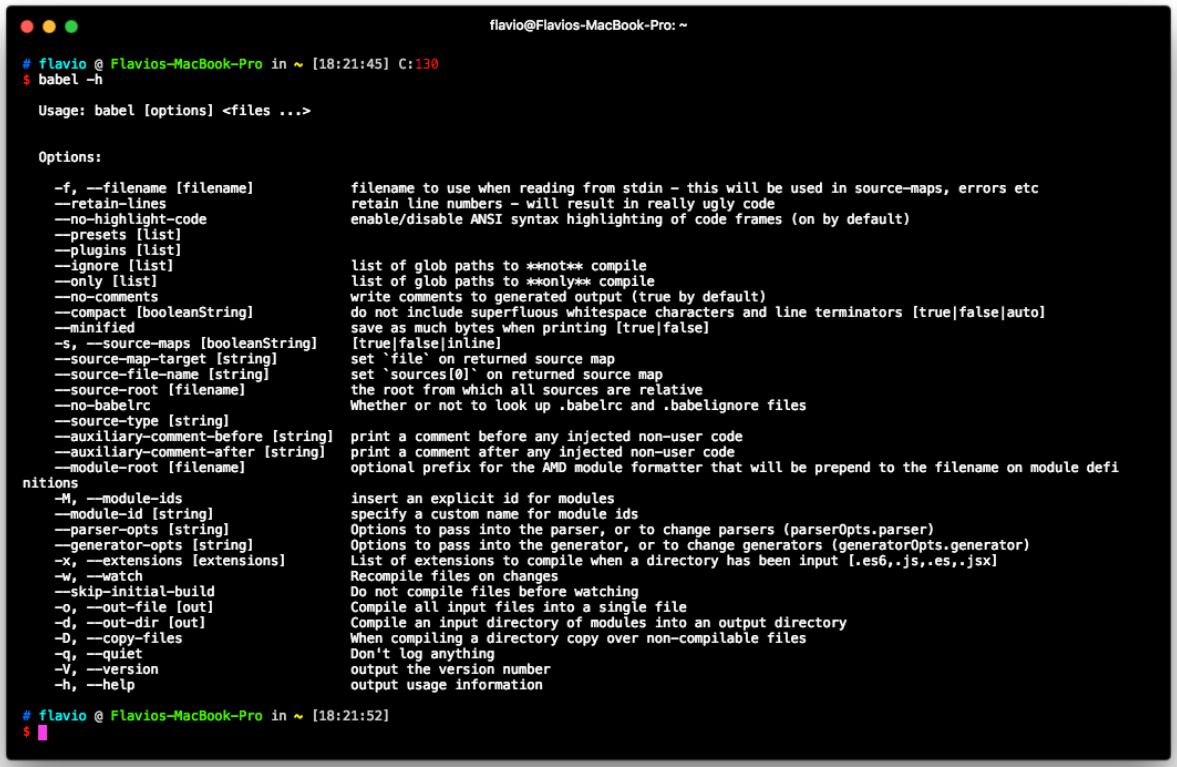
Babel is easily installed using npm or Yarn:

```
npm install --global babel-cli
```

or

```
yarn global add babel-cli
```

This will make the global `babel` command available in the command line:



Now inside your project install the babel-core and babel-loader packages, by running:

```
npm install babel-core babel-loader --save-dev
```

or

```
yarn add --dev babel-core babel-loader
```

By default Babel does not provide anything, it's just a blank box that you can fill with plugins to solve your specific needs.

An example Babel configuration

Babel out of the box does not do anything useful, you need to configure it.

To solve the problem we talked about in the introduction (using arrow functions in every browser), we can run

```
npm install --save-dev \
    babel-plugin-transform-es2015-arrow-functions
```

or (Yarn)

```
yarn add --dev \
     babel-plugin-transform-es2015-arrow-functions
```

to download the package in the `node_modules` folder of our app, then we need to add

```
{
  "plugins": ["transform-es2015-arrow-functions"]
}
```

to the `.babelrc` file present in the application root folder. If you don't have that file already, you just create a blank file, and put that content into it.

> *TIP: If you have never seen a dot file (a file starting with a dot) it might be odd at first because that file might not appear in your file manager, as it's a hidden file.*

Now if we have a `script.js` file with this content:

```
var a = () => {};
var a = (b) => b;

const double = [1,2,3].map((num) => num * 2);
console.log(double); // [2,4,6]

var bob = {
  _name: "Bob",
  _friends: ["Sally", "Tom"],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};
console.log(bob.printFriends());
```

running `babel script.js` will output the following code:

```
var a = function () {};var a = function (b) {
  return b;
};

const double = [1, 2, 3].map(function (num) {
  return num * 2;
});console.log(double); // [2,4,6]

var bob = {
  _name: "Bob",
  _friends: ["Sally", "Tom"],
  printFriends() {
    var _this = this;

    this._friends.forEach(function (f) {
      return console.log(_this._name + " knows " + f);
    });
  }
};
console.log(bob.printFriends());
```

As you can see arrow functions have all been converted to JavaScript ES5 `function`s.

Babel presets

We just saw in the previous article how Babel can be configured to transpile specific JavaScript features.

You can add much more plugins, but you can't add to the configuration features one by one, it's not practical.

This is why Babel offers **presets**.

The most popular presets are `es2015`, `env` and `react`.

`es2015` preset

This preset provides all the **ES2015** features. You install it by running

```
npm install --save-dev babel-preset-es2015
```

or

```
yarn add --dev babel-preset-es2015
```

and by adding

```
{
  "presets": ["es2015"]
}
```

to your `.babelrc` file.

`env` preset

The `env` preset is very nice: you tell it which environments you want to support, and it does everything for you, **supporting all modern JavaScript features**.

E.g. “support the last 2 versions of every browser, but for Safari let’s support all versions since Safari 7`

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

or “I don’t need browsers support, just let me work with Node.js 6.10”

```
{
  "presets": [
    ["env", {
      "targets": {
        "node": "6.10"
      }
```

```
    }]
  ]
}
```

`react` preset

The `react` preset is very convenient when writing React apps, by adding `preset-flow`, `syntax-jsx`, `transform-react-jsx`, `transform-react-display-name`.

By including it, you are all ready to go developing React apps, with JSX transforms and Flow support.

More info on presets

https://babeljs.io/docs/plugins/

Using Babel with webpack

If you want to run modern JavaScript in the browser, Babel on its own is not enough, you also need to bundle the code. Webpack is the perfect tool for this.

> *TIP: read the **webpack guide** if you're not familiar with webpack*

Modern JS needs two different stages: a compile stage, and a runtime stage. This is because some ES6+ features need a polyfill or a runtime helper.

To install the Babel polyfill runtime functionality, run

```
npm install --save babel-polyfill \
                   babel-runtime \
                   babel-plugin-transform-runtime
```

or

```
yarn add babel-polyfill \
         babel-runtime \
         babel-plugin-transform-runtime
```

Now in your `webpack.config.js` file add:

```
entry: [
  'babel-polyfill',
  // your app scripts should be here
],

module: {
  loaders: [
    // Babel loader compiles ES2015 into ES5 for
    // complete cross-browser support
    {
      loader: 'babel-loader',
      test: /\.js$/,
      // only include files present in the `src` subdirectory
      include: [path.resolve(__dirname, "src")],
      // exclude node_modules, equivalent to the above line
      exclude: /node_modules/,
      query: {
        // Use the default ES2015 preset
        // to include all ES2015 features
        presets: ['es2015'],
        plugins: ['transform-runtime']
      }
    }
  ]
}
```

By keeping the presets and plugins information inside the `webpack.config.js` file, we can avoid having a `.babelrc` file.

JEST


Jest

- Introduction to Jest
- Installation
- Create the first Jest test
- Run Jest with VS Code
- Matchers
- Setup
- Teardown
- Group tests using describe()
- Testing asynchronous code
    - Callbacks
    - Promises
    - Async/await
- Mocking
    - Spy packages without affecting the functions code
    - Mock an entire package
    - Mock a single function
    - Pre-built mocks
- Snapshot testing

Introduction to Jest

Jest is a library for testing JavaScript code.

It's an open source project maintained by Facebook, and it's especially well suited for React code testing, although not limited to that: it can test any JavaScript code. Its strengths are:

- it's fast
- it can perform **snapshot testing**
- it's opinionated, and provides everything out of the box without requiring you to make choices

Jest is a tool very similar to Mocha, although they have differences:

- Mocha is less opinionated, while Jest has a certain set of conventions
- Mocha requires more configuration, while Jest works usually out of the box, thanks to being opinionated
- Mocha is older and more established, with more tooling integrations

In my opinion the biggest feature of Jest is it's an out of the box solution that works without having to interact with other testing libraries to perform its job.

Installation

Jest is automatically installed in `create-react-app`, so if you use that, you don't need to install Jest.

Jest can be installed in any other project using Yarn:

```
yarn add --dev jest
```

or npm:

```
npm install --save-dev jest
```

notice how we instruct both to put Jest in the `devDependencies` part of the `package.json` file, so that it will only be installed in the development environment and not in production.

Add this line to the scripts part of your `package.json` file:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

so that tests can be run using `yarn test` or `npm run test`.

Alternatively, you can install Jest globally:

```
yarn global add jest
```

and run all your tests using the `jest` command line tool.

Create the first Jest test

Projects created with `create-react-app` have Jest installed and preconfigured out of the box, but adding Jest to any project is as easy as typing

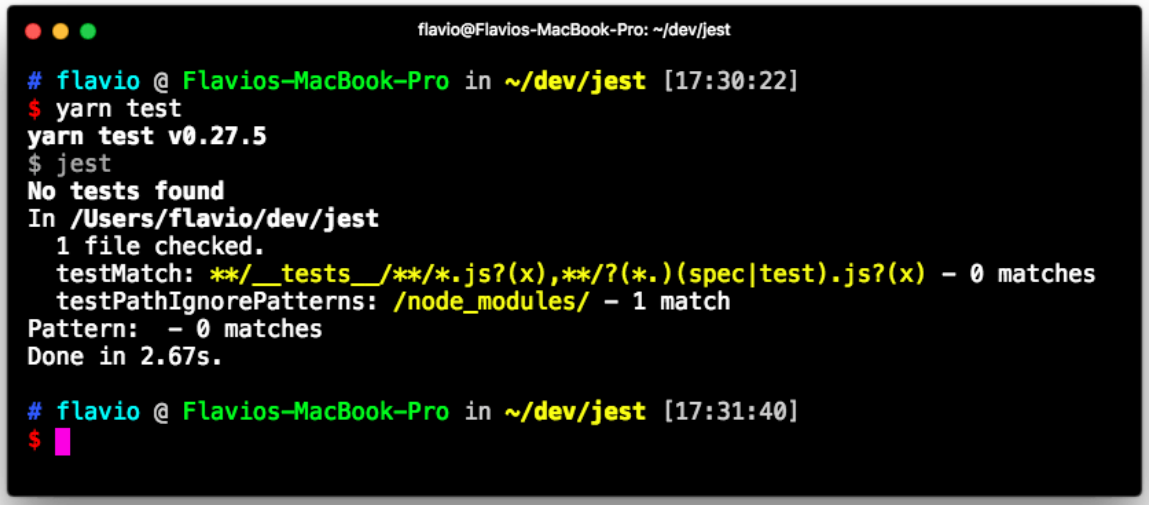
```
yarn add --dev jest
```

Add to your `package.json` this line:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

and run your tests by executing `yarn test` in your shell.

Now, you don't have any test here, so nothing is going to be executed:



Let's create the first test. Open a `math.js` file and type a couple functions that we'll later test:

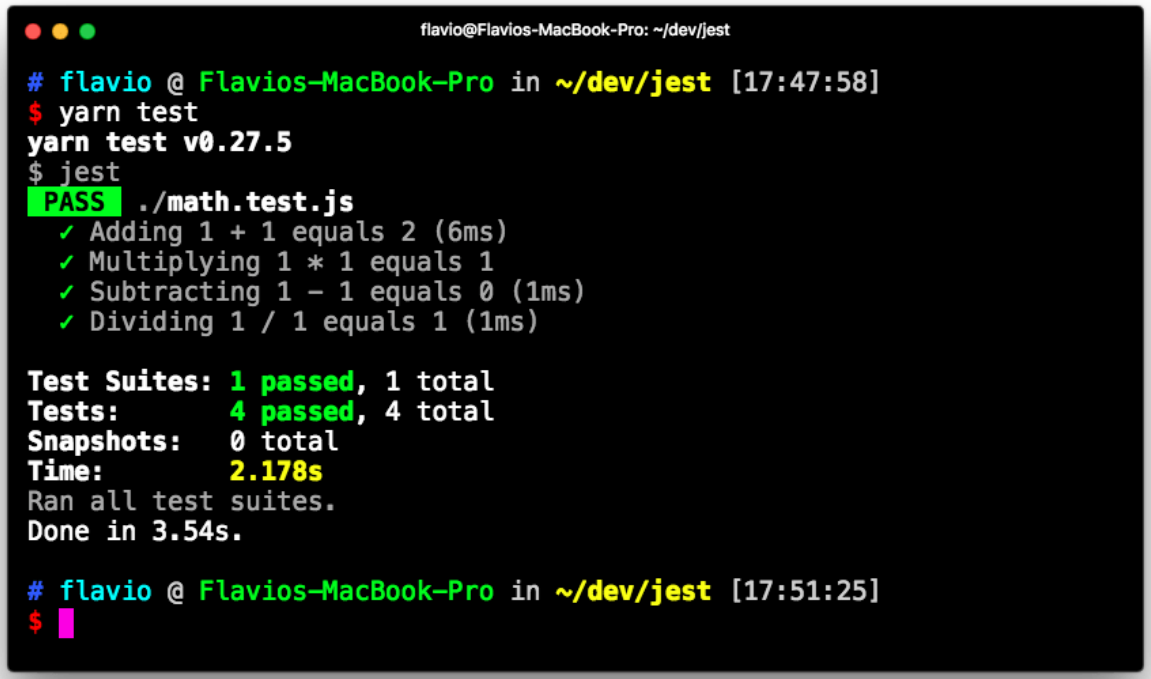
```
const sum = (a, b) => a + b
const mul = (a, b) => a * b
const sub = (a, b) => a - b
const div = (a, b) => a / b

export default { sum, mul, sub, div }
```

Now create a `math.test.js` file, in the same folder, and there we'll use Jest to test the functions defined in `math.js`:

```
const { sum, mul, sub, div } = require("./math")

test("Adding 1 + 1 equals 2", () => {
  expect(sum(1, 1)).toBe(2)
})
test("Multiplying 1 * 1 equals 1", () => {
  expect(mul(1, 1)).toBe(1)
})
test("Subtracting 1 - 1 equals 0", () => {
  expect(sub(1, 1)).toBe(0)
```

```
})
test("Dividing 1 / 1 equals 1", () => {
  expect(div(1, 1)).toBe(1)
})
```

Running `yarn test` results in Jest being run on all the test files it finds, and returning us the end result:

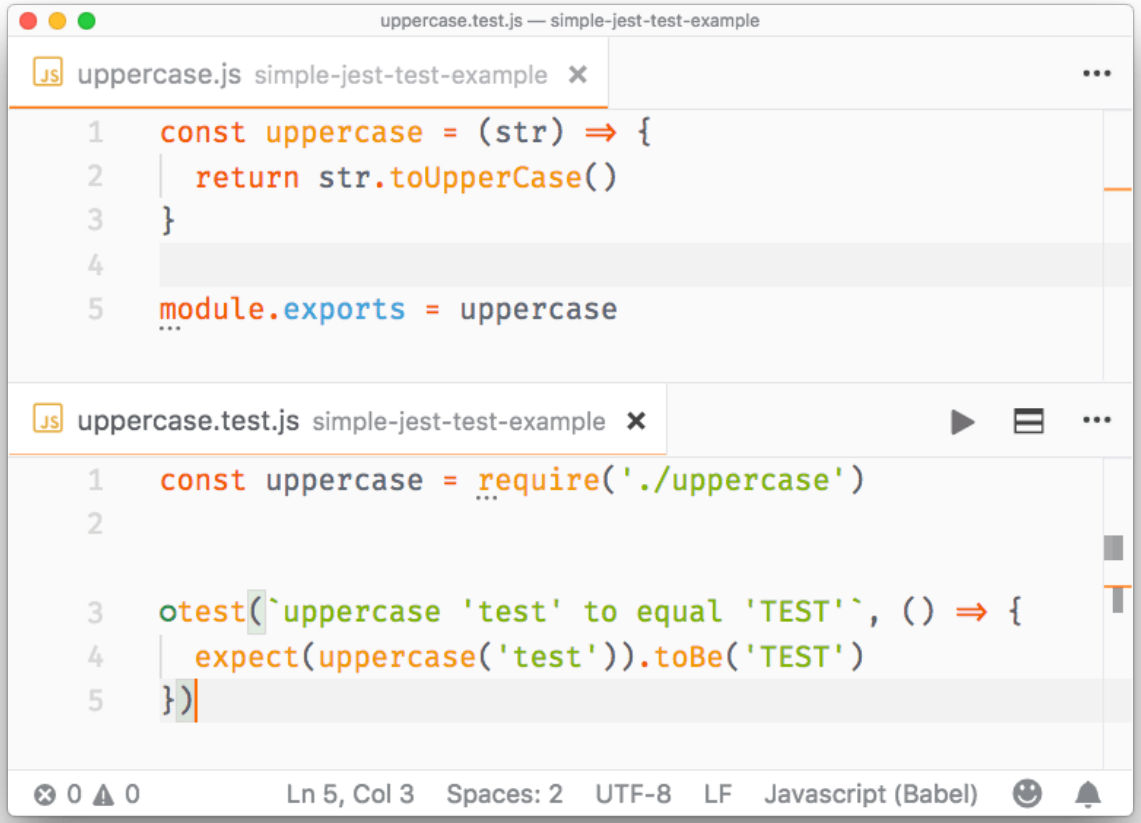


Run Jest with VS Code

Visual Studio Code is a great editor for JavaScript development. The Jest extension (https://marketplace.visualstudio.com/items?itemName=Orta.vscode-jest) offers a top notch integration for our tests.

Once you install it, it will automatically detect if you have installed Jest in your devDependencies and run the tests. You can also invoke the tests

manually by selecting the **Jest: Start Runner** command. It will run the tests and stay in watch mode to re-run them whenever you change one of the files that have a test (or a test file):



Matchers

In the previous article I used `toBe()` as the only **matcher**:

```
test("Adding 1 + 1 equals 2", () => {
  expect(sum(1, 1)).toBe(2)
})
```

A matcher is a method that lets you test values.

Most commonly used matchers, comparing the value of the result of `expect()` with the value passed in as argument, are:

- `toBe` compares strict equality, using `===`
- `toEqual` compares the values of two variables. If it's an object or array, checks equality of all the properties or elements
- `toBeNull` is true when passing a null value
- `toBeDefined` is true when passing a defined value (opposite as above)
- `toBeUndefined` is true when passing an undefined value
- `toBeCloseTo` is used to compare floating values, avoid rounding errors
- `toBeTruthy` true if the value is considered true (like an `if` does)
- `toBeFalsy` true if the value is considered false (like an `if` does)
- `toBeGreaterThan` true if the result of expect() is higher than the argument
- `toBeGreaterThanOrEqual` true if the result of expect() is equal to the argument, or higher than the argument
- `toBeLessThan` true if the result of expect() is lower than the argument
- `toBeLessThanOrEqual` true if the result of expect() is equal to the argument, or lower than the argument
- `toMatch` is used to compare strings with regular expression pattern matching
- `toContain` is used in arrays, true if the expected array contains the argument in its elements set
- `toHaveLength(number)` : checks the length of an array

- `toHaveProperty(key, value)` : checks if an object has a property, and optionally checks its value
- `toThrow` checks if a function you pass throws an exception (in general) or a specific exception
- `toBeInstanceOf()` : checks if an object is an instance of a class

All those matchers can be negated using `.not.` inside the statement, for example:

```
test("Adding 1 + 1 does not equal 3", () => {
  expect(sum(1, 1)).not.toBe(3)
})
```

For use with promises, you can use `.resolves` and `.rejects`:

```
expect(Promise.resolve('lemon')).resolves.toBe('lemon')

expect(Promise.reject(new
Error('octopus'))).rejects.toThrow('octopus')
```

Setup

Before running your tests you will want to perform some initialization.

To do something once before all the tests run, use the `beforeAll()` function:

```
beforeAll(() => {
  //do something
})
```

To perform something before each test runs, use `beforeEach()`:

```
beforeEach(() => {
  //do something
})
```

Teardown

Just as you could do with the setup, you can perform something after each test runs:

```
afterEach(() => {
  //do something
})
```

and after all tests end:

```
afterAll(() => {
  //do something
})
```

Group tests using describe()

You can create groups of tests, in a single file, that isolate the setup and teardown functions:

```
describe('first set', () => {
  beforeEach(() => {
    //do something
  })
```

```
  afterAll(() => {
    //do something
  })
  test(/*...*/)
  test(/*...*/)
}

describe('second set', () => {
  beforeEach(() => {
    //do something
  })
  beforeAll(() => {
    //do something
  })
  test(/*...*/)
  test(/*...*/)
}
```

Testing asynchronous code

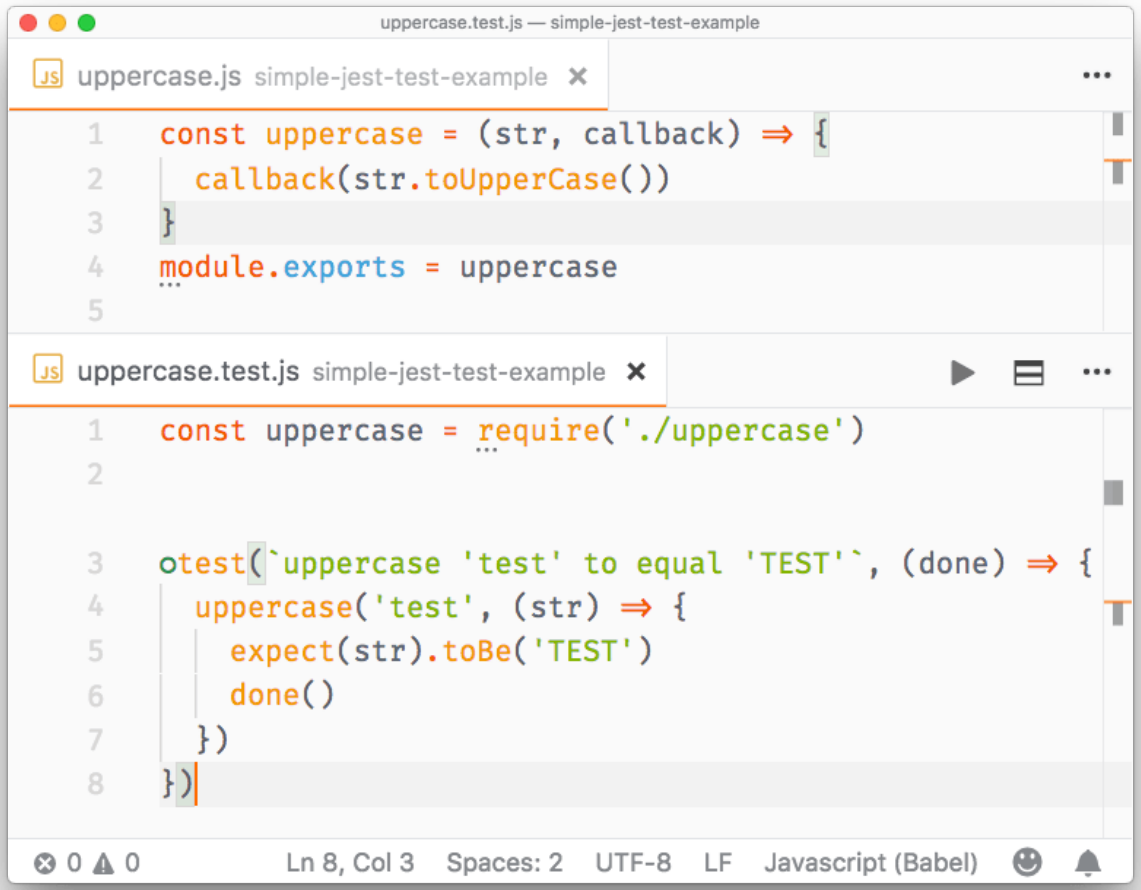
Asynchronous code in modern JavaScript can have basically 2 forms: callbacks and promises. On top of promises we can use async/await.

Callbacks

You can't have a test in a callback, because Jest won't execute it - the execution of the test file ends before the callback is called. To fix this, pass a parameter to the test function, which you can conveniently call `done`. Jest will wait until you call `done()` before ending that test:

```
//uppercase.js
function uppercase(str, callback) {
  callback(str.toUpperCase())
}
```

```
module.exports = uppercase

//uppercase.test.js
const uppercase = require('./src/uppercase')

test(`uppercase 'test' to equal 'TEST'`, (done) => {
  uppercase('test', (str) => {
    expect(str).toBe('TEST')
    done()
  }
})
```



Promises

With functions that return promises, we simply **return a promise** from the test:

```
//uppercase.js
const uppercase = (str) => {
  return new Promise((resolve, reject) => {
    if (!str) {
      reject('Empty string')
      return
    }
    resolve(str.toUpperCase())
  })
}
module.exports = uppercase

//uppercase.test.js
const uppercase = require('./uppercase')
test(`uppercase 'test' to equal 'TEST'`, () => {
  return uppercase('test').then(str => {
    expect(str).toBe('TEST')
  })
})
```

Promises that are rejected can be tested using `.catch()`:

```
//uppercase.js
const uppercase = (str) => {
  return new Promise((resolve, reject) => {
    if (!str) {
      reject('Empty string')
      return
    }
    resolve(str.toUpperCase())
  })
```

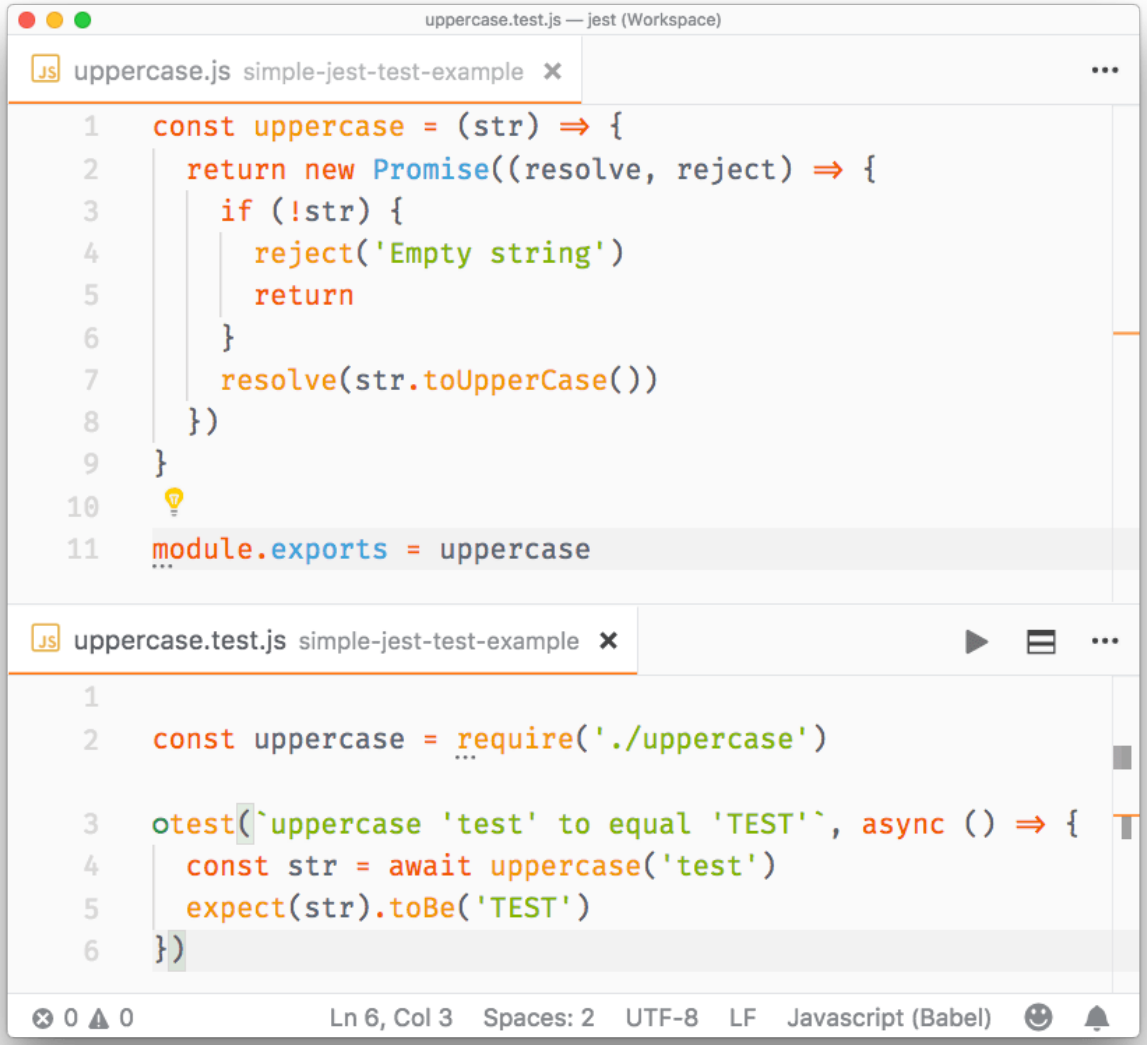
```
}

module.exports = uppercase

//uppercase.test.js
const uppercase = require('./uppercase')

test(`uppercase 'test' to equal 'TEST'`, () => {
  return uppercase('').catch(e => {
    expect(e).toMatch('Empty string')
  })
})
```

Async/await

To test functions that return promises we can also use async/await, which make the syntax very straightforward and simple:

```
//uppercase.test.js
const uppercase = require('./uppercase')
test(`uppercase 'test' to equal 'TEST'`, async () => {
```

```
  const str = await uppercase('test')
  expect(str).toBe('TEST')
})
```



Mocking

In testing, **mocking** allows you to test functionality that depends on:

- **Database**
- **Network** requests

- access to **Files**
- any **External** system

so that:

1. your tests run **faster**, giving a quick turnaround time during development
2. your tests are **independent** of network conditions, the state of the database
3. your tests do not **pollute** any data storage because they do not touch the database
4. any change done in a test does not change the state for subsequent tests, and re-running the test suite should start from a known and reproducible starting point
5. you don't have to worry about rate limiting on API calls and network requests

Mocking is useful when you want to avoid side effects (e.g. writing to a database) or you want to skip slow portions of code (like network access), and also avoids implications with running your tests multiple times (e.g. imagine a function that sends an email or calls a rate-limited API).

Even more important, if you are writing a **Unit Test**, you should test the functionality of a function in isolation, not with all its baggage of things it touches.

Using mocks, you can inspect if a module function has been called and which parameters were used, with:

- `expect().toHaveBeenCalled()` : check if a spied function has been called
- `expect().toHaveBeenCalledTimes()` : count how many times a spied function has been called
- `expect().toHaveBeenCalledWith()` : check if the function has been called with a specific set of parameters
- `expect().toHaveBeenLastCalledWith()` : check the parameters of the last time the function has been invoked

Spy packages without affecting the functions code

When you import a package, you can tell Jest to "spy" on the execution of a particular function, using `spyOn()` , without affecting how that method works.

Example:

```
const mathjs = require('mathjs')

test(`The mathjs log function`, () => {
  const spy = jest.spyOn(mathjs, 'log')
  const result = mathjs.log(10000, 10)

  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

Mock an entire package

Jest provides a convenient way to mock an entire package. Create a `__mocks__` folder in the project root, and in this folder create one JavaScript file for each of your packages.

Say you import `mathjs`. Create a `__mocks__/mathjs.js` file in your project root, and add this content:

```
module.exports = {
  log: jest.fn(() => 'test')
}
```

This will mock the log() function of the package. Add as many functions as you want to mock:

```
const mathjs = require('mathjs')

test(`The mathjs log function`, () => {
  const result = mathjs.log(10000, 10)
  expect(result).toBe('test')
  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

Mock a single function

More simply, you can mock a single function using `jest.fn()`:

```
const mathjs = require('mathjs')

mathjs.log = jest.fn(() => 'test')
test(`The mathjs log function`, () => {
  const result = mathjs.log(10000, 10)
  expect(result).toBe('test')
```

```
  expect(mathjs.log).toHaveBeenCalled()
  expect(mathjs.log).toHaveBeenCalledWith(10000, 10)
})
```

You can also use `jest.fn().mockReturnValue('test')` to create a simple mock that does nothing except returning a value.

Pre-built mocks

You can find pre-made mocks for popular libraries. For example this package https://github.com/jefflau/jest-fetch-mock allows you to mock `fetch()` calls, and provide sample return values without interacting with the actual server in your tests.

Snapshot testing

Snapshot testing is a pretty cool feature offered by Jest. It can memorize how your UI components are rendered, and compare it to the current test, raising an error if there's a mismatch.

This is a simple test on the App component of a simple `create-react-app` application (make sure you install `react-test-renderer`):

```
import React from 'react'
import App from './App'
import renderer from 'react-test-renderer'

it('renders correctly', () => {
  const tree = renderer
    .create(<App />)
    .toJSON()
```

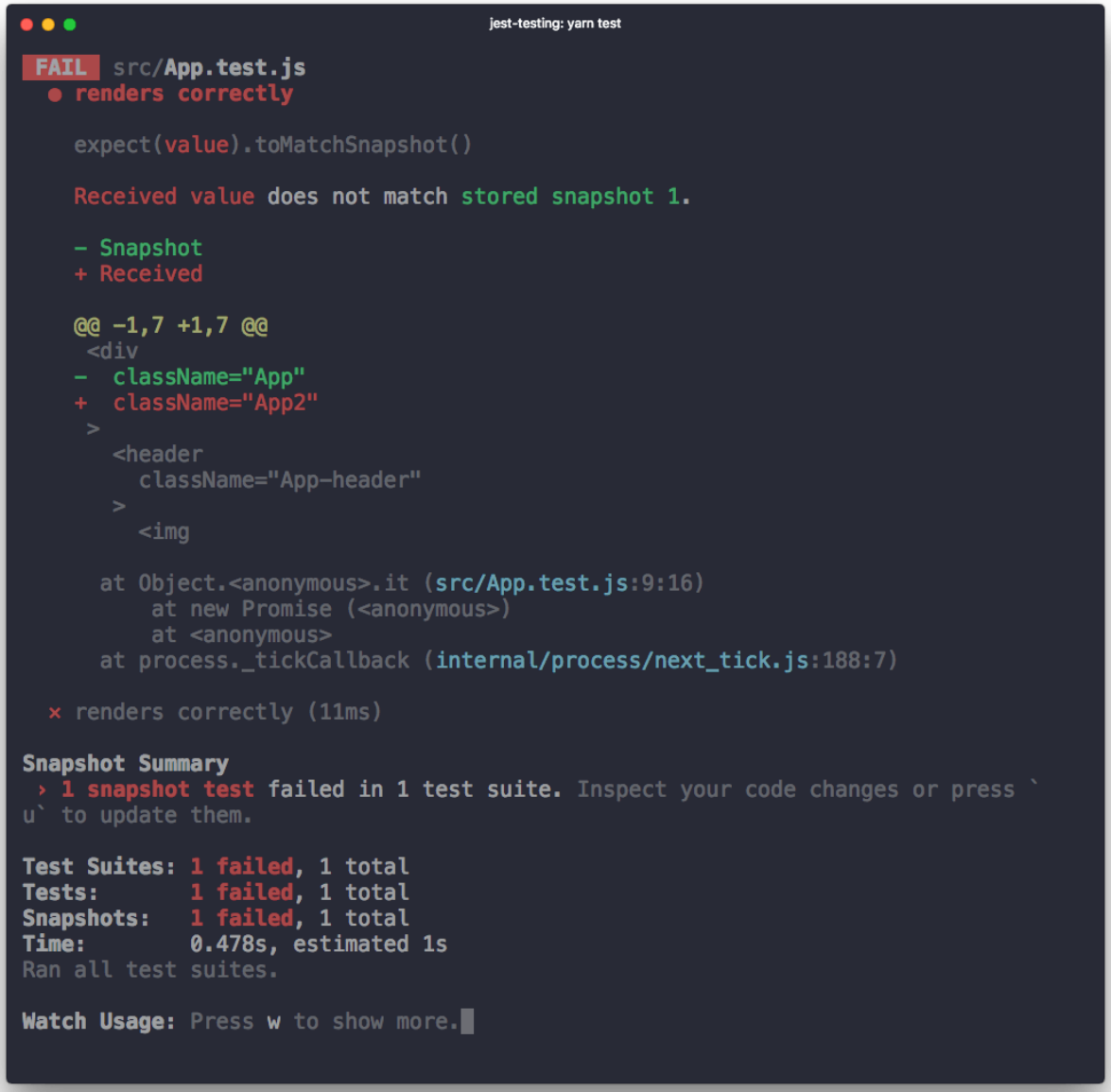
```
  expect(tree).toMatchSnapshot()
})
```

the first time you run this test, Jest saves the snapshot to the `__snapshots__` folder. Here's what App.test.js.snap contains:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`renders correctly 1`] = `
<div
  className="App"
>
  <header
    className="App-header"
  >
    <img
      alt="logo"
      className="App-logo"
      src="logo.svg"
    />
    <h1
      className="App-title"
    >
      Welcome to React
    </h1>
  </header>
  <p
    className="App-intro"
  >
    To get started, edit
    <code>
      src/App.js
    </code>
     and save to reload.
  </p>
</div>
`;
```

As you see it's the code that the App component renders, nothing more.

The next time the test compares the output of `<App />` to this. If App changes, you get an error:



When using `yarn test` in `create-react-app` you are in **watch mode**, and from there you can press `w` and show more options:

```
Watch Usage
 › Press u to update failing snapshots.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

If your change is intended, pressing `u` will update the failing snapshots, and make the test pass.

You can also update the snapshot by running `jest -u` (or `jest --updateSnapshot`) outside of watch mode.

YARN

Yarn is a JavaScript Package Manager, a direct competitor of npm, one of Facebook most popular Open Source projects

Intro to Yarn

Yarn is a JavaScript Package Manager, a direct competitor of npm, and it's one of Facebook most popular Open Source projects.

It's **compatible with npm packages**, so it has the great advantage of being a drop-in replacement for npm.

The reason you might want to use Yarn over npm are: - faster download of packages, which are installed in parallel - support for multiple registries - offline installation support

To me offline installation support seems like the *killer feature*, because once you have installed a package one time from the network, it gets cached and you can recreate a project from scratch without being connected (and without consuming a lot of your data, if you're on a mobile plan).

Since some projects could require a huge amount of dependencies, every time you run `npm install` to initialize a project you might download hundreds of megabytes from the network.

With Yarn, this is done just once.

This is not the only feature, many other goodies are provided by Yarn, which we'll see in this article.

In particular Yarn devotes a lot of care to security, by performing a checksum on every package it installs.

Tools eventually converge to a set of features that keeps them on the same level to stay relevant, so we'll likely see those features in npm in the future - competition is nice for us users.

Install Yarn

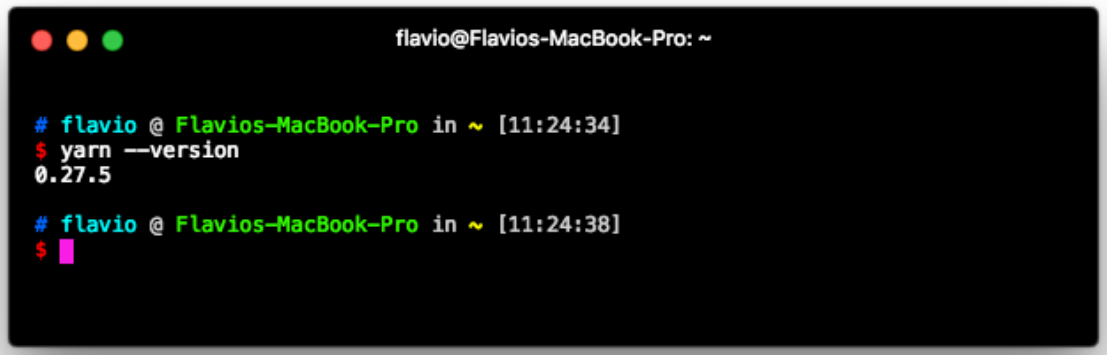
While there is a *joke* around about installing Yarn with npm (`npm install -g yarn`), it's not recommended by the Yarn team.

System-specific installation methods are listed at https://yarnpkg.com/en/docs/install. On MacOS for example you can use **Homebrew** and run

```
brew install yarn
```

but every Operating System has its own package manager of choice that will make the process very smooth.

In the end, you'll end up with the `yarn` command available in your shell:



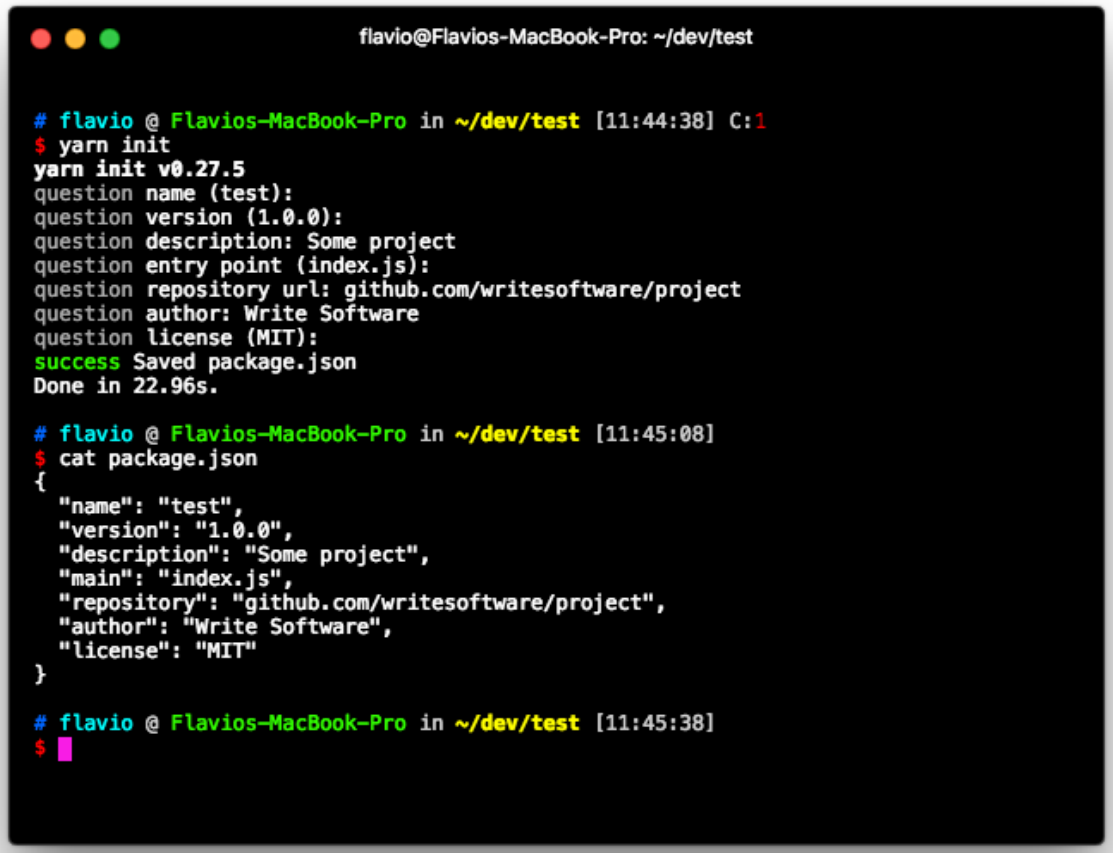
Managing packages

Yarn writes its dependencies to a file named `package.json`, which sits in the root folder of your project, and stores the dependencies files into the `node_modules` folder, *just like npm* if you used it in the past.

Initialize a new project

```
yarn init
```

starts an interactive prompt that helps you quick start a project:



Install the dependencies of an existing project

If you already have a `package.json` file with the list of dependencies but the packages have not been installed yet, run

```
yarn
```

or

```
yarn install
```

to start the installation process.

Install a package locally

Installing a package into a project is done using

```
yarn add package-name
```

> *This is equivalent to running `npm install --save package-name`, thus avoiding the invisible dependency issue when running `npm install package-name`, which does not add the dependency to the `package.json` file*

Install a package globally

```
yarn global add package-name
```

Install a package locally as a development dependency

```
yarn add --dev package-name
```

> *Equivalent to the `--save-dev` flag in npm*

Remove a package

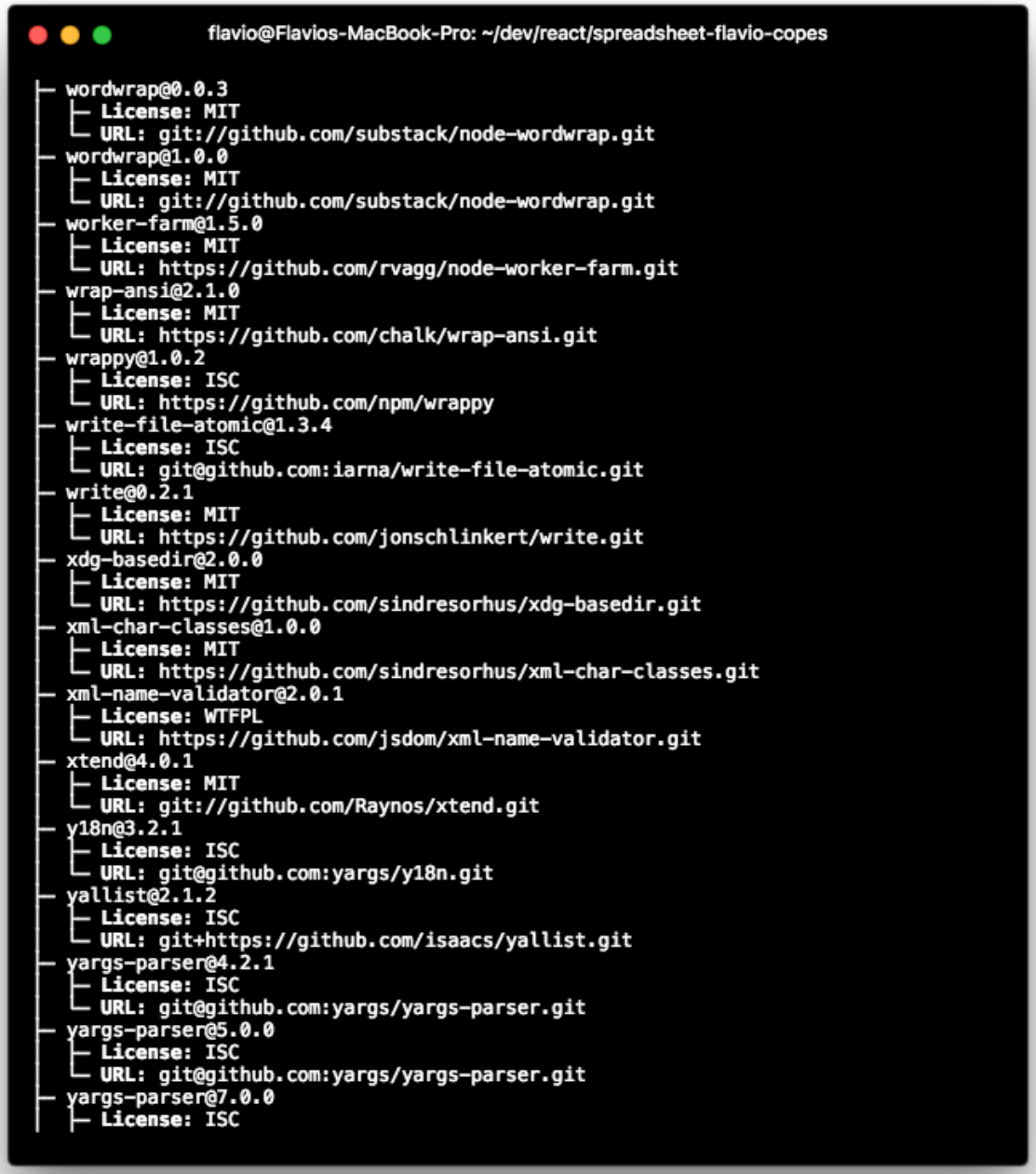
```
yarn remove package-name
```

Inspecing licenses

When installing many dependencies, which in turn might have lots of depencencies, you install a number of packages, of which you don't have any idea about the license they use.

Yarn provides a handy tool that prints the licens of any dependency you have:

```
yarn licenses ls
```



and it can also generate a disclaimer automatically including **all** the licenses of the projects you use:

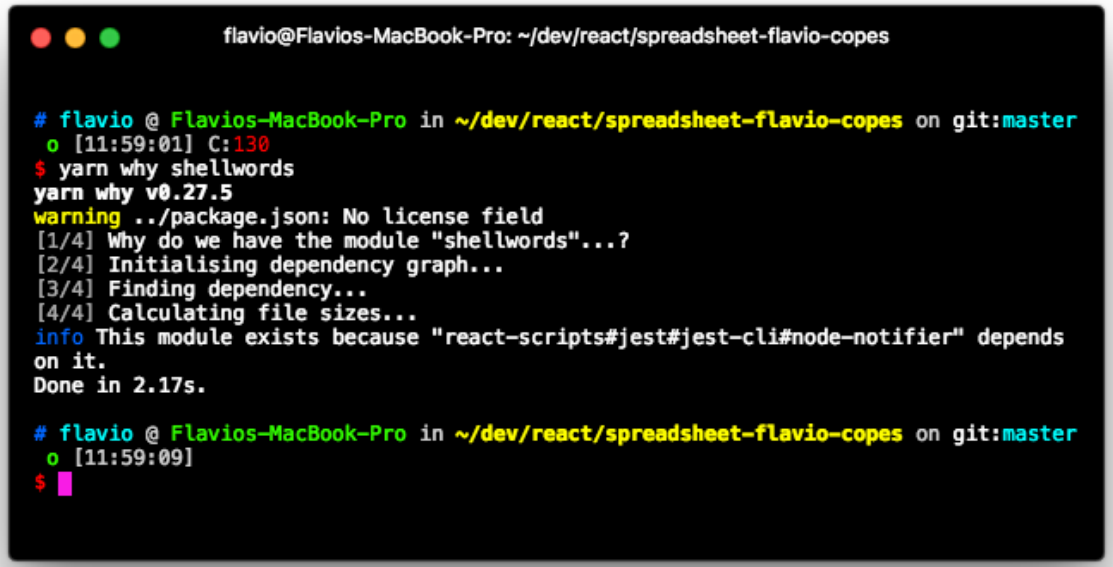
```
yarn licenses generate-disclaimer
```


Disclaimer generated by yarn

Inspecting dependencies

Do you ever check the `node_modules` folder and wonder why a specific package was installed? `yarn why` tells you:

```
yarn why package-name
```



Upgrading packages

If you want to upgrade a single package, run

```
yarn upgrade package-name
```

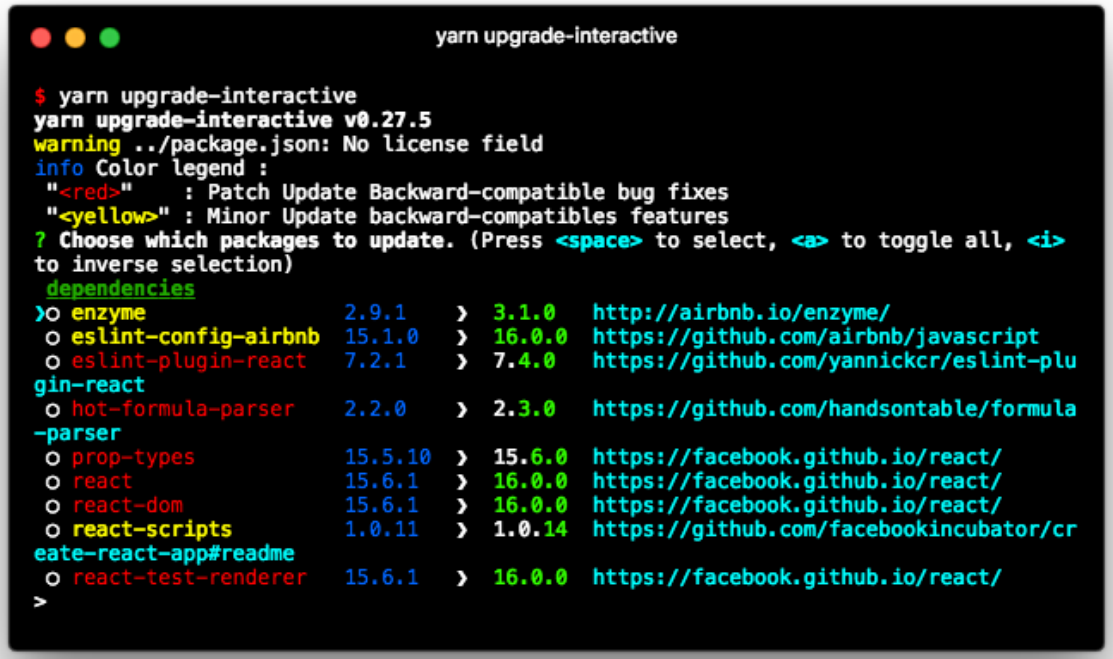
To upgrade all your packages, run

```
yarn upgrade
```

But this command can sometimes lead to problems, because you're blindly upgrading all the dependencies without worrying about major version changes.

Yarn has a great tool to selectively update packages in your project, which is a huge help for this scenario:

```
yarn upgrade-interactive
```



NPM

- Introduction to npm
- Downloads
    - Installing all dependencies
    - Installing a single package
    - Updating packages
- Versioning

- Running Tasks

Introduction to npm

`npm` means **node package manager**.

In January 2017 over 350000 packages were reported being listed in the npm registry, making it the biggest single language code repository on Earth, and you can be sure there is a package for (almost!) everything.

It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.

There are many things that `npm` does.

> *Yarn is an alternative to npm. Make sure you check it out as well.*

Downloads

`npm` manages downloads of dependencies of your project.

Installing all dependencies

If a project has a `packages.json` file, by running

```
npm install
```

it will install everything the project needs, in the `node_modules` folder, creating it if it's not existing already.

Installing a single package

You can also install a specific package by running

```
npm install <package-name>
```

Often you'll see more flags added to this command:

- `--save` installs and adds the entry to the `package.json` file *dependencies*
- `--save-dev` installs and adds the entry to the `package.json` file *devDependencies*

The difference is mainly that devDependencies are usually development tools, like a testing library, while `dependencies` are bundled with the app in production.

Updating packages

Updating is also made easy, by running

```
npm update
```

`npm` will check all packages for a newer version that satisfies your versioning constrains.

You can specify a single package to update as well:

```
npm update <package-name>
```

Versioning

In addition to plain downloads, `npm` also manages **versioning**, so you can specify any specific version of a package, or require a version higher or lower than what you need.

Many times you'll find that a library is only compatible with a major release of another library.

Or a bug in the latest release of a lib, still unfixed, is causing an issue.

Specifying an explicit version of a library also helps keeping everyone on the same exact version of a package, so that the whole team runs the same version until the `package.json` file is updated.

In all those cases, versioning helps a lot, and `npm` follows the Semantic Versioning (SEMVER) standard.

Running Tasks

The package.json file supports a format for specifying command line tasks that can be run by using

```
npm <task-name>
```

For example:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  },
}
```

It's very common to use this feature to run Webpack:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config
webpack.conf.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config
webpack.conf.js",
  },
}
```

So instead of those long commands, which is easy to forget or mistype, you can run

```
$ npm watch
$ npm dev
$ npm prod
```

ESLINT

Learn the basics of the most popular JavaScript linter, which can help to make your code adhere to a certain set of syntax conventions, check if the code contains possible sources of problems and if the code matches a set of standards you or your team define

- What is linter?
- ESLint
- Install ESLint globally
- Install ESLint locally
- Use ESLint in your favourite editor
- Common ESLint configurations
    - Airbnb style guide
    - React
    - Use a specific version of ECMAScript
    - Force strict mode
    - More advanced rules
- Disabling rules on specific lines

ESLint is a JavaScript linter.

What is linter?

Good question! A linter is a tool that identifies issues in your code.

Running a linter against your code can tell you many things:

- if the code adheres to a certain set of syntax conventions
- if the code contains possible sources of problems
- if the code matches a set of standards you or your team define

It will raise warnings that you, or your tools, can analyze and give you actionable data to improve your code.

ESLint

ESLint is a linter for the JavaScript programming language, written in Node.js.

It is hugely useful because JavaScript, being an interpreted language, does not have a compilation step and many errors are only possibly found at runtime.

ESLint will help you catch those errors. Which errors in particular you ask?

- avoid infinite loops in the `for` loop conditions
- make sure all getter methods return something
- disallow console.log (and similar) statements
- check for duplicate cases in a switch

- check for unreachable code
- check for JSDoc validity

and much more! The full list is available at https://eslint.org/docs/rules/

> *The growing popularity of Prettier as a code formatter made the styling part of ESLint kind of obsolete, but ESLint is still very useful to catch errors and code smells in your code.*

ESLint is very flexible and configurable, and you can choose which rules you want to check for, or which kind of style you want to enforce. Many of the available rules are disabled and you can turn them on in your `.eslintrc` configuration file, which can be global or specific to your project.

Install ESLint globally

Using npm.

```
npm install -g eslint

# create a `.eslintrc` configuration file
eslint --init

# run ESLint against any file with
eslint yourfile.js
```

Install ESLint locally

```
npm install eslint --save-dev

# create a `.eslintrc` configuration file
./node_modules/.bin/eslint --init

# run ESLint against any file with
./node_modules/.bin/eslint yourfile.js
```

Use ESLint in your favourite editor

The most common use of ESLint is within your editor of course.

Common ESLint configurations

ESLint can be configured in tons of different ways.

Airbnb style guide

A common setup is to use the Airbnb JavaScript coding style (https://github.com/airbnb/javascript) to lint your code.

Run

```
yarn add --dev eslint-config-airbnb
```

or

```
npm install --save-dev eslint-config-airbnb
```

to install the Airbnb configuration package, and add in your `.eslintrc` file in the root of your project:

```
{
  "extends": "airbnb",
}
```

React

Linting React code is easy with the React plugin:

```
yarn add --dev eslint-plugin-react
```

or

```
npm install --save-dev eslint-plugin-react
```

In your `.eslintrc` file add

```
{
  "extends": "airbnb",
  "plugins": [
      "react"
  ],
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    }
  }
}
```

Use a specific version of ECMAScript

ECMAScript changes version every year now.

The default is currently set to 5, which means pre-2015.

Turn on ES6 (or higher) by setting this property in `.eslintrc`:

```
{
  "parserOptions": {
    "ecmaVersion": 6,
  }
}
```

Force strict mode

```
{
  "parserOptions": {
    "ecmaFeatures": {
      "impliedStrict": true
    }
  }
}
```

More advanced rules

A detailed guide on rules can be found on the official site at https://eslint.org/docs/user-guide/configuring

Disabling rules on specific lines

Sometimes a rule might give a false positive, or you might be explicitly willing to take a route that ESLint flags.

In this case, you can disable ESLint entirely on a few lines:

```
/* eslint-disable */
alert('test');
/* eslint-enable */
```

or on a single line:

```
alert('test'); // eslint-disable-line
```

or just disable one or more specific rules for multiple lines:

```
/* eslint-disable no-alert, no-console */
alert('test');
console.log('test');
/* eslint-enable no-alert, no-console */
```

or for a single line:

```
alert('test'); // eslint-disable-line no-alert, quotes, semi
```

PRETTIER

Prettier is an opinionated code formatter. It is a great way to keep code formatted consistently for you and your team, and supports a lot of different languages out of the box

- Introduction to Prettier
- Less options
- Difference with ESLint

- Installation
- Prettier for beginners

Introduction to Prettier

Prettier is an opinionated code formatter.

It supports a lot of different syntax out of the box, including:

- JavaScript
- Flow, TypeScript
- CSS, SCSS, Less
- JSX
- GraphQL
- JSON
- Markdown

and with plugins (https://prettier.io/docs/en/plugins.html) you can use it for Python, PHP, Swift, Ruby, Java and more.

It integrates with the most popular code editors, including VS Code, Sublime Text, Atom and more.

Prettier is hugely popular, as in February 2018 it has been downloaded over 3.5 million times.

The most important links you need to know more about Prettier are

- https://prettier.io/
- https://github.com/prettier/prettier
- https://www.npmjs.com/package/prettier

Less options

I learned Go recently and one of the best things about Go is **gofmt**, an official tool that automatically formats your code according to common standards.

95% (made up stat) of the Go code around looks exactly the same, because this tool can be easily enforced and since the style is defined on you by the Go maintainers, you are much more likely to adapt to that standard instead of insisting on your own style. Like tabs vs spaces, or where to put an opening bracket.

This might sound like a limitation, but it's actually very powerful. All Go code looks the same.

Prettier is the `gofmt` for the rest of the world.

It has very few options, and **most of the decisions are already taken for you** so you can stop arguing about style and little things, and focus on your code.

Difference with ESLint

ESLint is a linter, it does not just format, but it also highlights some errors thanks to its static analysis of the code.

It is an invaluable tool and it can be used alongside Prettier.

ESLint also highlights formatting issues, but since it's a lot configurable, everyone could have a different set of formatting rules. Prettier provides a common ground for all.

Now, there are a few things you can customize, like:

- the tab width
- the use of single quotes vs double quotes
- the line columns number
- the use of trailing commas

and some others, but Prettier tries to keep the number of those customizations under control, to avoid becoming too customizable.

Installation

Prettier can run from the command line, and you can install it using Yarn or npm.

Another great use case for Prettier is to run it on PRs for your Git repositories, for example on GitHub.

If you use a supported editor the best thing is to use Prettier directly from the editor, and the Prettier formatting will be run every time you save.

For example here is the Prettier extension for VS Code: https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode

Prettier for beginners

If you think Prettier is just for teams, or for pro users, you are missing a good value proposition of this tool.

A good style enforces good habits.

Formatting is a topic that's mostly overlooked by beginners, but having a clean and consistent formatting is key to your success as a new developer.

Also, even if you started using JavaScript 2 weeks ago, with Prettier your code - style wise - will look just like code written from a JavaScript Guru writing JS since 1998.

GIT

Git is a free and Open Source *version control system* (VCS), a technology used to track older versions of files, providing the ability to roll back and maintain separate different versions at the same time

- What is Git
    - Distributed VCS
- Installing Git
    - OSX

    - Windows
    - Linux
- Initializing a repository
- Adding files to a repository
    - Add the file to the staging area
- Commit changes
- Branches
- Push and pull
    - Add a remote
    - Push
    - Pull
    - Conflicts
- Command Line vs Graphical Interface
    - GitHub Desktop
    - Tower
    - GitKraken
- A good Git workflow
    - The feature is a quick one
    - The feature will take more than one commit to finish
    - Hotfix
    - Develop is unstable. Master is the latest stable release

---

What is Git

Git is a free and Open Source *version control system* (VCS), a technology used to track older versions of files, providing the ability to roll back and maintain separate different versions at the same time.

Git is a successor of SVN and CVS, two very popular version control systems of the past. First developed by Linus Torvalds (the creator of Linux), today is the go-to system which you can't avoid if you make use of Open Source software.

Distributed VCS

Git is a distributed system. Many developers can *clone* a repository from a central location, work independently on some portion of code, and then *commit* the changes back to the central location where everybody updates.

Git makes it very easy for developers to collaborate on a codebase simultaneously and provides tools they can use to combine all the independent changes they make.

A very popular service that hosts Git repositories is GitHub, especially for Open Source software, but we can also mention BitBucket, GitLab and many others which are widely used by teams all over the world to host their code publicly and also privately.

Installing Git

Installing Git is quite easy on all platforms:

OSX

Using Homebrew (http://brew.sh/) , run:

```
brew install git
```

Windows

Download and install Git for Windows (https://git-for-windows.github.io/) .

Linux

Use the package manager of your distribution to install Git. E.g.

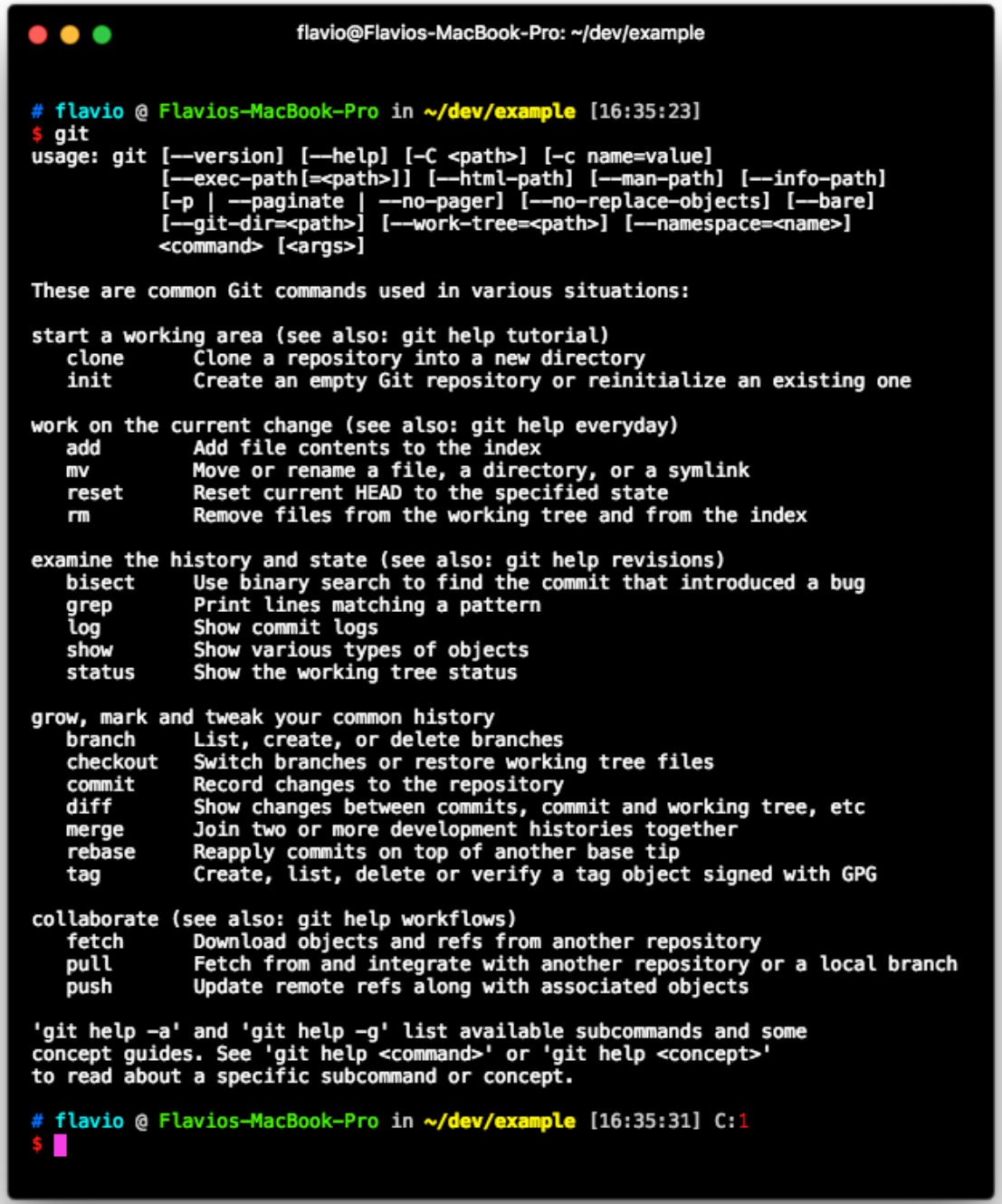
```
sudo apt-get install git
```

or

```
sudo yum install git
```

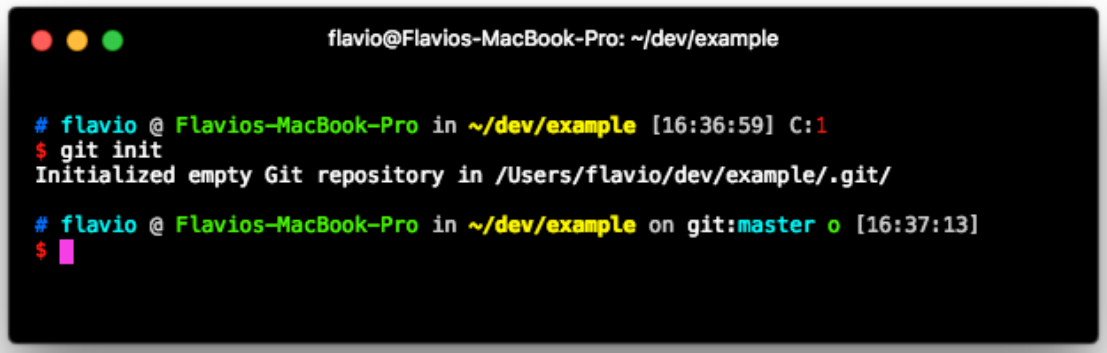
Initializing a repository

Once Git is installed on your system, you are able to access it using the command line by typing `git`.

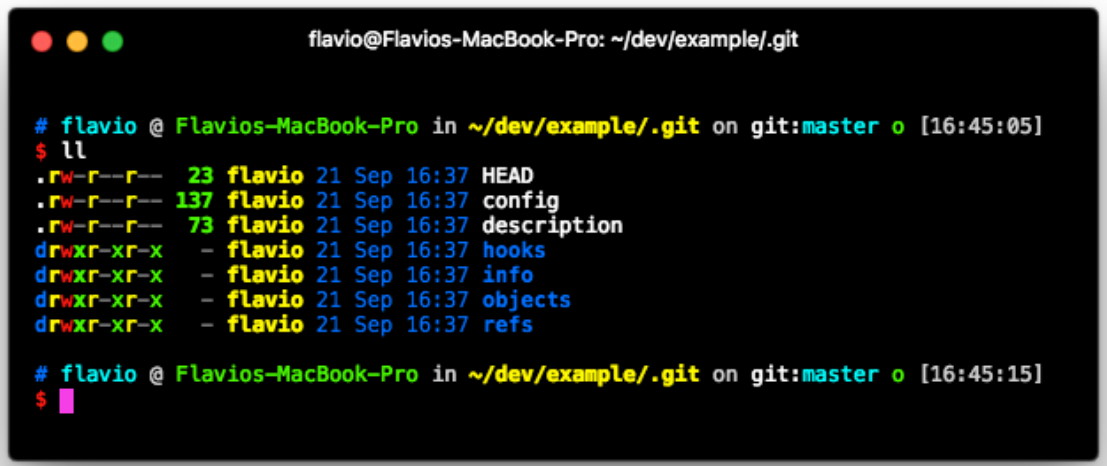


Suppose you have a clean folder. You can initialize a Git repository by typing

```
git init
```



What does this command do? It creates a `.git` folder in the folder where you ran it. If you don't see it, it's because it's a hidden folder, so it might not be shown everywhere, unless you set your tools to show hidden folders.



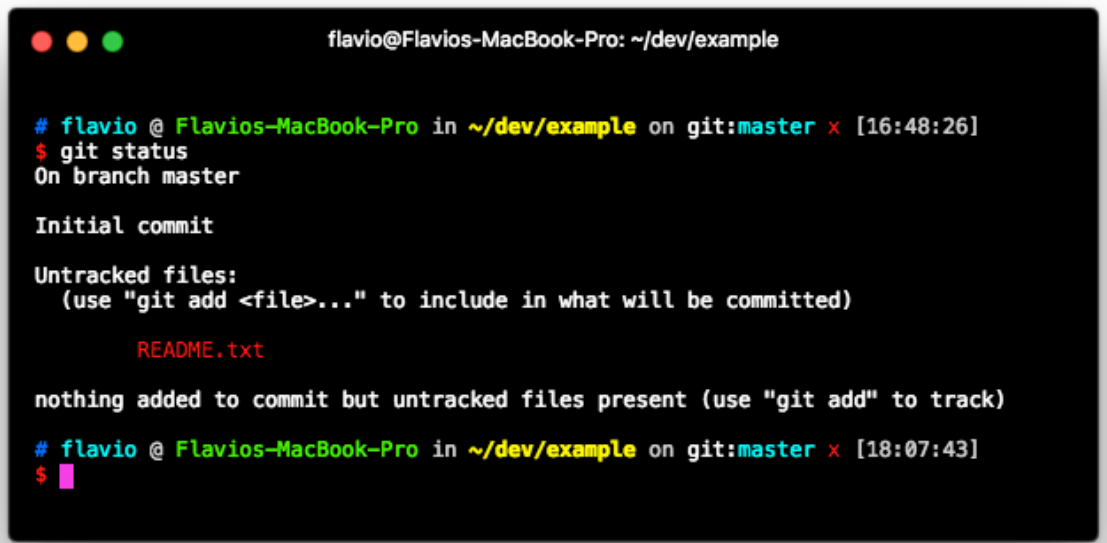
Anything related to Git in your newly created repository will be stored into this `.git` directory, all except the `.gitignore` file, which I'll talk about in the next article.

Adding files to a repository

Let's see how a file can be added to Git. Type:

```
echo "Test" > README.txt
```

to create a file. The file is now in the directory, but Git was not told to add it to its index, as you can see what `git status` tells us:

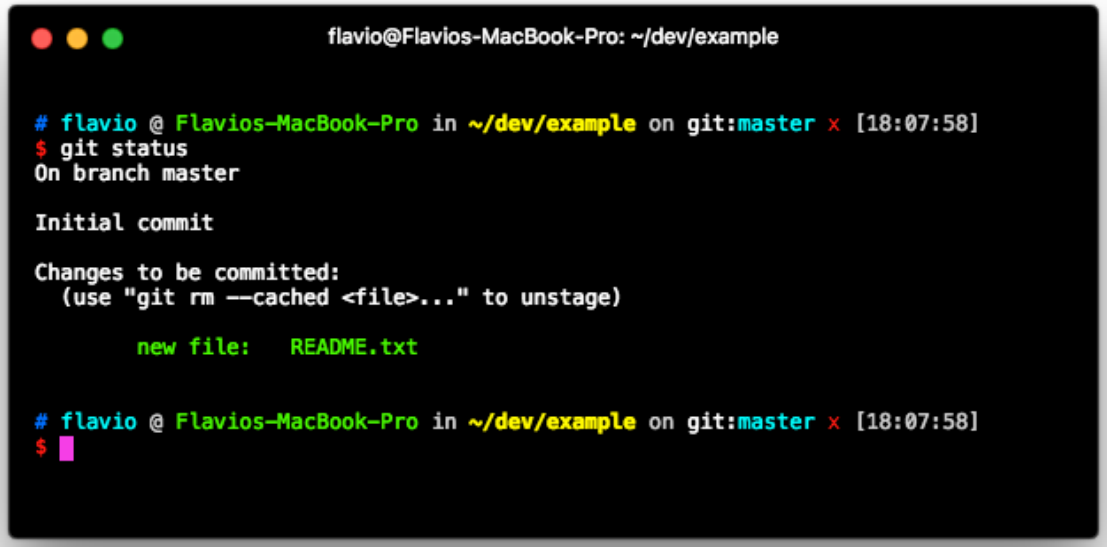


Add the file to the staging area

We need to add the file with

```
git add README.txt
```

to make it visible to Git, and be put into the **staging area**:



Once a file is in the staging area, you can remove it by typing:

```
git reset README.txt
```

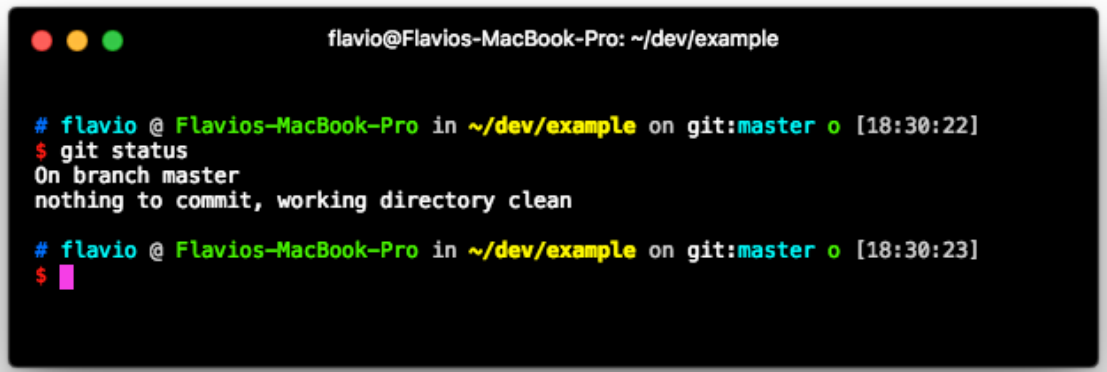
But usually what you do once you add a file is commit it.

Commit changes

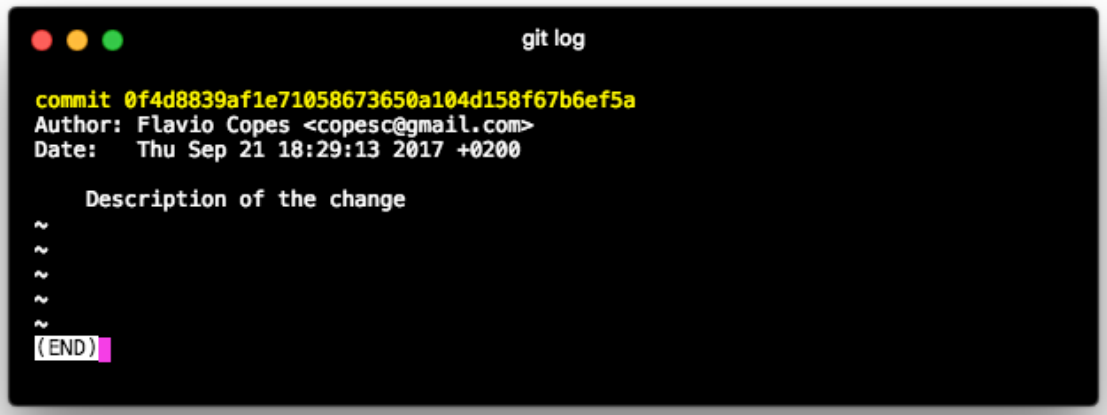
Once you have one or more changes to the staging area, you can commit them using

```
git commit -am "Description of the change"
```

This cleans the status of the staging area:



and permanently stores the edit you made into a record store, which you can inspect by typing `git log`:



Branches

When you commit a file to Git, you are committing it into the current branch.

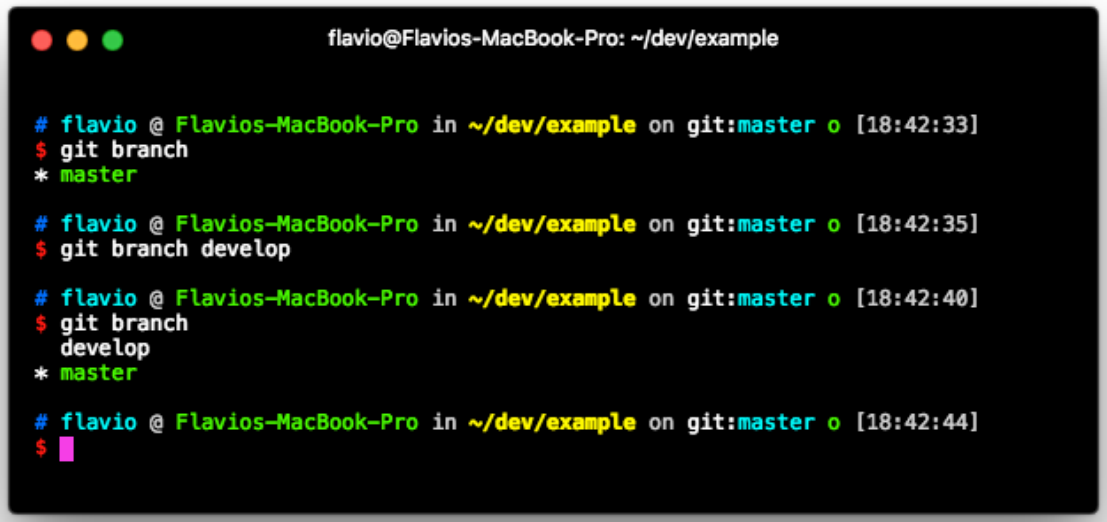
Git allows you to work simultaneously on multiple, separate branches, different lines of development which represent forks of the main branch.

Git is very flexible: you can have an indefinite number of branches active at the same time, and they can be developed independently until you want to merge one of them into another.

Git by default creates a branch called `master`. It's not special in any way other than it's the one created initially.

You can create a new branch called `develop` by typing

```
git branch develop
```



As you can see, `git branch` lists the branches that the repository has. The asterisk indicates the current branch.

When creating the new branch, that branch points to the latest commit made on the current branch. If you switch to it (using `git checkout develop`) and run `git log`, you'll see the same log as the branch that you were previously.

Push and pull

In Git you always commit locally. This is a very nice benefit over SVN or CSV where all commits had to be immediately pushed to a server.

You work offline, do as many commits as you want, and once you're ready you **push** them to the server, so your team members, or the community if you are pushing to GitHub, can access your latest and greatest code.

> *Push sends your changes.*
> *Pull downloads remote changes to your working copy.*

Before you can play with push and pull, however, you need to add a **remote**!

Add a remote

A remote is a clone of your repository, positioned on another machine.

I'll do an example with GitHub. If you have an existing repository, you can publish it on GitHub. The procedure involves creating a repository on the platform, through their web interface, then you add that repository as a remote, and you push your code there.

To add the remote type

```
git remote add origin https://github.com/YOU/REPONAME.git
```

> *An alternative approach is creating a blank repo on GitHub and cloning it locally, in which case the remote is automatically added for you*

Push

Once you're done, you can push your code to the remote, using the syntax `git push <remote> <branch>`, for example:

```
git push origin master
```

You specify `origin` as the remote, because you can technically have more than one remote. That is the name of the one we added previously, and it's a convention.

Pull

The same syntax applies to pulling:

```
git pull origin master
```

tells Git to push the `master` branch from `origin`, and merge it in the current local branch.

Conflicts

In both push and pull there is a problem to consider: if the remote contains changes incompatible with your set of commits, the operation will fail.

This happens when the remote contains changes subsequent to your latest pull, which affects lines of code you worked on as well.

In the case of push this is usually solved by pulling changes, analyzing the conflicts, and then making a new commit that solves them.

In the case of pull, your working copy will automatically be edited with the conflicting changes, and you need to solve them, and make a new commit so the codebase now includes the problematic changes that were made on the remote.

Command Line vs Graphical Interface

Up to now I talked about the command line Git application.

This was key to introduce you to how Git actually works, but in the day-to-day operations, you are most likely to use an app that exposes you those commands via a nice UI, although many developers I know like to use the CLI.

> *The CLI (command line) commands will still prove themselves to be useful if you need to setup Git using SSH on a remote server, for instance. It's not useless knowledge at all!*

That said, there are many very nice apps that are made to simplify the life of a developer that turn out very useful especially when you dive more into the complexity of a Git repository. The easy steps are easy everywhere, but things could quickly grow to a point where you might find it hard to use the CLI.

Some of the most popular apps are

GitHub Desktop

https://desktop.github.com

Free, at the time of writing only available for Mac and Win

Tower

https://www.git-tower.com

Paid, at the time of writing only available for Mac and Win

GitKraken

https://www.gitkraken.com

Free / Paid depending on the needs, for Mac, Win and Linux

A good Git workflow

Different developers and teams like to use different strategies to manage Git effectively. Here is a strategy I used on many teams and on widely used open source projects, and I saw used by many big and small projects as well.

The strategy is inspired by the famous A successful Git branching model (http://nvie.com/posts/a-successful-git-branching-model/) post.

I have only 2 permanent branches: **master** and **develop**.

Those are the rules I follow in my daily routine:

When I take on a new issue, or decide to incorporate a feature, there are 2 main roads:

The feature is a quick one

The commits I'll make won't break the code (or at least I hope so): I can commit on develop, or do a quick feature branch, and then merge it to develop.

The feature will take more than one commit to finish

Maybe it will take days of commits before the feature is finished and it gets stable again: I do a feature branch, then merge to develop once ready (it might take weeks).

Hotfix

If something on our production server requires immediate action, like a bugfix I need to get solved ASAP, I do a short hotfix branch, fix the thing,

test the branch locally and on a test machine, then merge it to master and develop.

Develop is unstable. Master is the latest stable release

The develop branch will always be in a state of flux, that's why it should be put on a 'freeze' when preparing a release. The code is tested and every workflow is checked to verify code quality, and it's prepared for a merge into master.

Every time develop or another hotfix branch is merged into master, I **tag it with a version number**, and if on GitHub I also create a **release**, so it's easy to move back to a previous state if something goes wrong.

GITHUB

GitHub is a website where millions of developers gather every day to collaborate on open source software. It's also the place that hosts billions of lines of code, and also a place where users of software go to report issues they might have. Learn all the most important pieces of GitHub that you should know as a developer

- Introduction to GitHub
- Why GitHub?

- GitHub issues
- Social coding
    - Follow
    - Stars
    - Fork
    - Popular = better
- Pull requests
- Project management
- Comparing commits
- Webhooks and Services
    - Webhooks
    - Services
- Final words

Introduction to GitHub

GitHub is a website where millions of developers gather every day to collaborate on open source software. It's also the place that hosts billions of lines of code, and also a place where users of software go to report issues they might have.

In short, it's a platform for software developers, and it's built around Git.

> *TIP: If you don't know about Git yet, checkout the Git guide.*

As a developer **you can't avoid using GitHub daily**, either to host your code or to make use of other people's code. This post explains you some key concepts of GitHub, and how to use some of its features that improve your workflow, and how to integrate other applications into your process.

Why GitHub?

Now that you know what GitHub *is*, you might ask *why* you should use it.

GitHub after all is managed by a private company, which profits from hosting people's code. So why should you use that instead of similar platforms such as BitBucket or GitLab, which are very similar?

Beside personal preferences, and technical reasons, there is one big reason: everyone uses GitHub, so the network effect is huge.

Major codebases migrated over time to Git from other version control systems, because of its convenience, and GitHub was historically well positioned into (and put a lot of effort to "win") the Open Source community.

So today any time you look up some library, you will 99% of the times find it on GitHub.

Apart from Open Source code, many developers also host private repositories on GitHub because of the convenience of a unique platform.

GitHub issues

GitHub issues are one of the most popular bug tracker in the world.

It provides the owners of a repository the ability to organize, tag and assign to milestones issues.

If you open an issue on a project managed by someone else, it will stay open until either you close it (for example if you figure out the problem you had) or if the repo owner closes it.

Sometimes you'll get a definitive answer, other times the issue will be left open and tagged with some information that categorizes it, and the developer could get back to it to fix a problem or improve the codebase with your feedback.

Most developers are not paid to support their code released on GitHub, so you can't expect prompt replies, but other times Open Source repositories are published by companies that either provide services around that code, or have commercial offerings for versions with more features, or a plugin-based architecture, in which case they might be working on the open source software as paid developers.

Social coding

Some years ago the GitHub logo included the "social coding" tagline.

What did this mean, and is that still relevant? It certainly is.

Follow

With GitHub **you can follow developers**, by going on their profile and clicking “follow”.

You can also follow a repository, by clicking the “**watch**” button on a repo.

In both cases the activity will show up in your dashboard. You don’t follow like in Twitter, where you see what people say, but **you see what people do**.

Stars

One big feat of GitHub is the ability to **star a repository**. This action will include it in your “starred repositories” list, which allows you to find things you found interesting before, and it’s also one of the most important rating mechanisms, as the more stars a repo has, the more important it is, and the more it will show up in search results.

Major projects can have 70.000 and more stars.

GitHub also has a trending page (https://github.com/trending) where it features the repositories that get the most stars in a determined period of time, e.g. today or this week or month.

Getting into those trending lists can cause other network effects like being featured on other sites, just because you have more visibility.

Fork

The last important network indicator of a project is the number of forks.

This is key to how GitHub works, as a fork is the base of a Pull Request (PR), a change proposal. Starting from your repository, a person forks it, makes some changes, then creates a PR to ask you to merge those changes.

Sometimes the person that forks never asks you to merge anything, just because they liked your code and decided to add something on top of it, or they fixed some bug they were experiencing.

A fork clones the files of a GitHub project, but not any of the stars or issues of the original project.

Popular = better

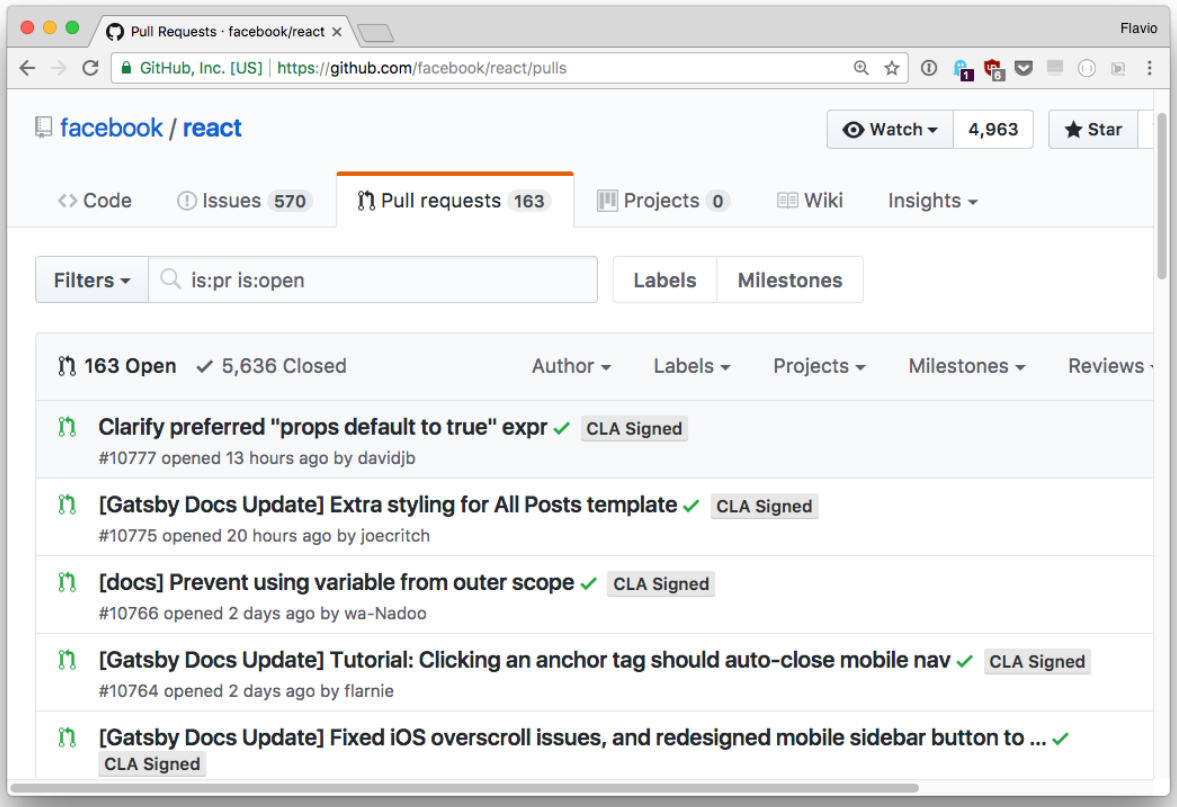
All in all, those are all key indicators of the popularity of a project, and generally along with the date of the latest commit and the involvement of the author in the issues tracker, is a useful indication of whether or not you should rely on a library or software.

Pull requests

Before I introduced what is a Pull Request (PR)

Starting from your repository, a person forks it, makes some changes, then creates a PR to ask you to merge those changes.

A project might have hundreds of PRs, generally the more popular a project, the more PRs, like the React project:



Once a person submits a PR, an easy process using the GitHub interface, it needs to be reviewed by the core maintainers of the project.

Depending on the *scope* of your PR (the number of changes, or the number of things affected by your change, or the complexity of the code touched) the maintainer might need more or less time to make sure your changes are compatible with the project.

A project might have a clear timeline of changes they want to introduce. The maintainer might like to keep things simple while you are introducing a complex architecture in a PR.

This is to say that **not always a PR gets accepted fast**, and also **there is no guarantee that the PR will even get accepted**.

In the example i posted above, there is a PR in the repo that dates back 1.5 years. And this happens in **all** the projects.

Project management

Along with issues, which are the place where developers get feedback from users, the GitHub interface offers other features aimed at helping project management.

One of those is **Projects**. It's very new in the ecosystem and very rarely used, but it's a **kanban board** that helps organizing issues and work that needs to be done.

The **Wiki** is intended to be used as a documentation for users. One of the most impressive usage of the Wiki I saw up to now is the Go Programming Language GitHub Wiki (https://github.com/golang/go/wiki) .

Another popular project management aid is **milestones**. Part of the issues page, you can assign issues to specific milestones, which could be release targets.

Speaking of releases, GitHub enhances the **Git tag** functionality by introducing **releases**.

A Git tag is a pointer to a specific commit, and if done consistently, helps you roll back to previous version of your code without referencing specific commits.

A GitHub release builds on top of Git tags and represents a complete release of your code, along with zip files, release notes and binary assets that might represent a fully working version of your code end product.

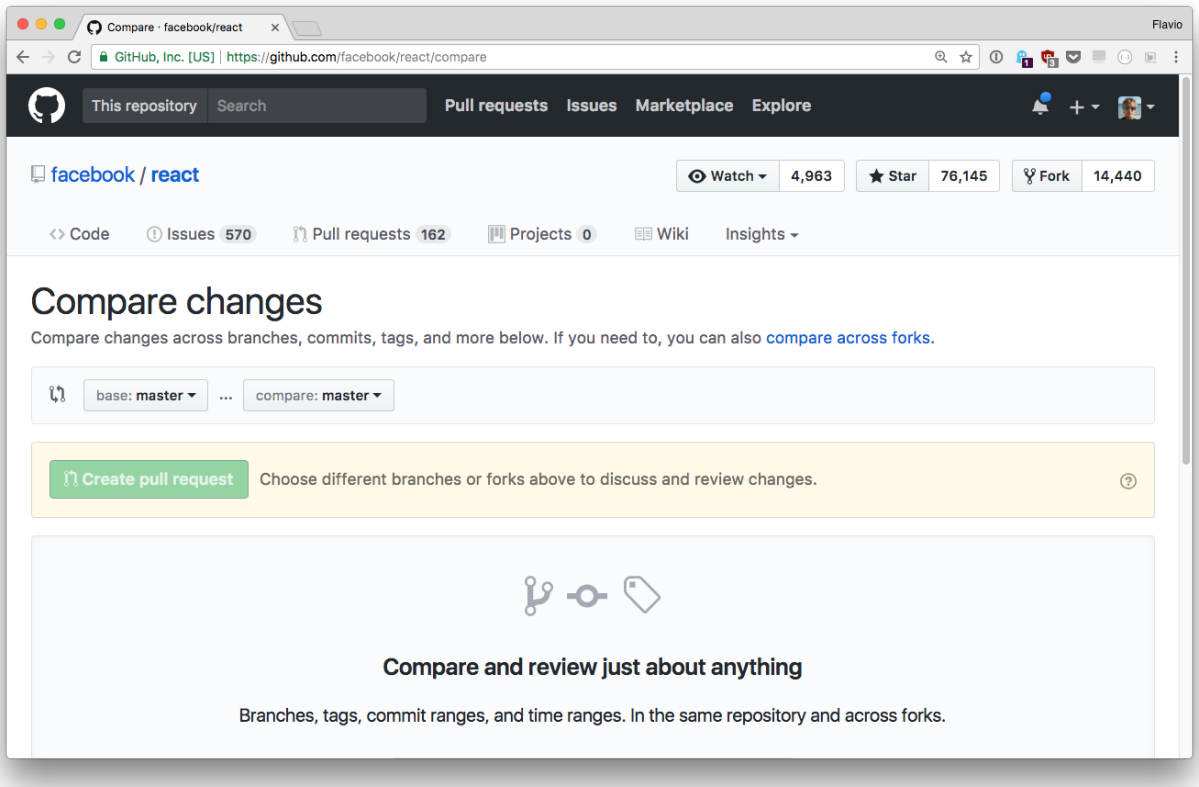
While a Git tag can be created programmatically (e.g. using the Command Line `git` program), creating a GitHub release is a manual process that happens through the GitHub UI. You basically tell GitHub to create a new release and tell them which tag you want to apply that release to.

Comparing commits

GitHub offers many tools to work with your code.

One of the most important things you might want to do is compare one branch to another one. Or, compare the latest commit with the version you are currently using, to see which changes were made over time.

GitHub allows you to do this with the **compare view**, just add `/compare` to the repo name, for example:
https://github.com/facebook/react/compare



For example here I choose to compare the latest *React v15.x* to the latest *v16.0.0-rc* version available at the time of writing, to check what's changed:

Compare React Versions

The view shows you **the commits made** between two releases (or tags or commits references) and **the actual diff**, **if the number of changes is lower than a reasonable amount**.

Webhooks and Services

GitHub offers many features that help the developer workflow. One of them is webhooks, the other one is services.

Webhooks

Webhooks allow external services to be pinged when certain events happen in the repository, like when code is pushed, a fork is made, a tag was created or deleted.

When an event happens, GitHub sends a POST request to the URL we told it to use.

A common usage of this feature is to ping a remote server to fetch the latest code from GitHub when we push an update from our local computer.

We push to GitHub, GitHub tells the server we pushed, the server pulls from GitHub.

Services

GitHub services, and the new GitHub apps, are 3rd part integrations that improve the developer experience or provide a service to you.

For example you can setup a test runner to run the tests automatically every time you push some new commits, using TravisCI (https://travis-ci.org/) .

You can setup Continuous Integration using CircleCI (https://circleci.com/) .

You might create a Codeclimate (https://codeclimate.com/) integration that analyzes the code and provides a report of technical debt and test

coverage.

Final words

GitHub is an amazing tool and service to take advantage of, a real gem in today's developer toolset. This tutorial will help you start, but the real experience of working on GitHub on open source (or closed source) projects is something not to be missed.

BROWSER DEV TOOLS

The Browser DevTools are a fundamental element in the frontend developer toolbox, and they are available in all modern browsers. Discover the basics of what they can do for you

- The Browser DevTools
- HTML Structure and CSS

    - The HTML panel
    - The CSS styles panel
- The Console
    - Executing custom JavaScript
    - Error reporting
- The emulator
- The network panel
- JavaScript debugger
- Application and Storage
    - Storage
    - Application
- Security tab
- Audits

The Browser DevTools

I don't think there was a time where websites and web applications were easy to build, as for backend technologies, but client-side development was surely easier than now, generally speaking.

Once you figured out the differences between Internet Explored and Netscape Navigator, and avoided the proprietary tags and technology, all you had to use was HTML and later CSS.

JavaScript was a tech for creating dialog boxes and a little bit more, but was definitely not as pervasive as today.

Although lots of web pages are still plain HTML + CSS, like this page, many other websites are real applications that run in the browser.

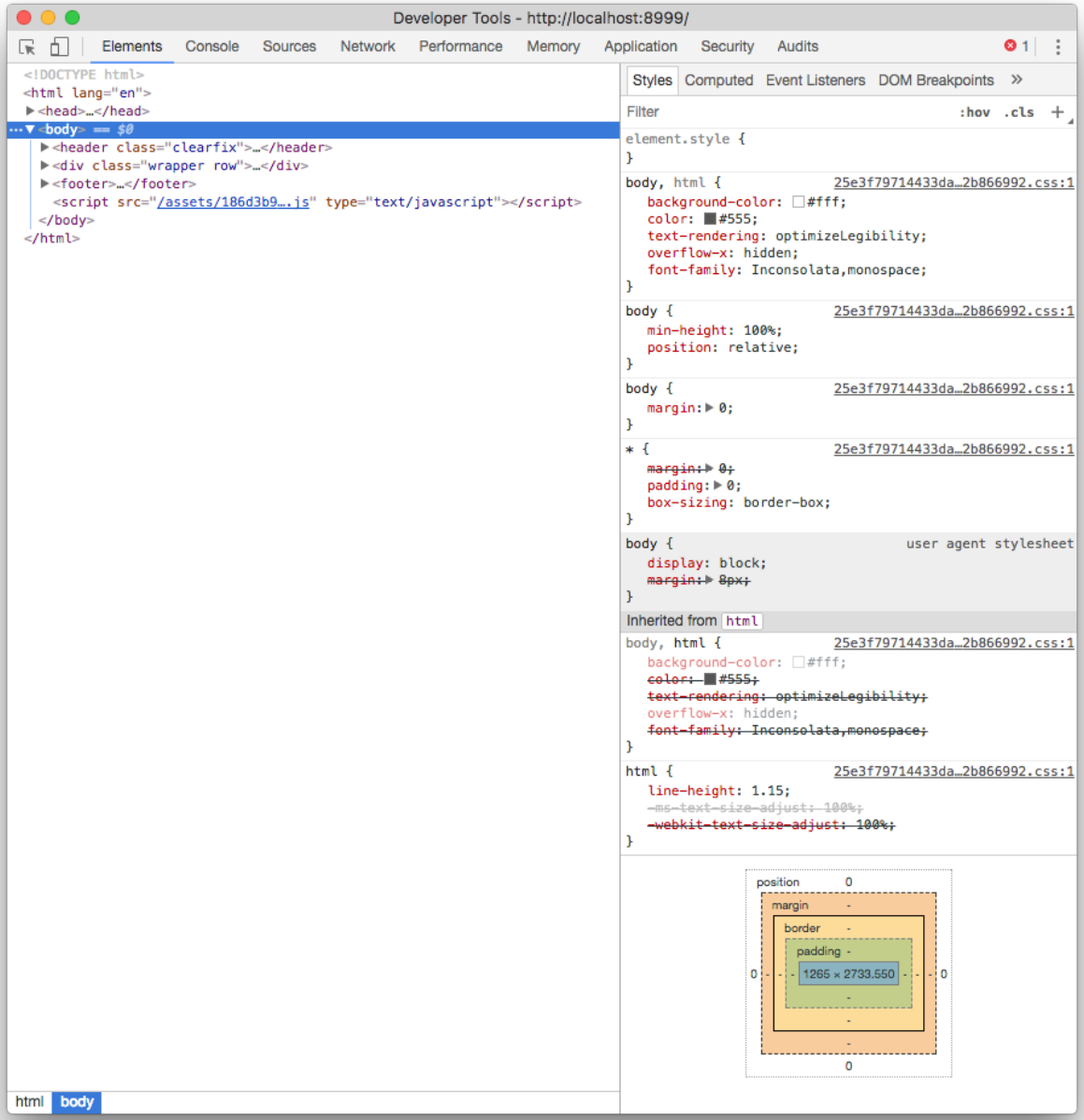
Just providing the source of the page, like browser did once upon a time, was not enough.

Browser had to provide much more information on how they rendered the page, and what the page is currently doing, hence they introduced a **feature for developers**: their **developer tools**.

Every browser is different and so their dev tools are slightly different. At the time of writing my favorite developer tools are provided by Chrome, and this is the browser we'll talk here, although also Firefox and Edge have great tools as well. I will soon add coverage of the Firefox DevTools.

HTML Structure and CSS

The most basic form of usage, and a very common one, is inspecting the content of a webpage. When you open the DevTools that's the panel the Elements panel is what you see:



The HTML panel

On the left, the HTML that composes the page.

Hovering the elements in the HTML panel highlights the element in the page, and clicking the first icon in the toolbar allows you to click an element in the page, and analyze it in the inspector.

You can drag and drop elements in the inspector to live change their positioning in the page.

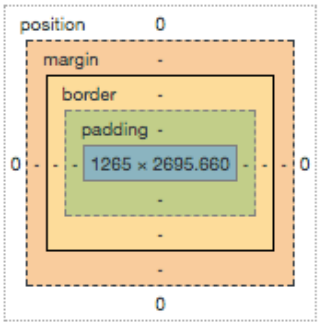
The CSS styles panel

On the right, the CSS styles that are applied to the currently selected element.

In addition to editing and disabling properties, you can add a new CSS property, with any target you want, by clicking the + icon.

Also you can trigger a state for the selected element, so you can see the styles applied when it's active, hovered, on focus.

At the bottom, the **box model** of the selected element helps you figure out margins, paddings, border and dimensions at a quick glance:



The Console

The second most important element of the DevTools is the Console.

The Console can be seen on its own panel, or by pressing `Esc` in the Elements panel, it will show up in the bottom.

The Console serves mainly two purposes: *executing custom JavaScript* and *error reporting*.

Executing custom JavaScript

At the bottom of the Console there is a blinking cursor. You can type any JavaScript there, and it will be promptly executed. As an example, try running:

```
alert('test')
```

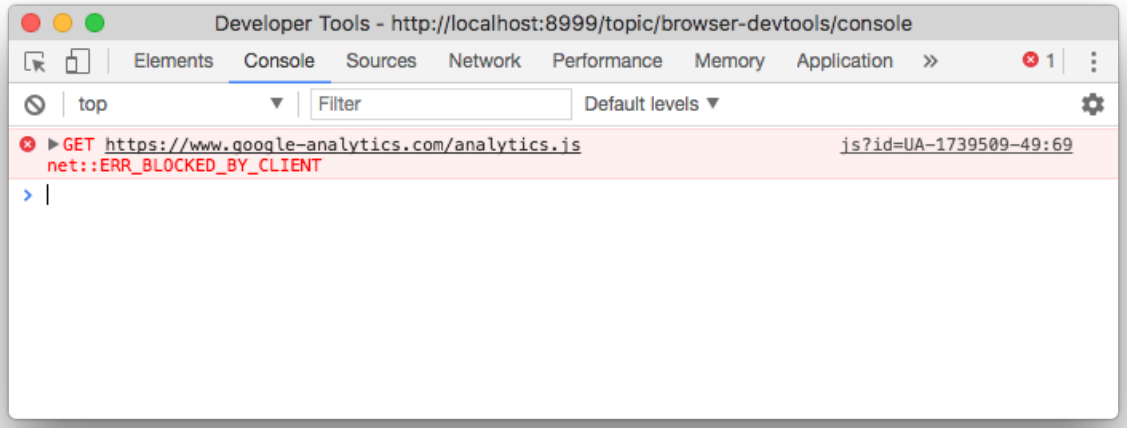
The special identifier `$0` allows you to reference the element currently selected in the elements inspector. If you want to reference that as a jQuery selector, use `$($0)`.

You can write more than one line with `shift-enter`. Pressing enter at the end of the script runs it.

Error reporting

Any error, warning or information that happens while rendering the page, and subsequently executing the JavaScript, is listed here.

For example failing to load a resource from the network, with information on *why*, is reported in the console.



In this case, clicking the resource URL brings you to the Network panel, showing more info which you can use to determine the cause of the problem.

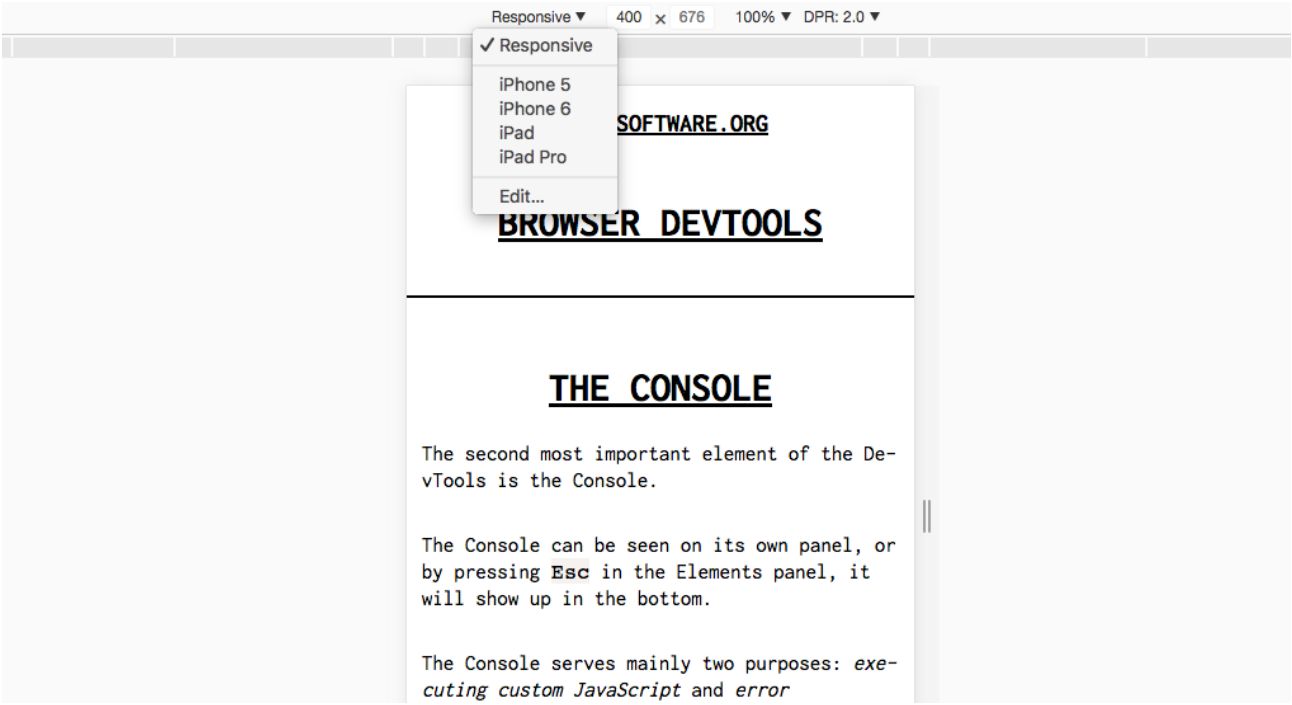
You can filter those messages by level (Error / Warning / Info) and also filter them by content.

Those messages can be user-generated in your own JavaScript by using the **Console API**:

```
console.log('Some info message')
console.warn('Some warning message')
console.error('Some error message')
```

The emulator

The Chrome DevTools embed a very useful device emulator which you can use to visualize your page in every device size you want.

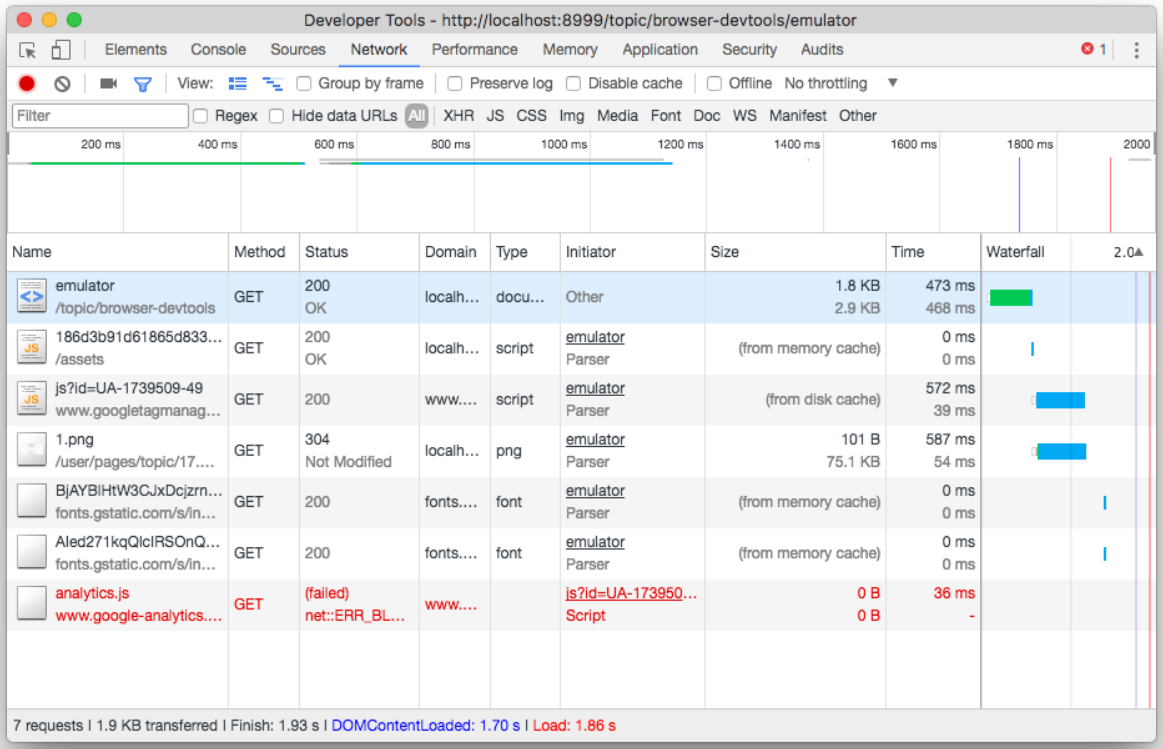


You can choose from the presets the most popular mobile devices, including iPhones, iPads, Android devices and much more, or specify the pixel dimensions yourself, and the screen definition (1x, 2x retina, 3x retina HD).

In the same panel you can setup **network throttling** for that specific Chrome tab, to emulate a low speed connection and see how the page loads, and the "**show media queries**" option shows you how media queries modify the CSS of the page.

The network panel

The Network Panel of the DevTools allows you to see all the connections that the browser must process while rendering a page.



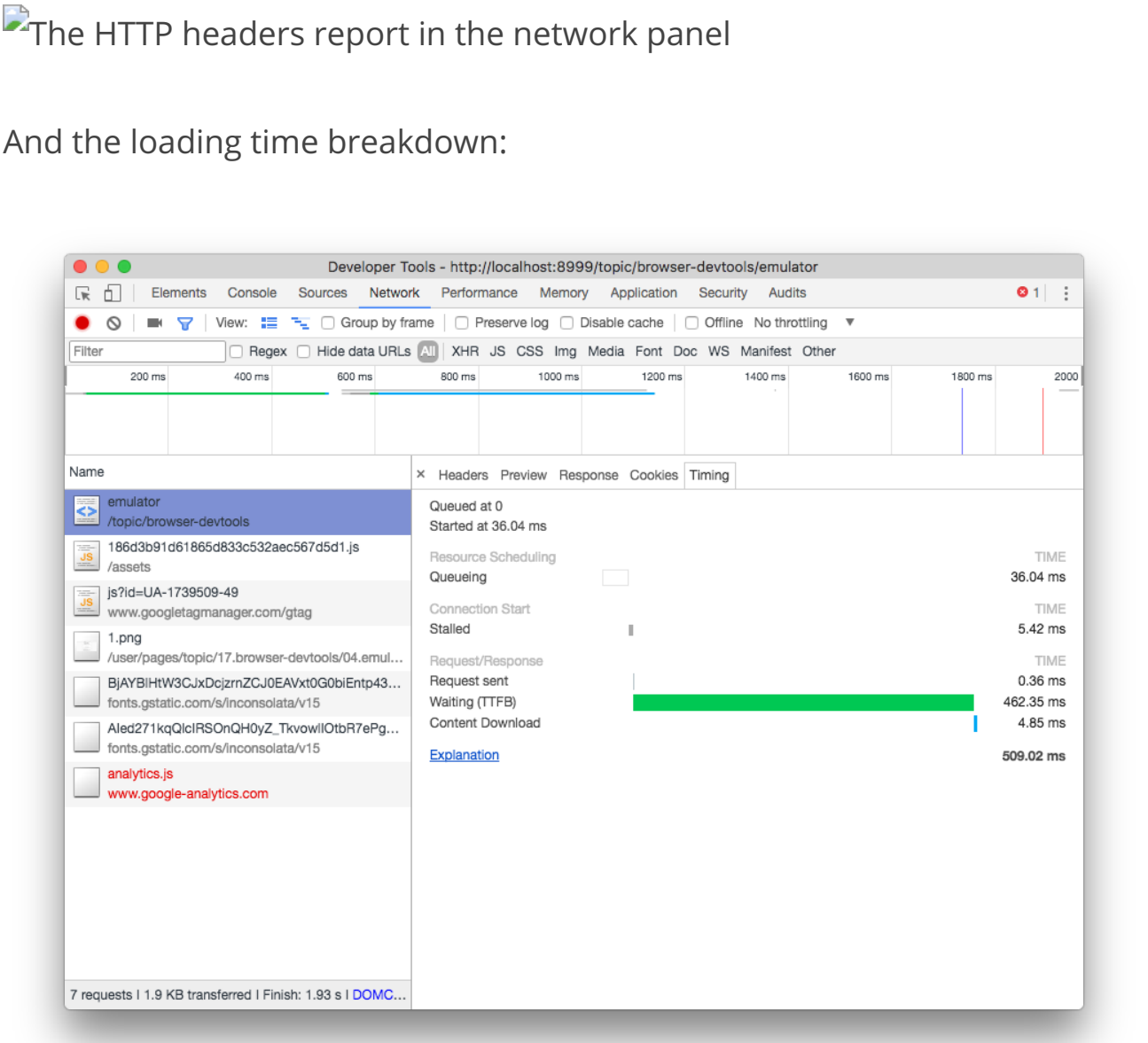
At a quick glance the page shows:

- a toolbar where you can setup some options and filters
- a loading graph of the page as a whole
- every single request, with HTTP method, response code, size and other details
- a footer with the summary of the total requests, the total size of the page and some timing indications.

A very useful option in the toolbar is **preserve log**. By enabling it, you can move to another page, and the logs will not be cleared.

Another very useful tool to track loading time is **disable cache**. This can be enabled globally in the DevTools settings as well, to always disable cache when DevTools is open.

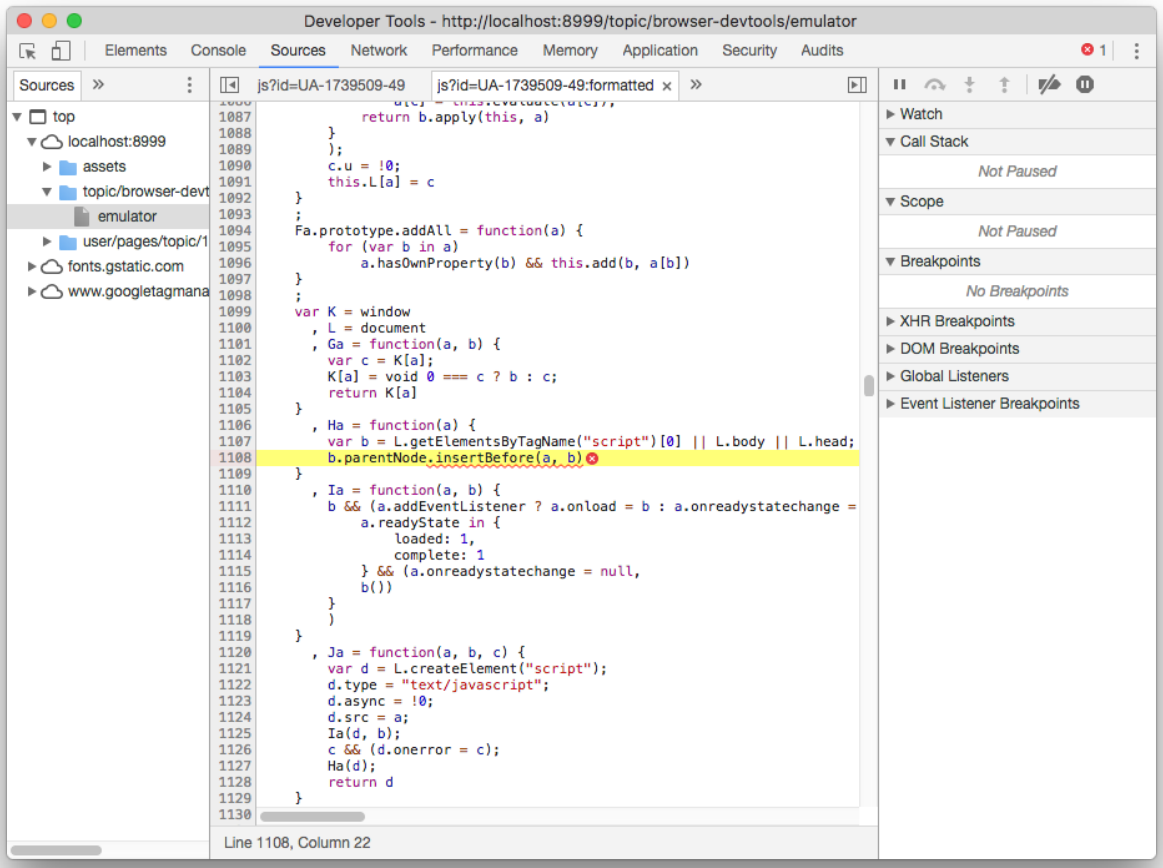
Clicking a specific request in the list shows up the detail panel, with HTTP Headers report:



And the loading time breakdown:

JavaScript debugger

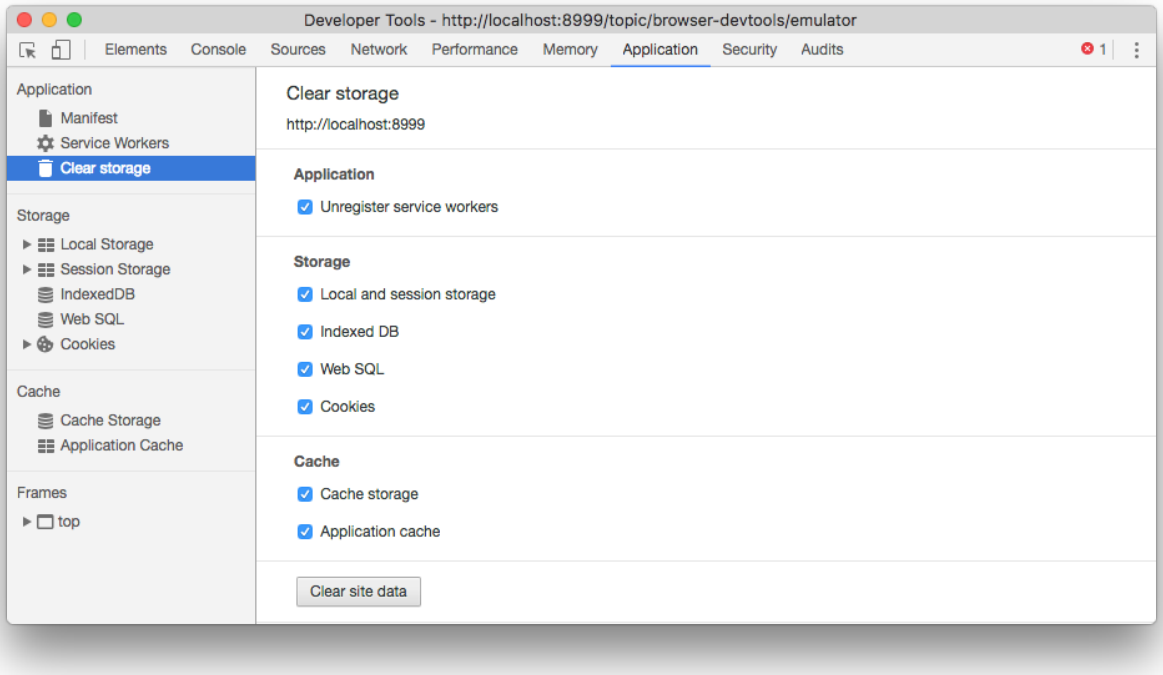
If you click an error message in the DevTools Console, the Sources tab opens and in addition to pointing you to the file and line where the error happened, you have the option to use the JavaScript debugger.



This is a full-featured debugger. You can set breakpoints, watch variables, and listen to DOM changes or break on specific XHR (AJAX) network requests, or event listeners.

Application and Storage

The Application tab gives you lots of information about which information is stored inside the browser relative to your website.



Storage

You gain access to detailed reports and tools to interact with the application storage:

- Local Storage
- Session Storage
- IndexedDb
- Web SQL
- Cookies

and you can quickly wipe any information, to start with a clean slate.

Application

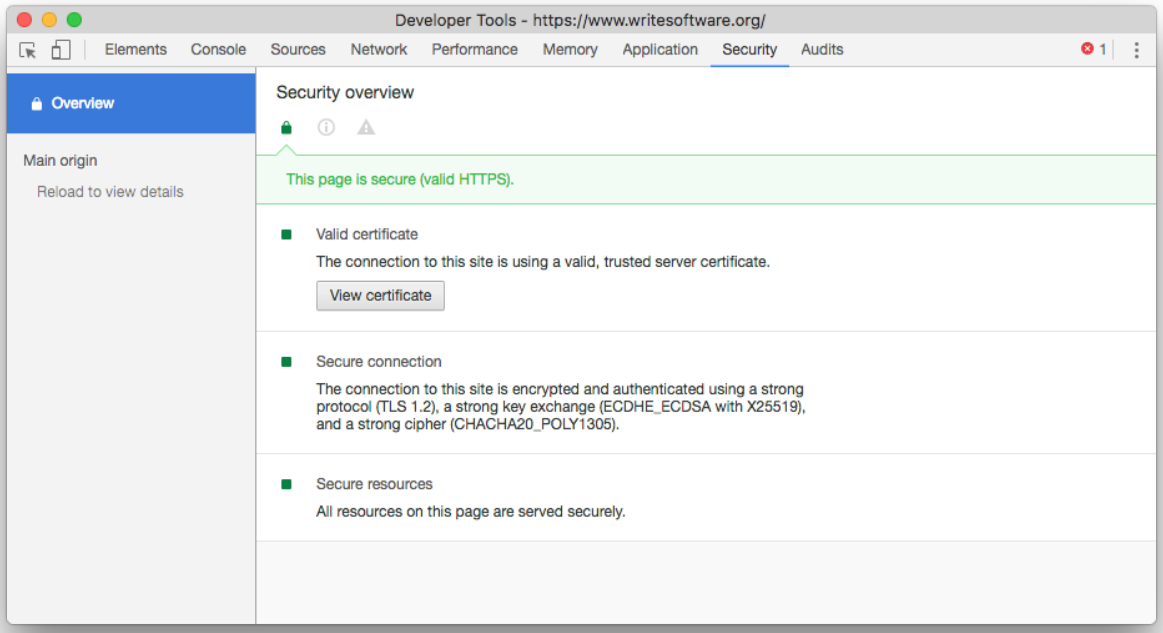
This tab also gives you tools to inspect and debug Progressive Web Apps.

Click **manifest** to get information about the web app manifest, used to allow mobile users to add the app to their home, and simulate the “add to homescreen” events.

Service workers let you inspect your application service workers. If you don’t know what service workers are, in short they are a fundamental technology that powers modern web apps, to provide features like notification, capability to run offline and synchronize across devices.

Security tab

The Security tab gives you all the information that the browser has relatively to the security of the connection to the website.

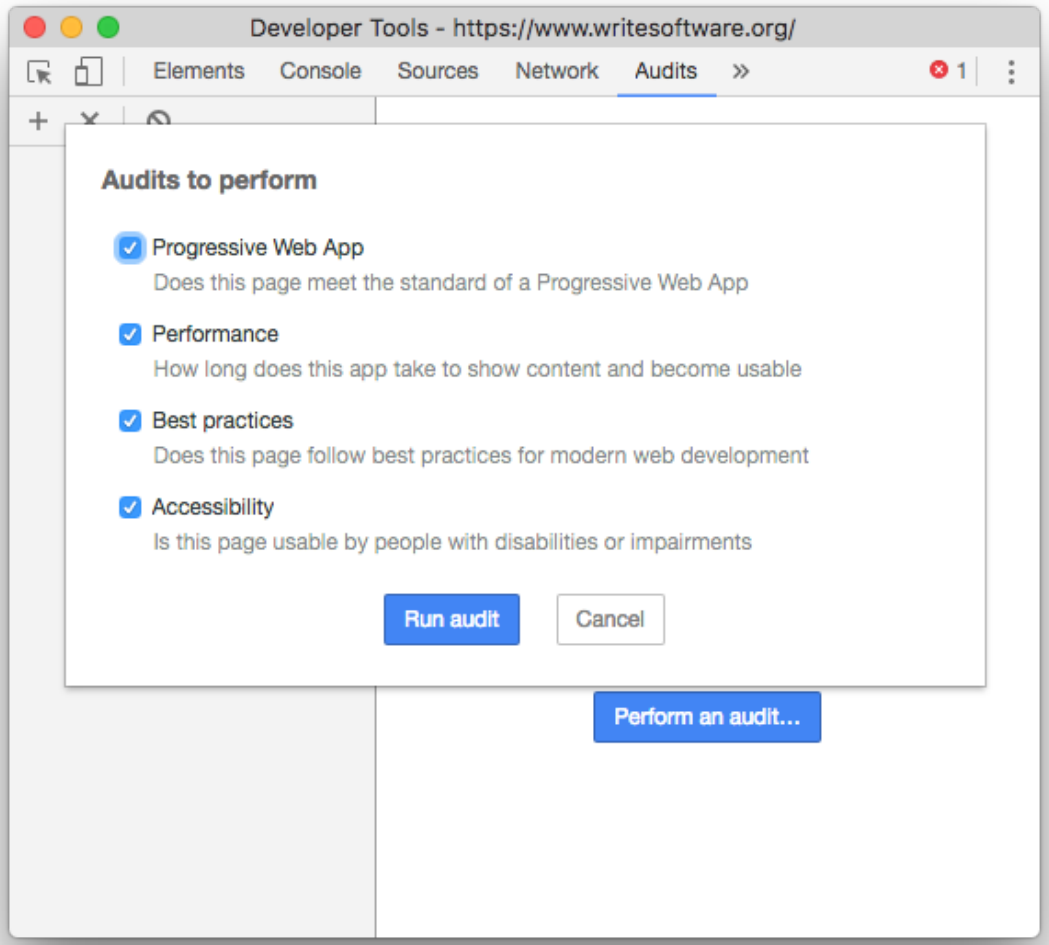


If there is any problem with the HTTPS connection, if the site is served over SSL, it will provide you more information about what's causing it.

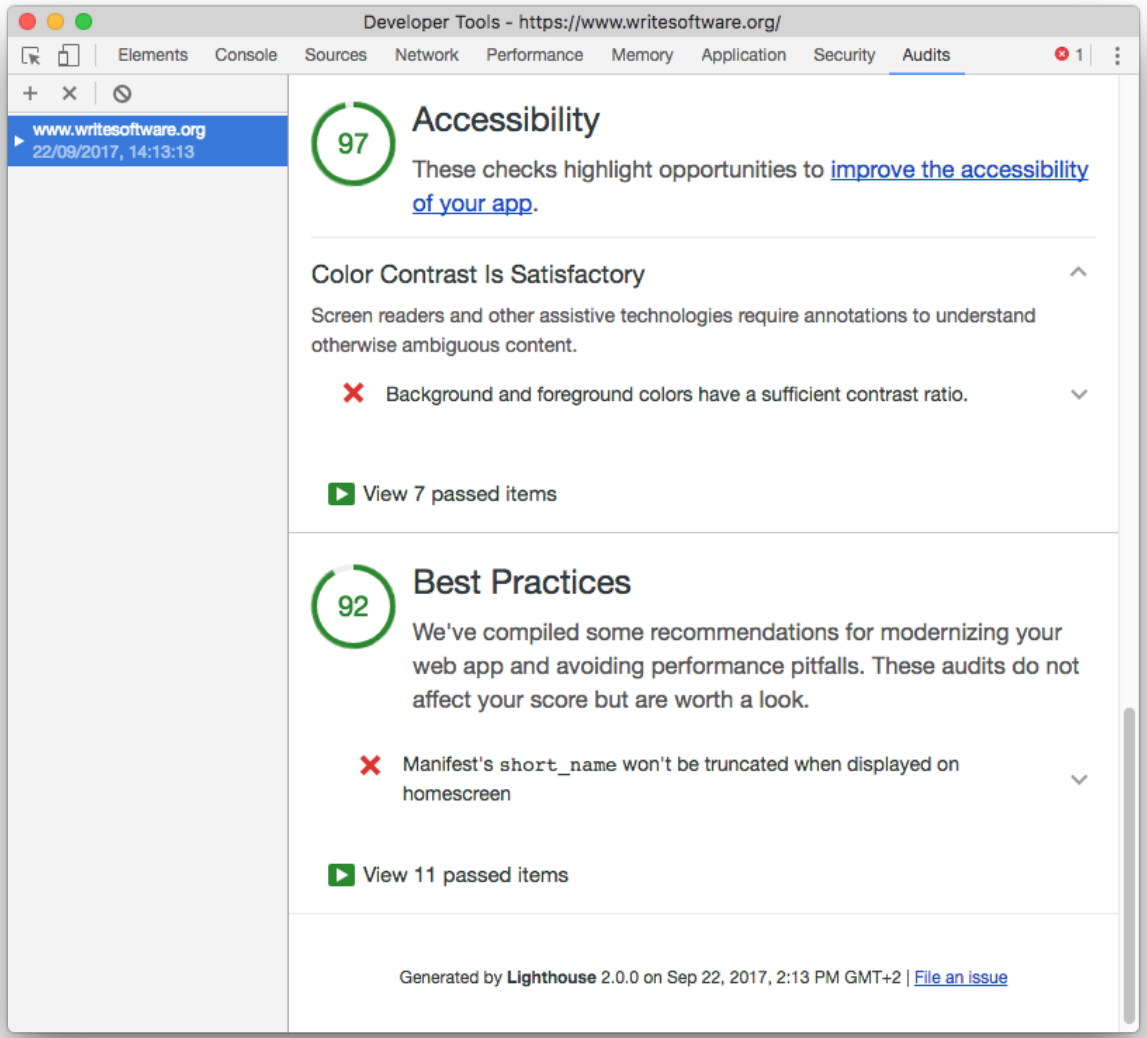
Audits

The Audits tab will help you find and solve some issues relative to performance and in general the quality of the experience that users have when accessing your website.

You can perform various kinds of audits depending on the kind of website:



The audit is provided by Lighthouse (https://developers.google.com/web/tools/lighthouse/) , an open source automated website quality check tool. It takes a while to run, then it provides you a very nice report with key actions to check.



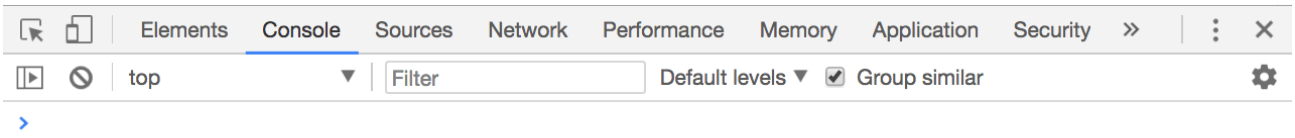
If you want to know more about the Chrome DevTools, check out this Chrome DevTools Tips list 😀

THE CONSOLE API

Every browser exposes a console that lets you interact with the Web Platform APIs and also gives you an inside look at the code by printing messages that are generated by your JavaScript code running in the page

Every browser exposes a console that lets you interact with the Web Platform APIs and also gives you an inside look at the code by printing

messages that are generated by your JavaScript code running in the page.



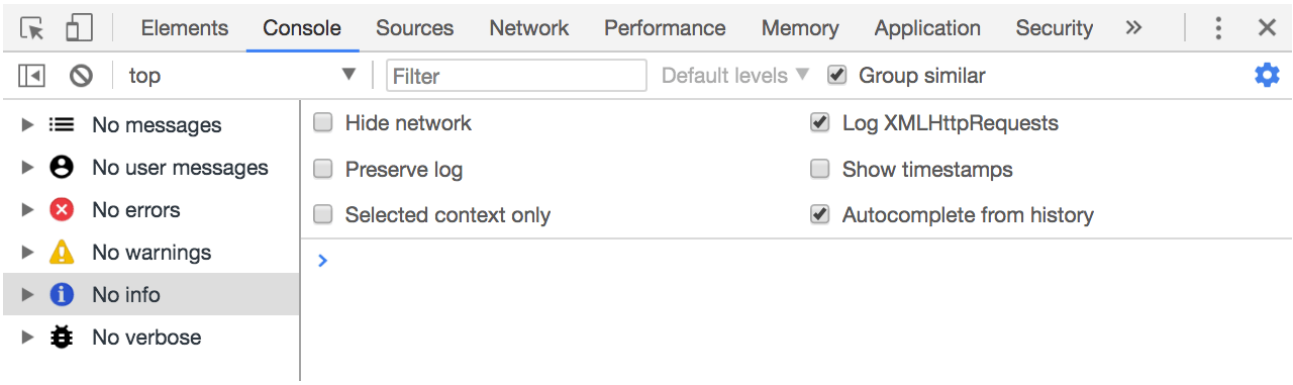
- Overview of the console
- Use console.log formatting
- Clear the console
- Counting elements
- Log more complex objects
- Logging different error levels
- Preserve logs during navigation
- Grouping console messages
- Print the stack trace
- Calculate the time spent
- Generate a CPU profile

Overview of the console

The console toolbar is simple. There's a button to clear the console messages, something you can also do by clicking `cmd-K` in macOS, or `ctrl-K` on Windows, a second button that activates a filtering sidebar,

that lets you filter by text, or by type of message, for example error, warning, info, log, or debug messages.

You can also choose to hide network-generated messages, and just focus on the JavaScript log messages.



The console is not just a place where you can see messages, but also the best way to interact with JavaScript code, and many times the DOM. Or, just get information from the page.

Let's type our first message. Notice the >, let's click there and type

```
console.log('test')
```

The console acts as a **REPL**, which means read–eval–print loop. In short, it interprets our JavaScript code and prints something.

Use console.log formatting

As you see, `console.log('test')` prints 'test' in the Console.

Using `console.log` in your JavaScript code can help you debug for example by printing static strings, but you can also pass it a variable, which can be a JavaScript native type (for example an integer) or an object.

You can pass multiple variables to `console.log`, for example:

```
console.log('test1', 'test2')
```

We can also format pretty phrases by passing variables and a format specifier.

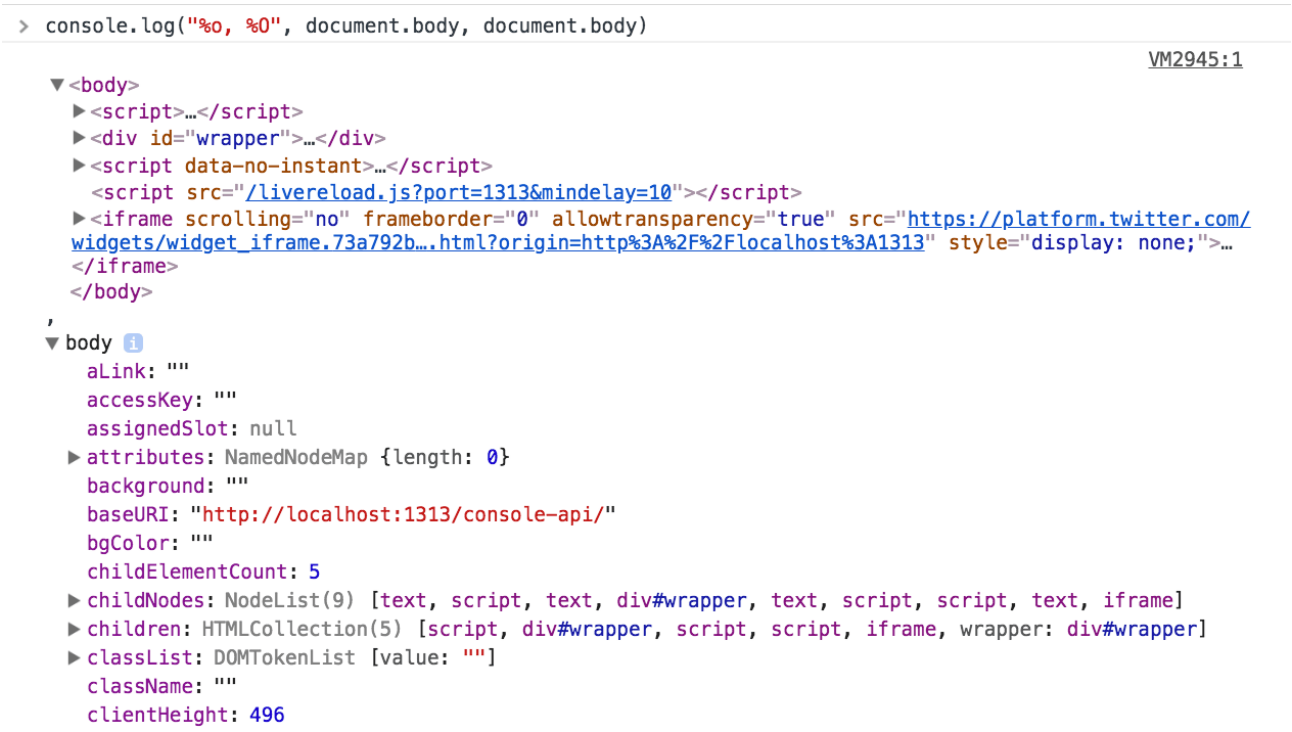
For example:

```
console.log("My %s has %d years", 'cat', 2)
```

- `%s` format a variable as a string
- `%d` or `%i` format a variable as an integer
- `%f` format a variable as a floating point number
- `%o` can be used to print a DOM Element
- `%O` used to print an object representation

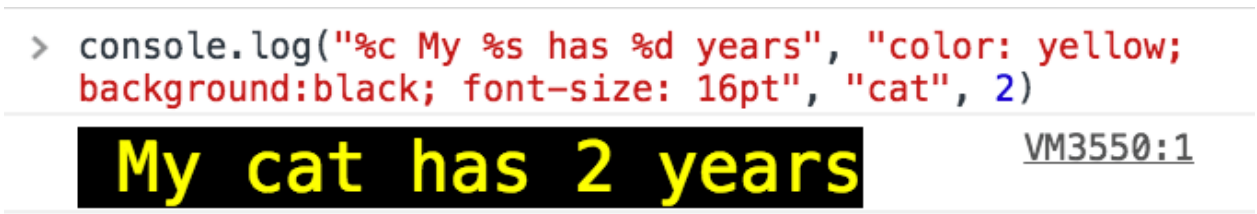
Example:

```
console.log("%o, %O", document.body, document.body)
```



Another useful format specifier is `%c`, which allows to pass CSS to format a string. For example:

```
console.log("%c My %s has %d years", "color: yellow;
background:black; font-size: 16pt", "cat", 2)
```



Clear the console

There are three ways to clear the console while working on it, with various input methods.

The first way is to click the **Clear Console Log** button on the console toolbar.

The second method is to type `console.clear()` inside the console, or in your a JavaScript function that runs in your app / site.

You can also just type `clear()`.

The third way is through a keyboard shortcut, and it's `cmd-k` (mac) or `ctrl + l` (Win)

Counting elements

`console.count()` is a handy method.

Take this code:

```
const x = 1
const y = 2
const z = 3
console.count("The value of x is " + x + " and has been checked ..
how many times?")
console.count("The value of x is " + x + " and has been checked ..
how many times?")
console.count("The value of y is " + y + " and has been checked ..
how many times?")
```

What happens is that count will count the number of times a string is printed, and print the count next to it:

```
> const x = 1
  const y = 2
  const z = 3
  console.count("The value of x is " + x + " and has
  been checked .. how many times?")
  console.count("The value of x is " + x + " and has
  been checked .. how many times?")
  console.count("The value of y is " + y + " and has
  been checked .. how many times?")
```

```
The value of x is 1 and has been checked ..  VM5252:4
how many times?: 1
The value of x is 1 and has been checked ..  VM5252:5
how many times?: 2
The value of y is 2 and has been checked ..  VM5252:6
how many times?: 1
```

You can just count apples and oranges:

```
const oranges = ['orange', 'orange']
const apples = ['just one apple']
oranges.forEach((fruit) => {
  console.count(fruit)
})
apples.forEach((fruit) => {
  console.count(fruit)
})
```

```
> const oranges = ['orange', 'orange']
  const apples = ['just one apple']
  oranges.forEach((fruit) => {
    console.count(fruit)
  })
  apples.forEach((fruit) => {
    console.count(fruit)
  })
```

```
orange: 1                                    VM5502:4
orange: 2                                    VM5502:4
just one apple: 1                            VM5502:7
```

Log more complex objects

`console.log` is pretty amazing to inspect variables. You can pass it an object too, and it will do its best to print it to you in a readable way. Most of the times this means it prints a string representation of the object.

For example try

```
console.log([1, 2])
```

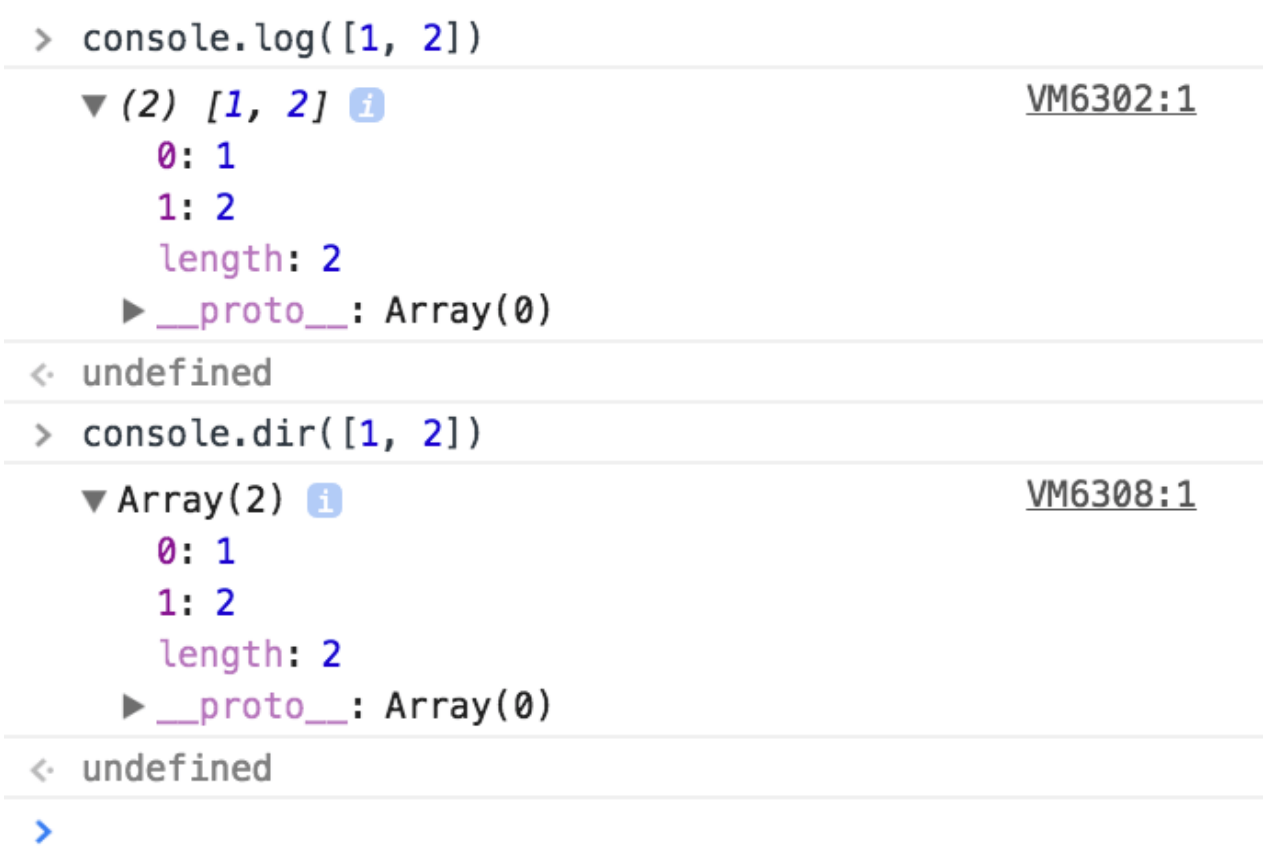
Another option to print objects is to use `console.dir`:

```
console.dir([1, 2])
```

As you can see this method prints the variable in a JSON-like representation, so you can inspect all its properties.

The same thing that console.dir outputs is achievable by doing

```
console.log("%O", [1,2])
```



Which one to use depends on what you need to debug of course, and one of the two can do the best job for you.

Another function is `console.table()` which prints a nice table.

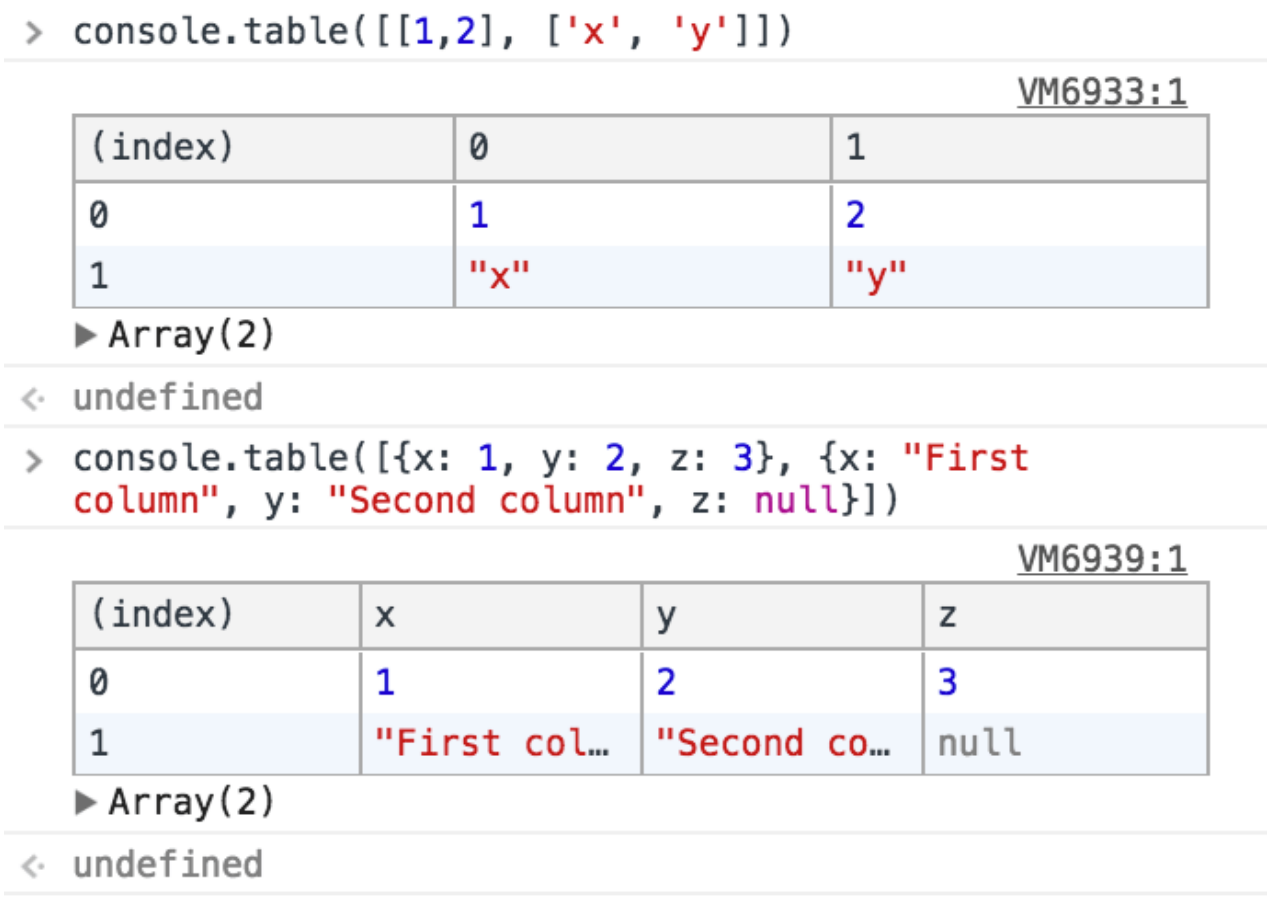
We just need to pass it an array of elements, and it will print each array item in a new row.

For example

```
console.table([[1,2], ['x', 'y']])
```

or you can also set column names, by passing instead of an array, an Object Literal, so it will use the object property as the column name

```
console.table([{x: 1, y: 2, z: 3}, {x: "First column", y: "Second
column", z: null}])
```



`console.table` can also be more powerful and if you pass it an object literal that in turn contains an object, and you pass an array with the column names, it will print a table with the row indexes taken from the object literal. For example:

```
const shoppingCart = {}
shoppingCart.firstItem = {'color': 'black', 'size': 'L'}
shoppingCart.secondItem = {'color': 'red', 'size': 'L'}
shoppingCart.thirdItem = {'color': 'white', 'size': 'M'}
console.table(shoppingCart, ["color", "size"])
```

```
> const shoppingCart = {}
shoppingCart.firstItem = {'color': 'black', 'size':
'L'}
shoppingCart.secondItem = {'color': 'red', 'size':
'L'}
shoppingCart.thirdItem = {'color': 'white', 'size':
'M'}
console.table(shoppingCart, ["color", "size"])
```

VM7490:5

(index)	color	size
firstItem	"black"	"L"
secondItem	"red"	"L"
thirdItem	"white"	"M"

▶ Object

```
<· undefined
```

Logging different error levels

As we saw console.log is great for printing messages in the Console.

We'll now discover three more handy methods that will help us debug, because they implicitly indicate various levels of error.

First, **console.info()**

As you can see a little 'i' is printed beside it, making it clear the log message is just an information.

Second, **console.warn()**

prints a yellow exclamation point.

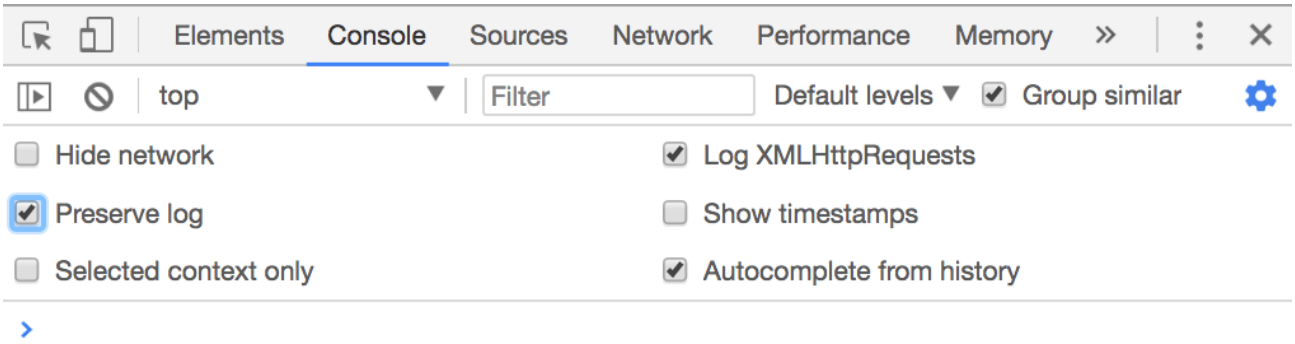
If you activate the Console filtering toolbar, you can see that the Console allows you to filter messages based on the type, so it's really convenient to differentiate messages because for example if we now click 'Warnings', all the printed messages that are not warnings will be hidden.

The third function is **console.error()**

this is a bit different than the others because in addition to printing a red X which clearly states there's an error, we have the full stack trace of the function that generated the error, so we can go and try to fix it.

Preserve logs during navigation

Console messages are cleared on every page navigation, unless you check the **Preserve log** in the console settings:



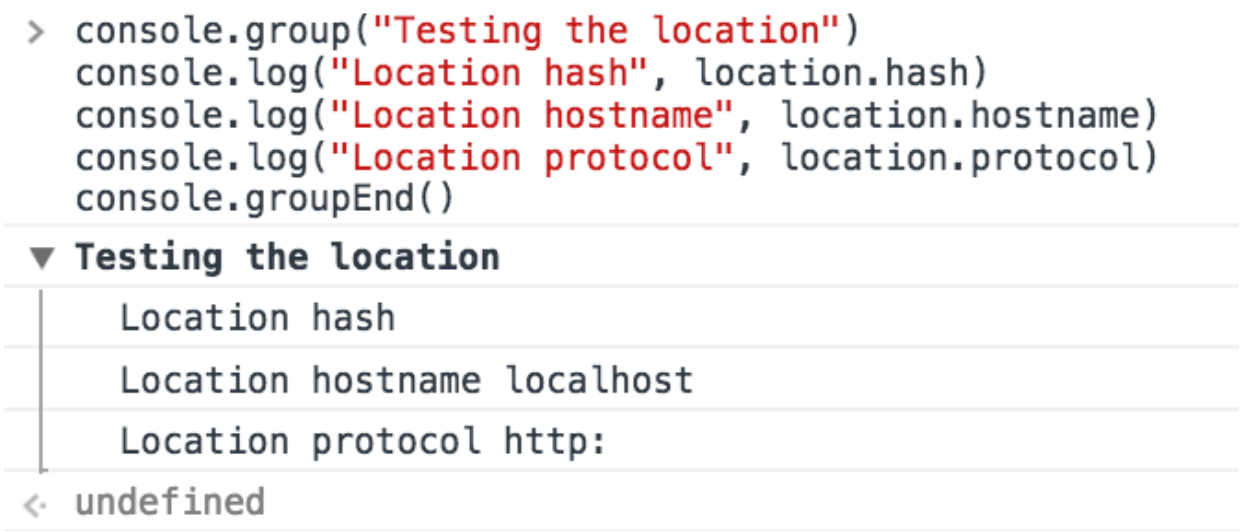
Grouping console messages

The Console messages can grow in size and the noise when you're trying to debug an error can be overwhelming.

To limit this problem the Console API offers a handy feature: Grouping the Console messages.

Let's do an example first.

```
console.group("Testing the location")
console.log("Location hash", location.hash)
console.log("Location hostname", location.hostname)
console.log("Location protocol", location.protocol)
console.groupEnd()
```



As you can see the Console creates a group, and there we have the Log messages.

You can do the same, but output a collapsed message that you can open on demand, to further limit the noise:

```
console.groupCollapsed("Testing the location")
console.log("Location hash", location.hash)
console.log("Location hostname", location.hostname)
console.log("Location protocol", location.protocol)
console.groupEnd()
```

The nice thing is that those groups can be nested, so you can end up doing

```
console.group("Main")
console.log("Test")
```

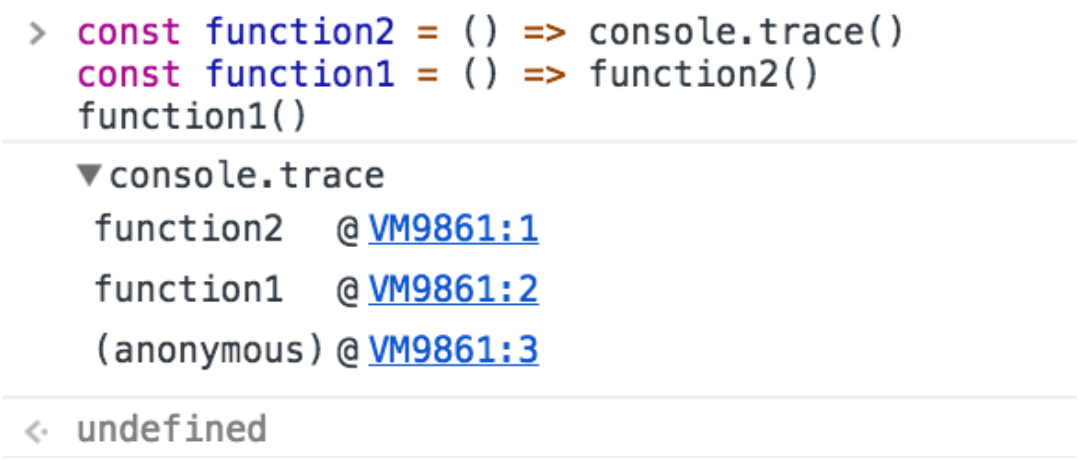
```
console.group("1")
console.log("1 text")
console.group("1a")
console.log("1a text")
console.groupEnd()
console.groupCollapsed("1b")
console.log("1b text")
console.groupEnd()
console.groupEnd()
```

Print the stack trace

There might be cases where it's useful to print the call stack trace of a function, maybe to answer the question *how did you reach that part of code?*

You can do so using `console.trace()`:

```
const function2 = () => console.trace()
const function1 = () => function2()
function1()
```



Calculate the time spent

You can easily calculate how much time a function takes to run, using `time()` and `timeEnd()`

```
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
  console.time('doSomething()')
  //do something, and measure the time it takes
  doSomething()
  console.timeEnd('doSomething()')
}
measureDoingSomething()
```

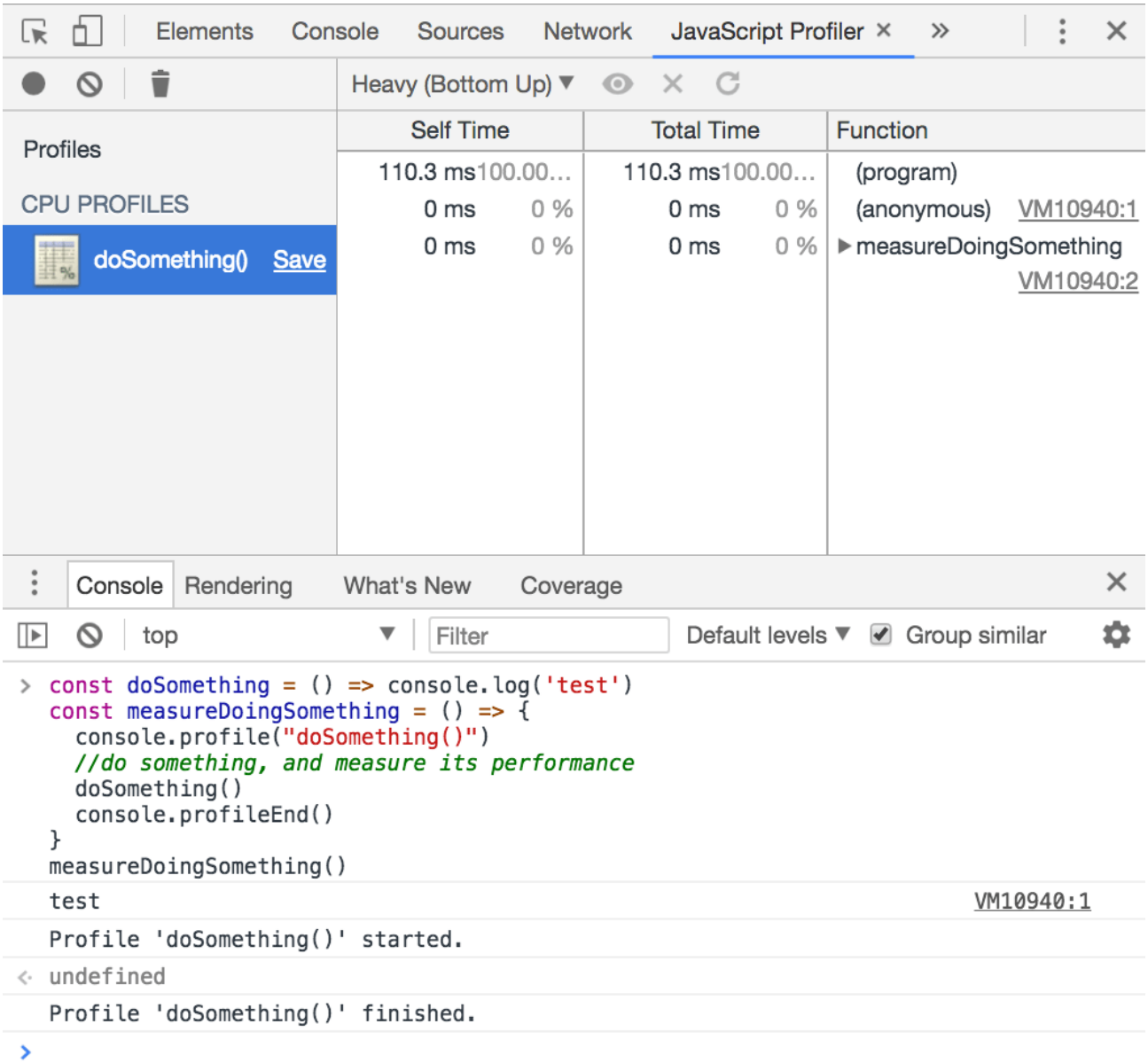
```
> const doSomething = () => console.log('test')
  const measureDoingSomething = () => {
    console.time('doSomething()')
    //do something, and measure the time it takes
    doSomething()
    console.timeEnd('doSomething()')
  }
  measureDoingSomething()
  test
  doSomething(): 0.530029296875ms
```

Generate a CPU profile

The DevTools allow you to analyze the CPU profile performance of any function.

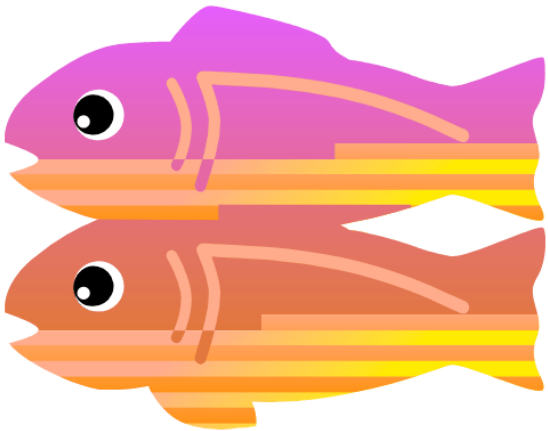
You can start that manually, but the most accurate way to do so is to wrap what you want to monitor between the `profile()` and `profileEnd()` commands. They are similar to `time()` and `timeEnd()`, except they don't just measure time, but create a more detailed report.

```
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
  console.profile("doSomething()")
  //do something, and measure its performance
  doSomething()
  console.profileEnd()
}
measureDoingSomething()
```


Elements
Console
Sources
Network
JavaScript Profiler
Heavy (Bottom Up)
Profiles
CPU PROFILES
doSomething()
Save
Self Time
Total Time
Function
110.3 ms 100.00...
110.3 ms 100.00...
(program)
0 ms 0 %
0 ms 0 %
(anonymous) VM10940:1
0 ms 0 %
0 ms 0 %
measureDoingSomething
VM10940:2
Console
Rendering
What's New
Coverage
top
Filter
Default levels
Group similar
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
console.profile("doSomething()")
//do something, and measure its performance
doSomething()
console.profileEnd()
}
measureDoingSomething()
test VM10940:1
Profile 'doSomething()' started.
undefined
Profile 'doSomething()' finished.

GLITCH, A GREAT PLATFORM FOR DEVELOPERS

Glitch is a pretty amazing platform to learn and experiment with code. This post introduces you to Glitch and makes you go from zero to hero



Glitch (https://glitch.com/) is a great platform to learn how to code.

I use Glitch on many of my tutorials, I think it's a **great tool** to **showcase** concepts, and also **allow people to use your projects** and build upon

them.

Here is an example project I made on Glitch with React and React Router: https://glitch.com/edit/#!/flaviocopes-react-router-v4

With Glitch you can easily create **demos** and **prototypes** of applications written in JavaScript, from simple web pages to advanced frameworks such as React or Vue, and server-side Node.js apps.

It is built on top of **Node**, and you have the ability to install any npm package you want, run webpack and much more.

It's brought to you by the people that made some hugely successful products, including Trello and Stack Overflow, so it has a lot of credibility bonuses for that.

Why do I think Glitch is great?

Glitch "clicked" for me in how it presents itself in a **funny** interface, but **not dumbed down**.

You have access to **logs**, the **console**, and lots of internal stuff.

Also, the concept of remixing so prominent in the interface makes me much more likely to create lots of projects there, as I never have to start from a clean slate.

You can start diving into the code **without losing time** setting up an environment, version control, and focus on the idea, with an automatic

HTTPS URL and a **CDN** for the media assets.

Also, there's no lock-in at all, it's just Node.js (or if you don't use server-side JavaScript, it's just HTML, JS and CSS)

Is it free?

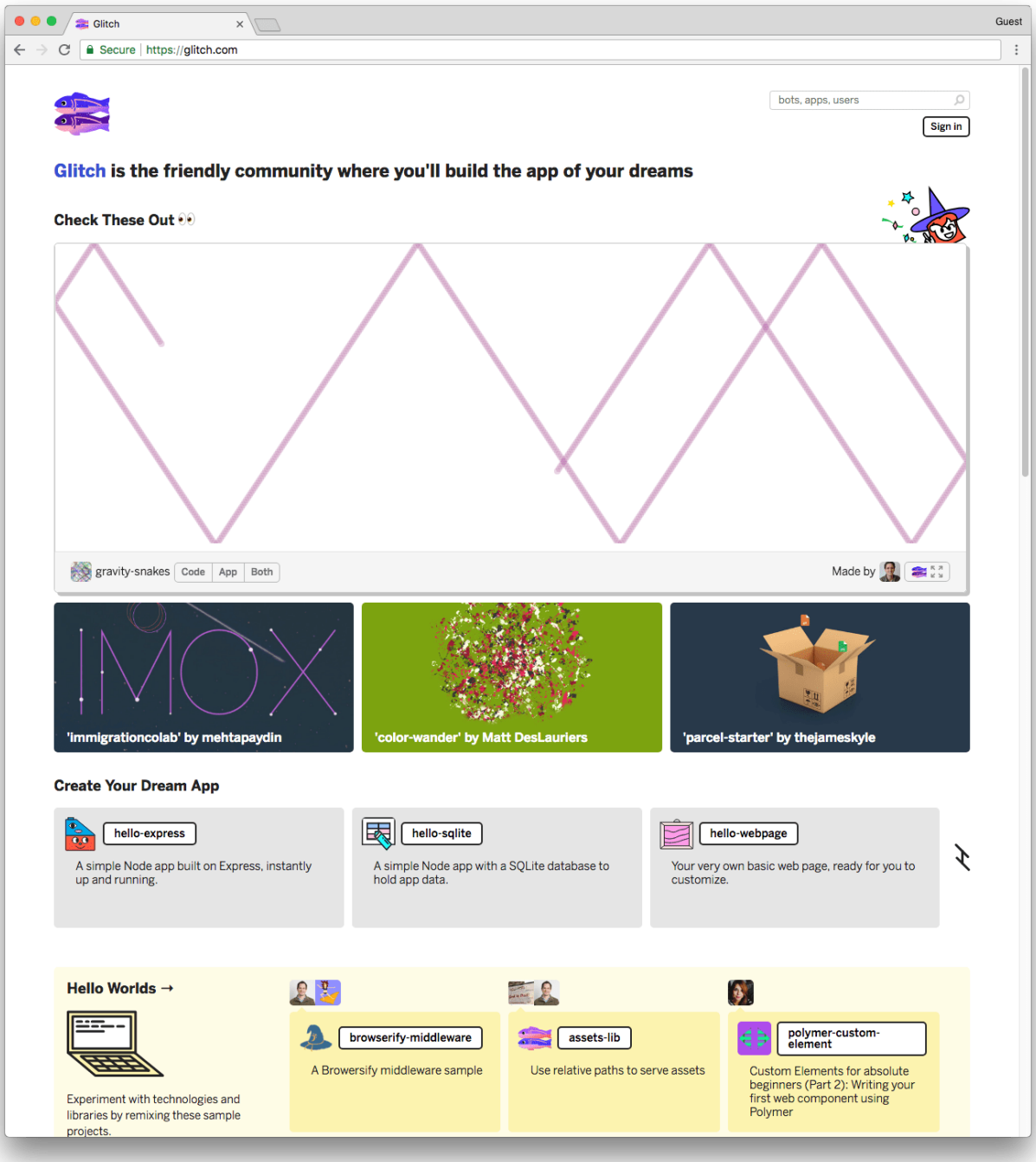
Yes, it's free, and in the future they might add even more features on top for a paid plan, but they state that the current Glitch will always be free as it is now.

There are reasonable limits like

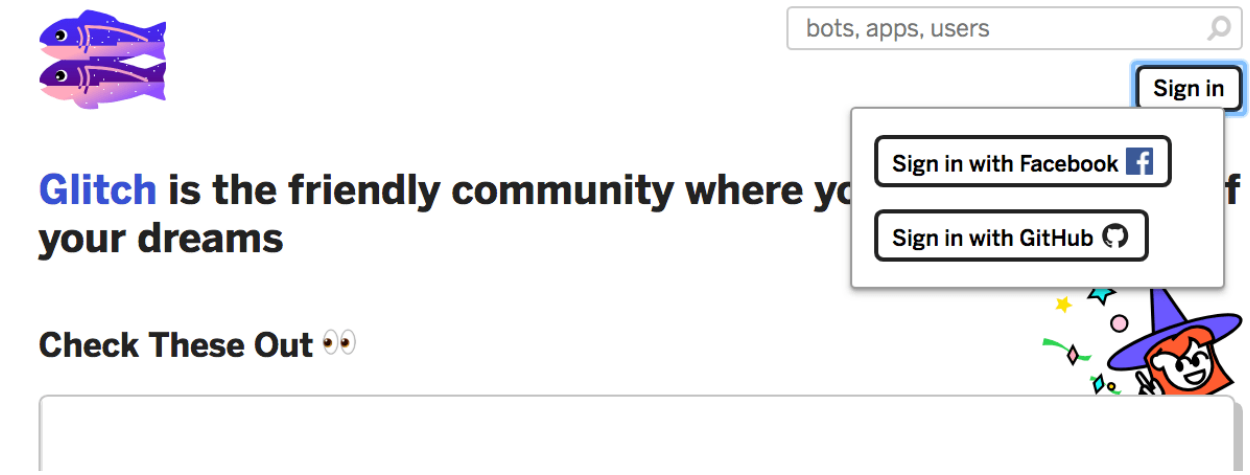
- You have 128MB of space, excluding npm packages, plus 512MB for media assets
- You can serve up to 4000 requests per hour
- Apps are stopped if not accessed for 5 minutes and they do not receive any HTTP request, and long running apps are stopped after 12 hours. As soon as an HTTP request comes in, they start again

An overview of Glitch

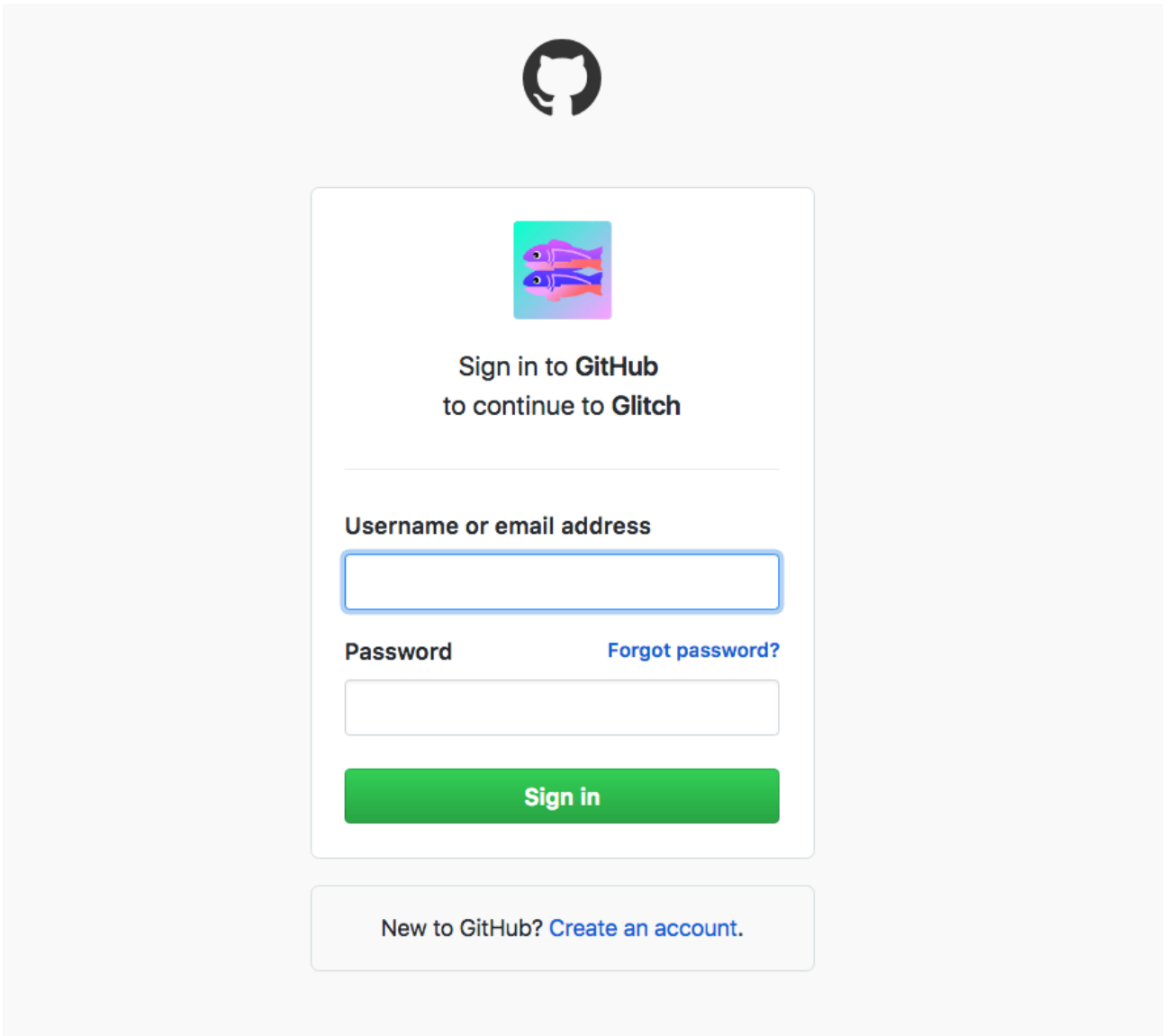
This is the Glitch homepage, it shows a few projects that they decided to showcase because they are cool, and some starter projects:



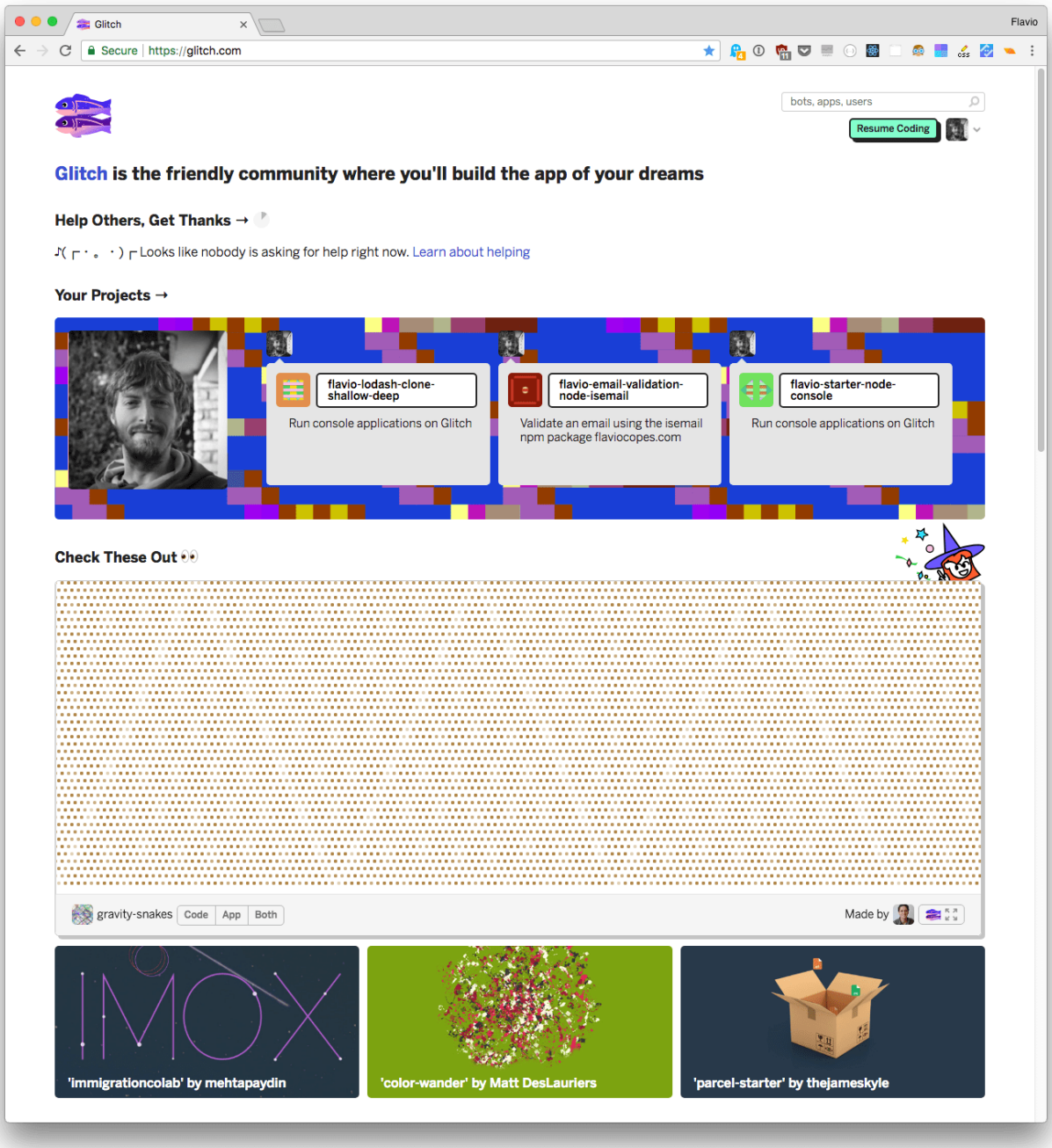
Creating an account is free and easy, just press “Sign in” and choose between Facebook and GitHub as your “entry points” (I recommend GitHub):



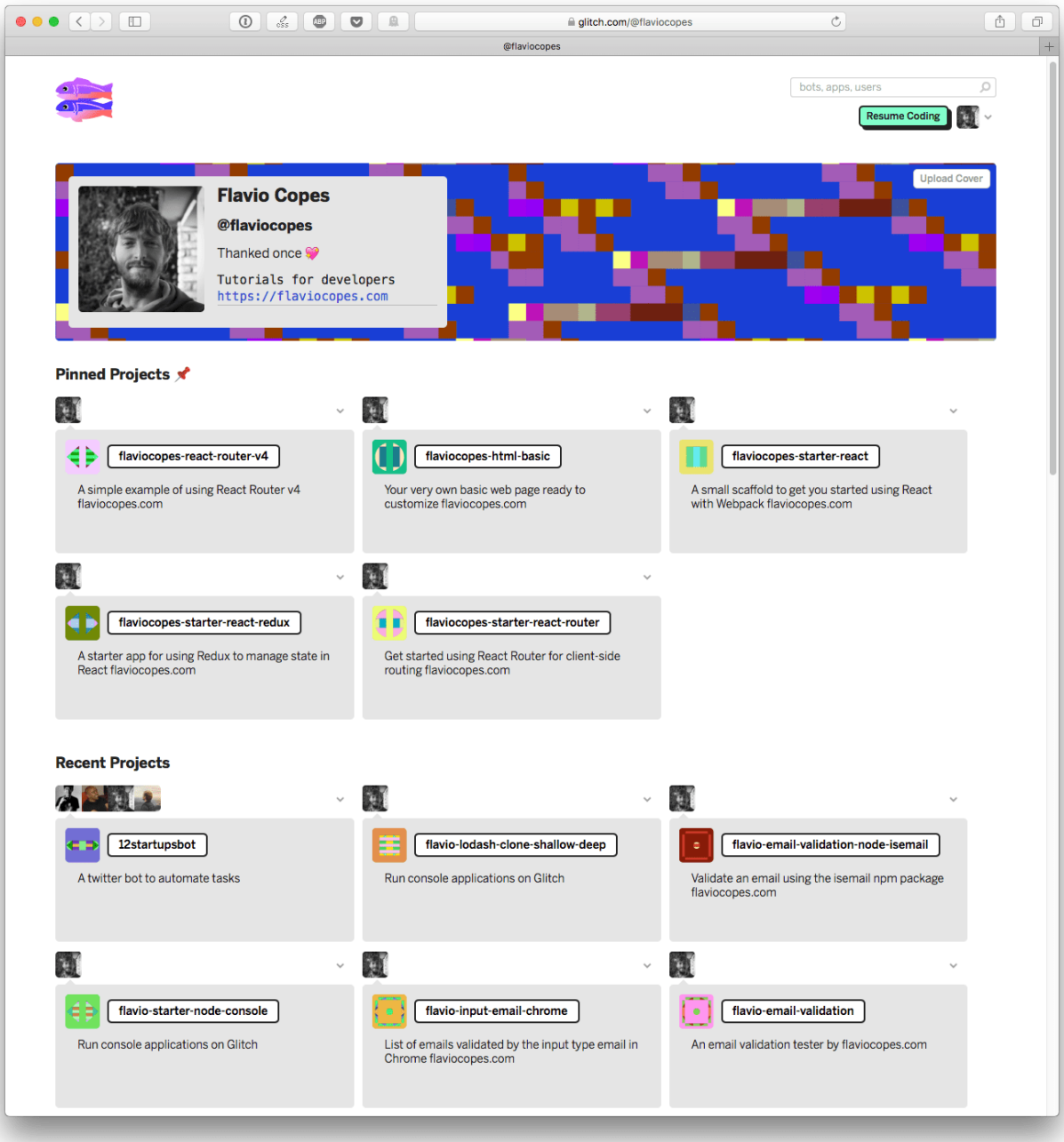
You are redirected to GitHub to authorize:



Once logged in, the home page changes to also show your projects:



Clicking **Your Projects** sends you to your profile page, which has your name in the URL. Mine is https://glitch.com/@flaviocopes.



You can **pin** projects, to find them more easily when you'll have lots of them.

The concept of remixing

When you first start, of course you will have no projects of your own.

Glitch makes it super easy to start, and you never start from a blank project. **You always *remix* another project**.

You can remix a project you like, maybe one you found on Twitter or featured in the Glitch homepage, or you can start from a project that's a boilerplate to start something:

- A simple web page (https://glitch.com/~hello-webpage)
- Node.js Express app (https://glitch.com/~hello-express)
- A Node.js console (https://glitch.com/~node-console)
- A Create-React-App app (https://glitch.com/~create-react-app-sample)
- A Nuxt starter app (https://glitch.com/~nuxt-starter)

There are many other starter glitches in these collections:

- Hello World Glitches (https://glitch.com/hello-worlds)
- Building Blocks (https://glitch.com/building-blocks)

If you're learning to code right now, the Learn to Code glitch Collection (https://glitch.com/learn-to-code) is very nice.

I have created a few starter apps that I constantly use for my demos and tests, and they are:

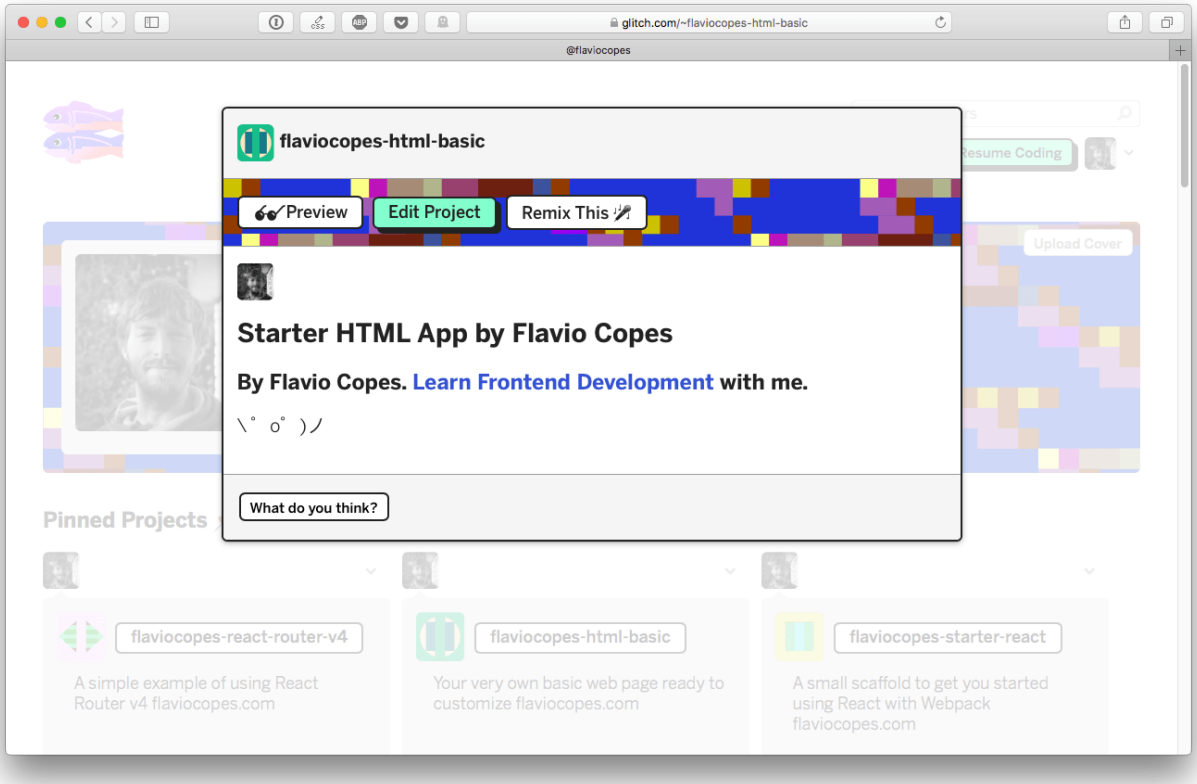
- Simple HTML + CSS + JS glitch (https://glitch.com/~flaviocopes-html-basic)
- React + webpack starter glitch (https://glitch.com/~flaviocopes-starter-react)

Glitch makes it very easy to create your own building blocks, and by pinning them in your profile, you can have them always on the top, easy to

find.

Remix a glitch

Once you have a glitch you want to build upon, you just click it, and a window shows up:

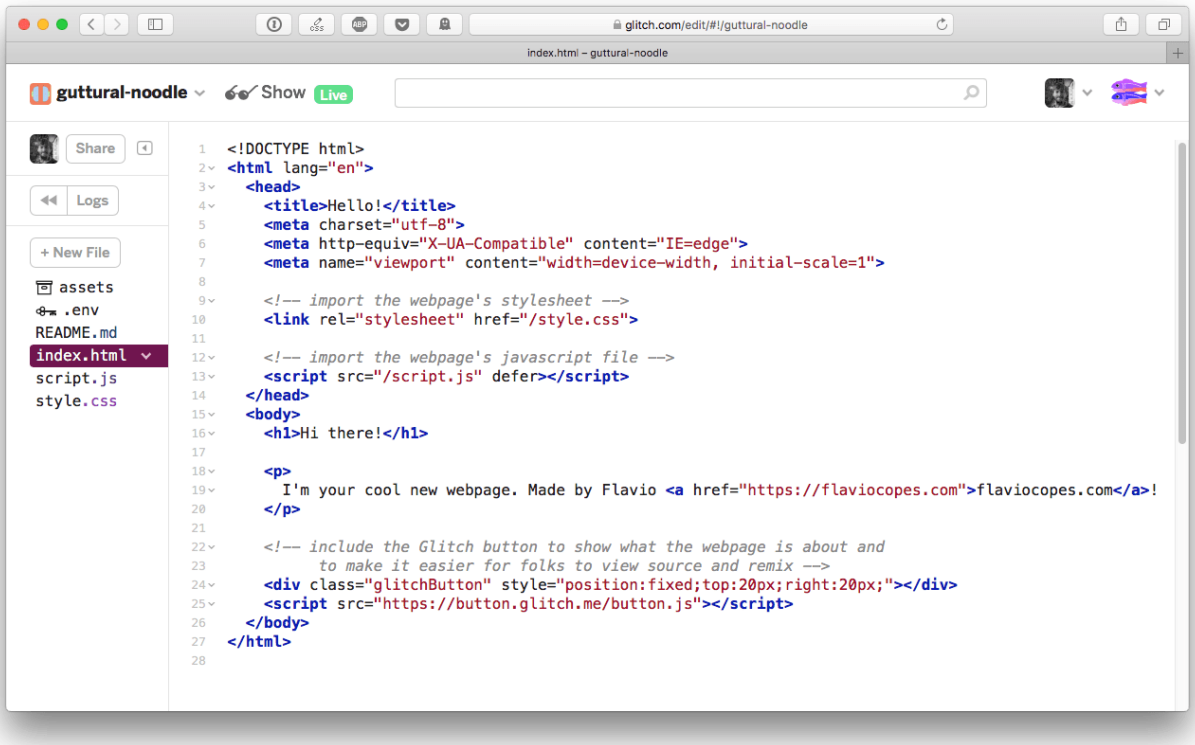


There are 3 buttons:

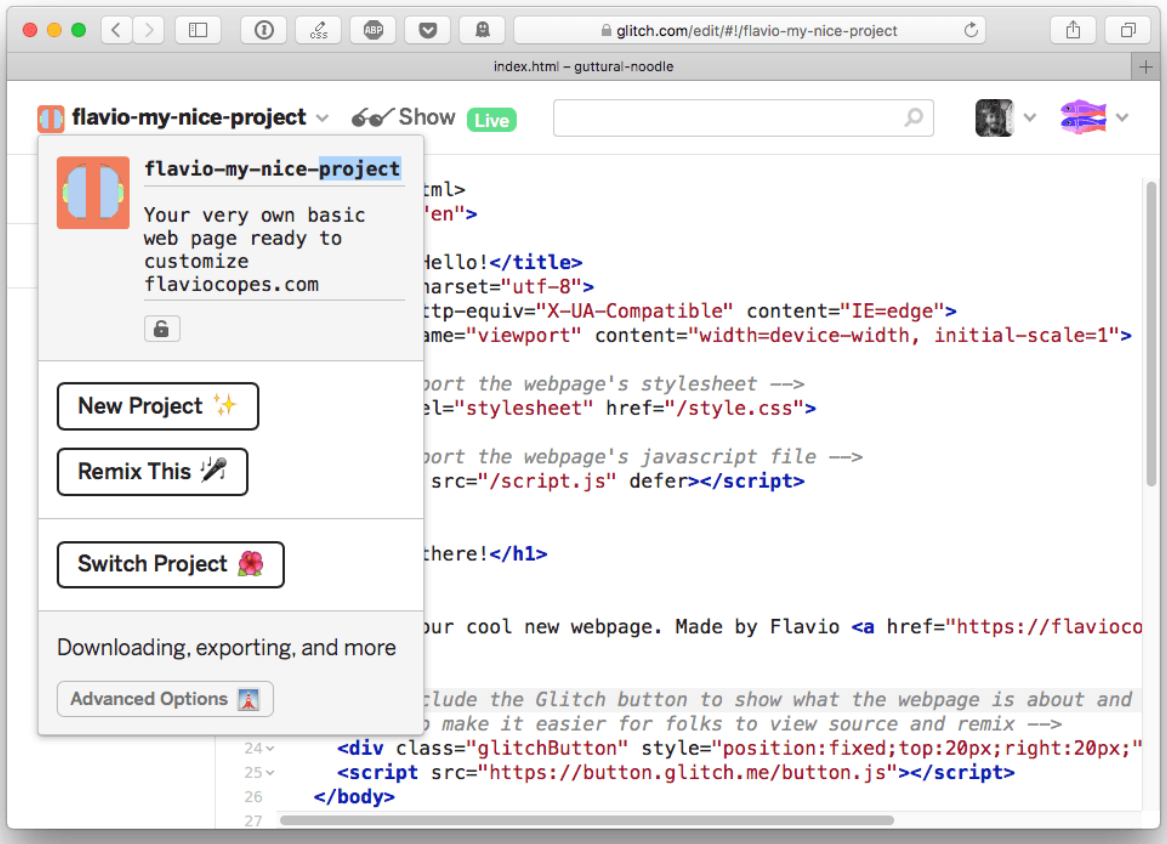
- **Preview** a glitch is code that does something. This shows the result of the glitch.
- **Edit Project** shows the source of the project, and you can start editing it
- **Remix This** clones the glitch to a new one

Every time you remix a glitch, a new project is created, with a random name.

Here is a glitch right after creating it by remixing another one:



Glitch gave it the name `guttural-noodle`. Clicking the name you can change it:

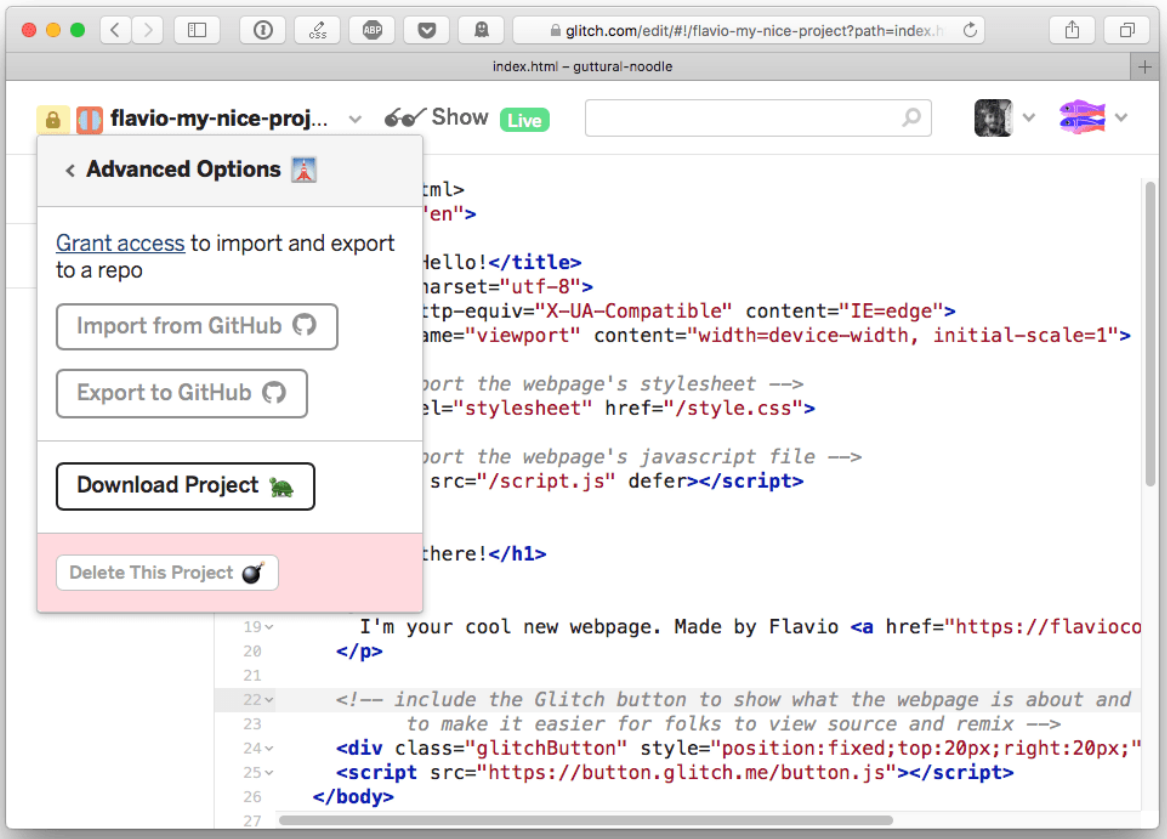


You can also change the description.

From here you can also create a new glitch from zero, remix the current glitch, or go to another one.

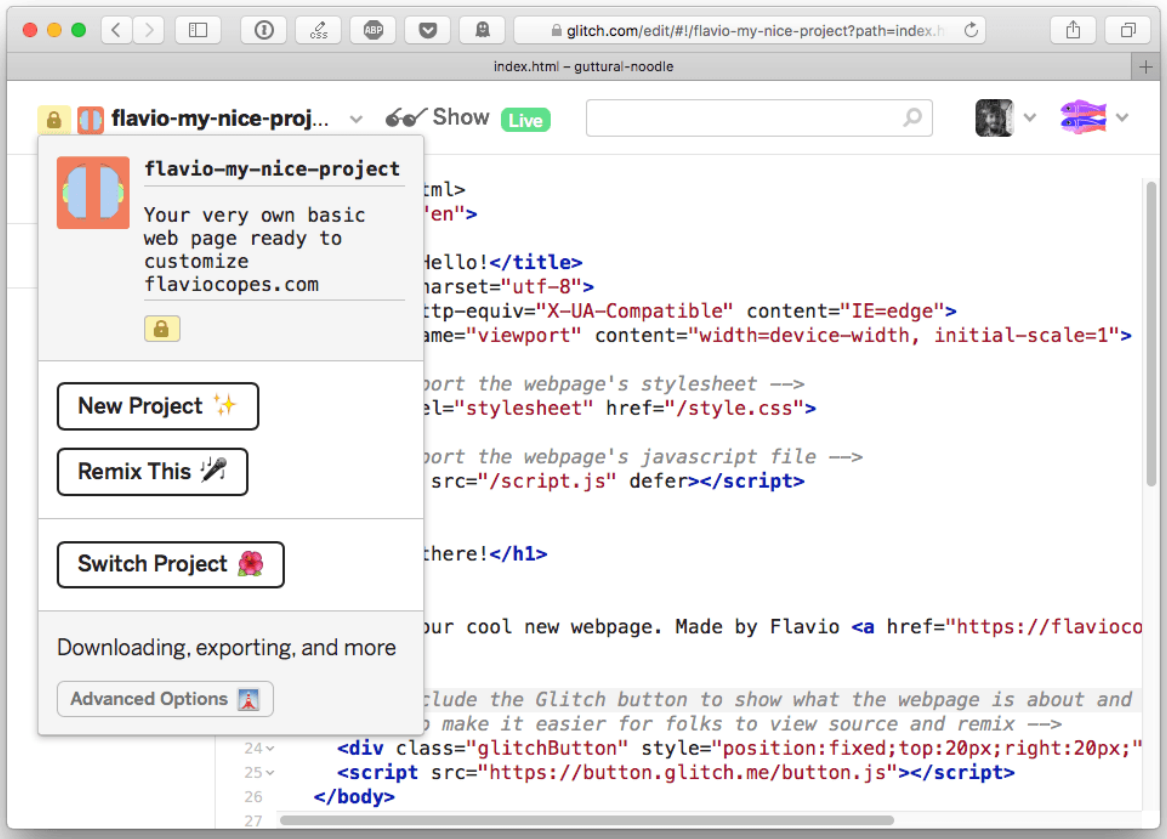
GitHub import/export

There is an easy import/export from/to GitHub, which is very convenient:



Keep your project private

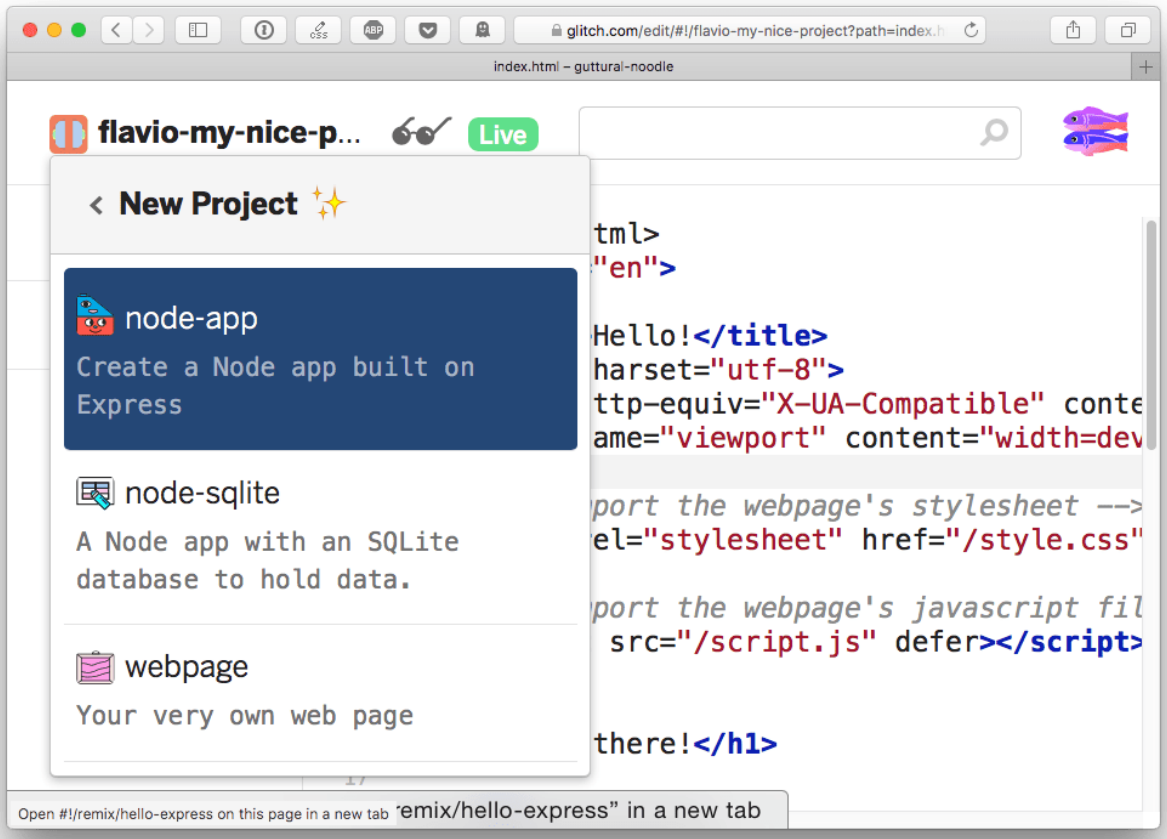
Clicking the lock makes the glitch private:



Create a new project

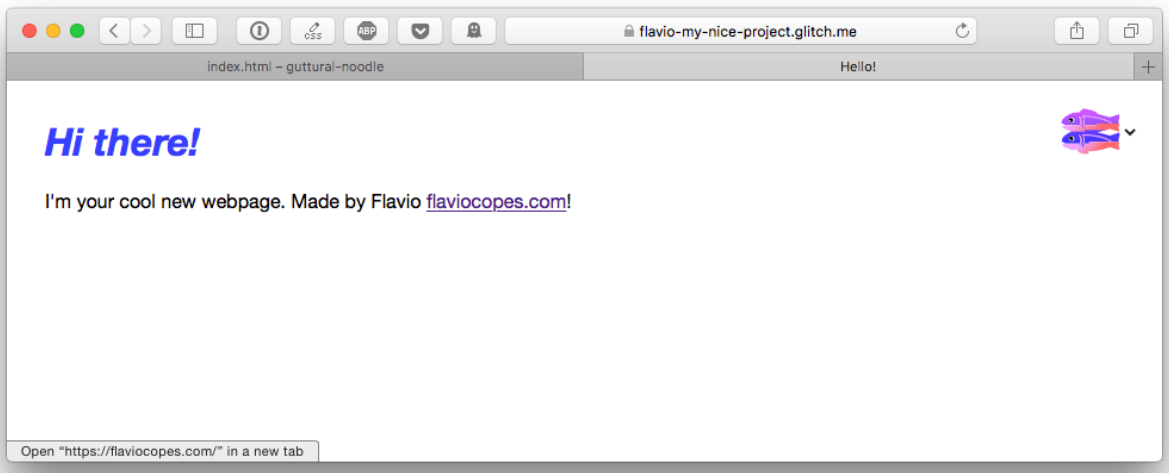
Clicking “New Project” shows 3 options:

- node-app
- node-sqlite
- webpage



This is a shortcut to going out to find those starter apps, and remix them. Under the hoods, clicking one of those options remixes an existing glitch.

On any glitch, clicking "Show" will open a new tab where the app is run:



App URL

Notice the URL, it's:

```
https://flavio-my-nice-project.glitch.me
```

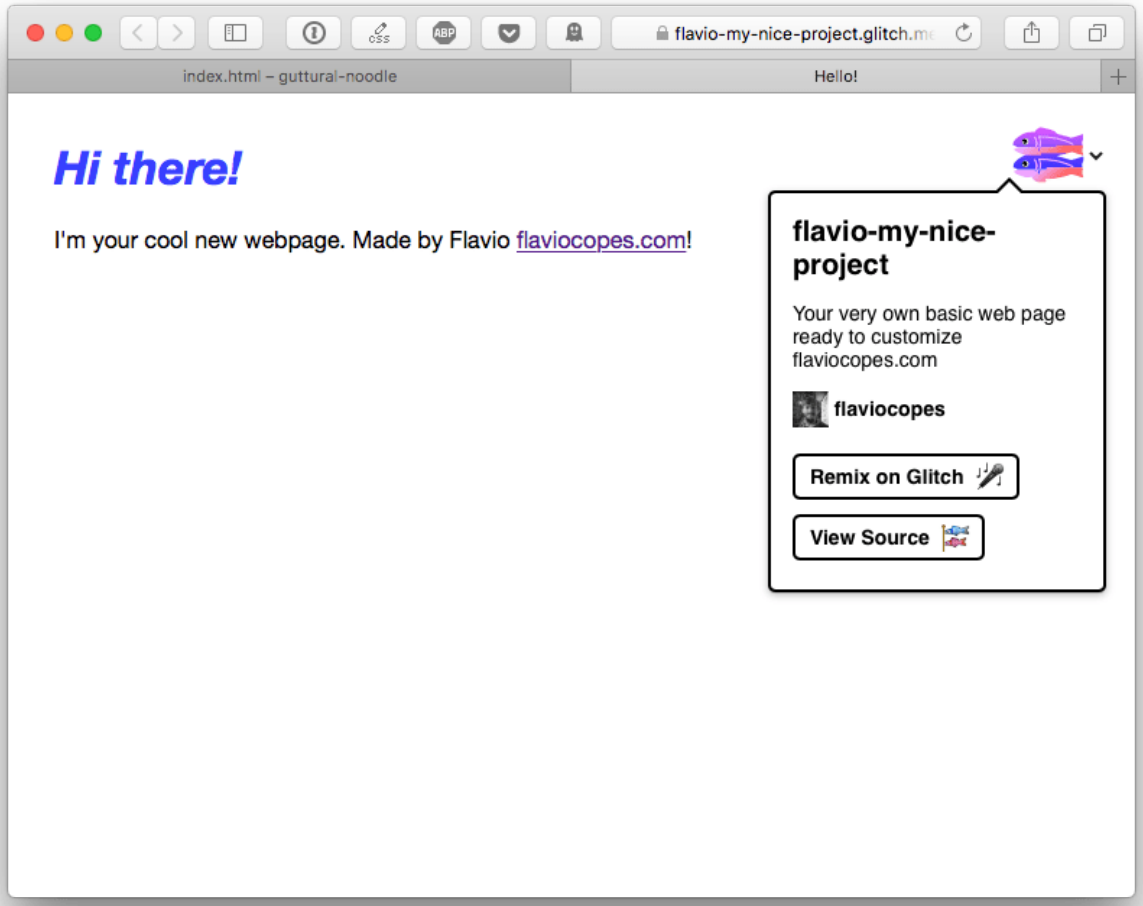
That reflects the app name.

The editing URL is a bit different:

```
https://glitch.com/edit/#!/flavio-my-nice-project
```

The preview runs on a subdomain of `glitch.me`, while editing is done on `glitch.com`.

Noticed the fishes on the right of the page? It's a little JavaScript that Glitch recommend to add to the page, to let other people remix the project or see the source:



Running the app

Any time you make a change to the source, the app is rebuilt, and the live view is refreshed.

This is so convenient, real time applying changes gives an immediate feedback that's a great help when developing.

Secrets

You don't want any API key or password that might be used in the code to be seen by everyone. Any of those secret strings must be put in the special `.env` file, which has a key next to it.

If you invite collaborators, they will be able to see the content, as they are part of the project.

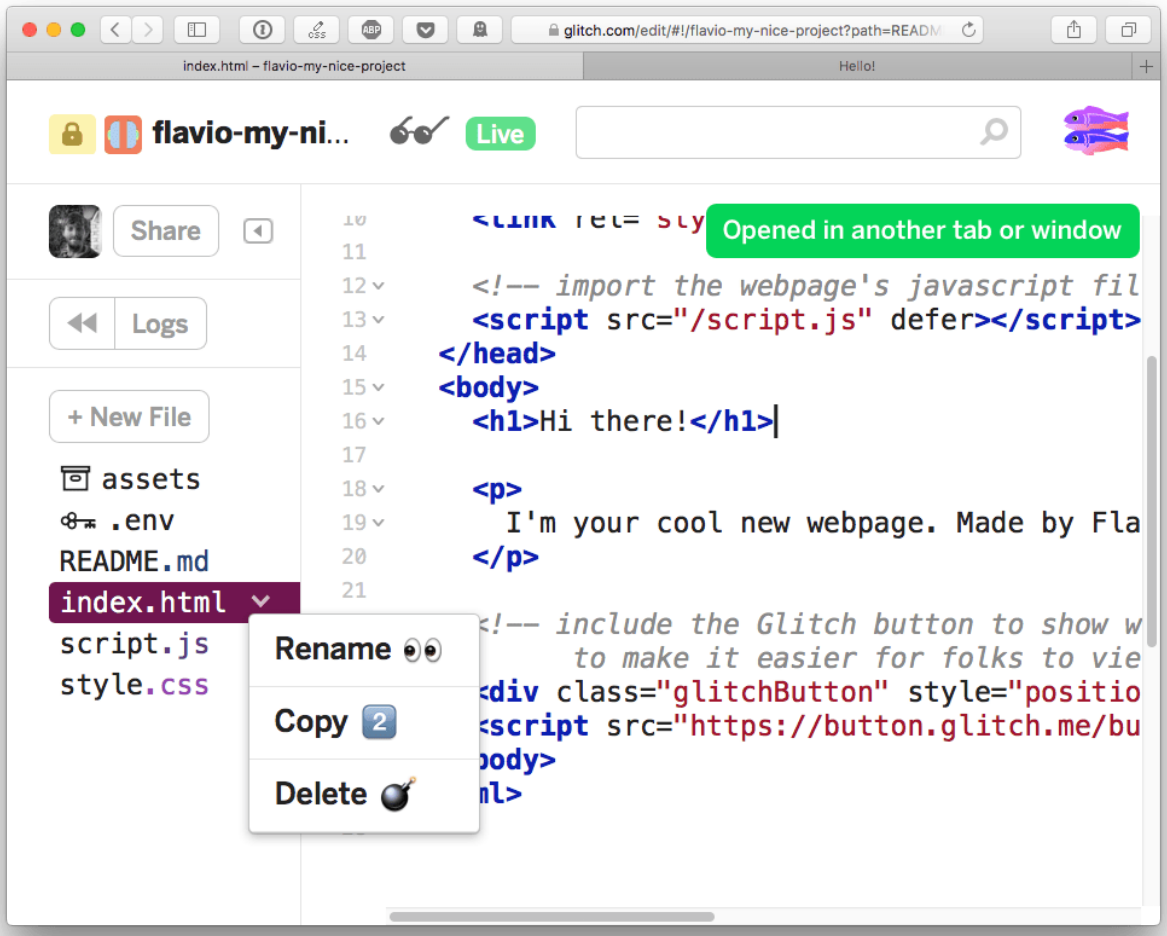
But anyone remixing it, or people invited by you to help, will not see the file content.

Managing files

Adding a new file to a project is easy.

You can **drag and drop files and folders from your local computer**, or click the "**New File**" button above the files list.

It's also intuitive how to rename, copy or delete files:



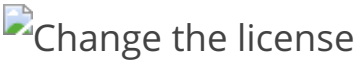
One-click license and code of conduct

Having a license in the code is one of the things that's overlooked in sample projects, but determines what others can do, or can't do, with your project. Without a license, a project is not open source, and all rights are reserved, so the code cannot be redistributed, and other people cannot do anything with it (note: this is my understanding and IANAL - I Am Not A Lawyer).

Glitch makes it *super easy* to add a license, in the **New File** panel:

Add a license

You can easily change it as well:



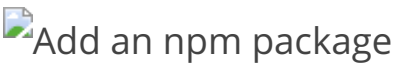
The code of conduct is another very important piece for any project and community. It makes contributors feel welcomed and protected in their participation to the community.

The **Add Code of Conduct** button adds a sample code of conduct for open source projects you can start from.

Adding an npm package

Click the `package.json` file, and if you don't have one yet, create an empty one.

Click the **Add Package** button that now appears on top, and you can add new package.



Also, if you have a package that needs to be updated, Glitch will show the number of packages that need an update, and you can update them to the latest release with a simple click:

Use a custom version of Node.js

You can set the Node.js version to any of these (https://nodeversions.glitch.me/) in your `package.json`. Using `.x` will use the latest release of a major version, which is the most useful thing, like this:

```
{
  //...
  "engines": {
    "node": "8.x"
  }
}
```

Storage

Glitch has a persistent file system. Files are kept on disk even if your app is stopped, or you don't use if for a long time.

This allows you to store data on disk, using local databases or file-based storage (flat-file).

If you put your data in the `.data` folder, this special name indicates the content will not be copied to a new project with the glitch is remixed.

Embedding a glitch in a page

Key to using Glitch to create tutorials is the ability to embed the code and the presentation view, in a page.

Click **Share** and **Embed Project** to open the Embed Project view. From there you can choose to only embed the code, the app, or customize the height of the widget - and get its HTML code to put on your site:

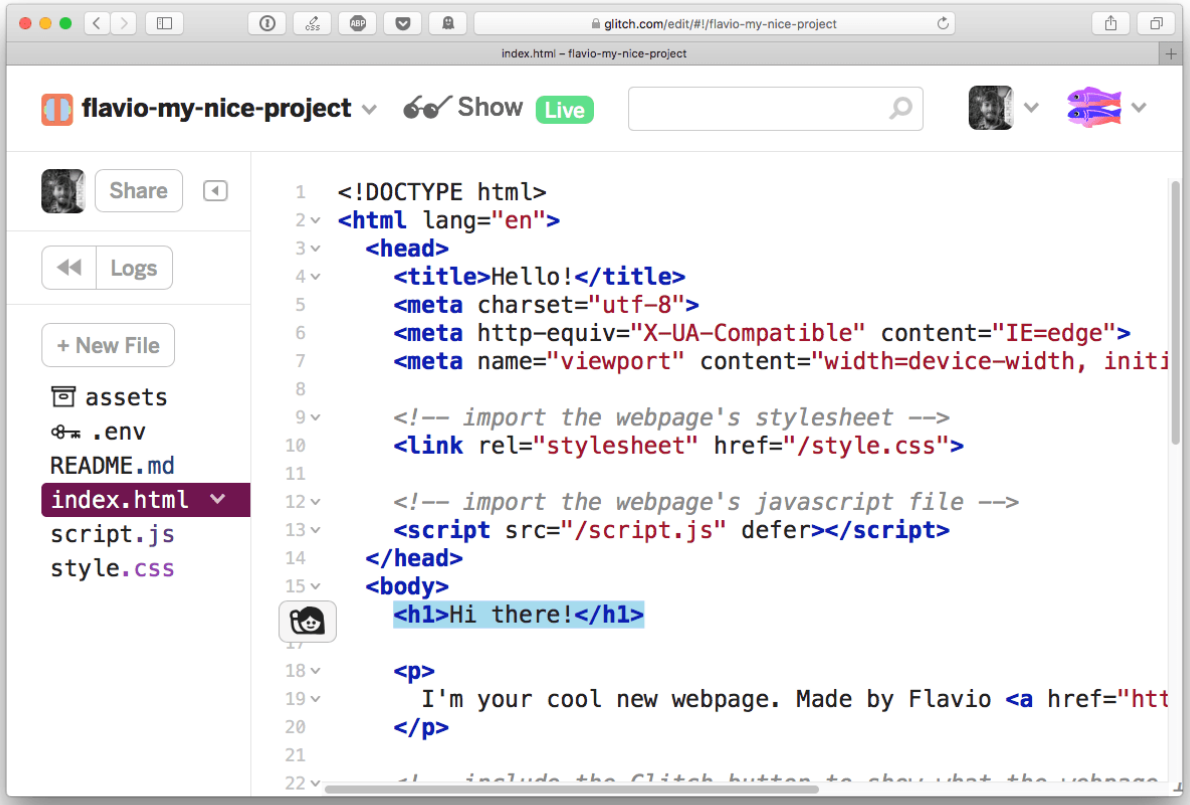
Collaborating on a glitch

From the Share panel, the **Invite Collaborators to edit** link lets you invite anyone to edit the glitch in real time with you.

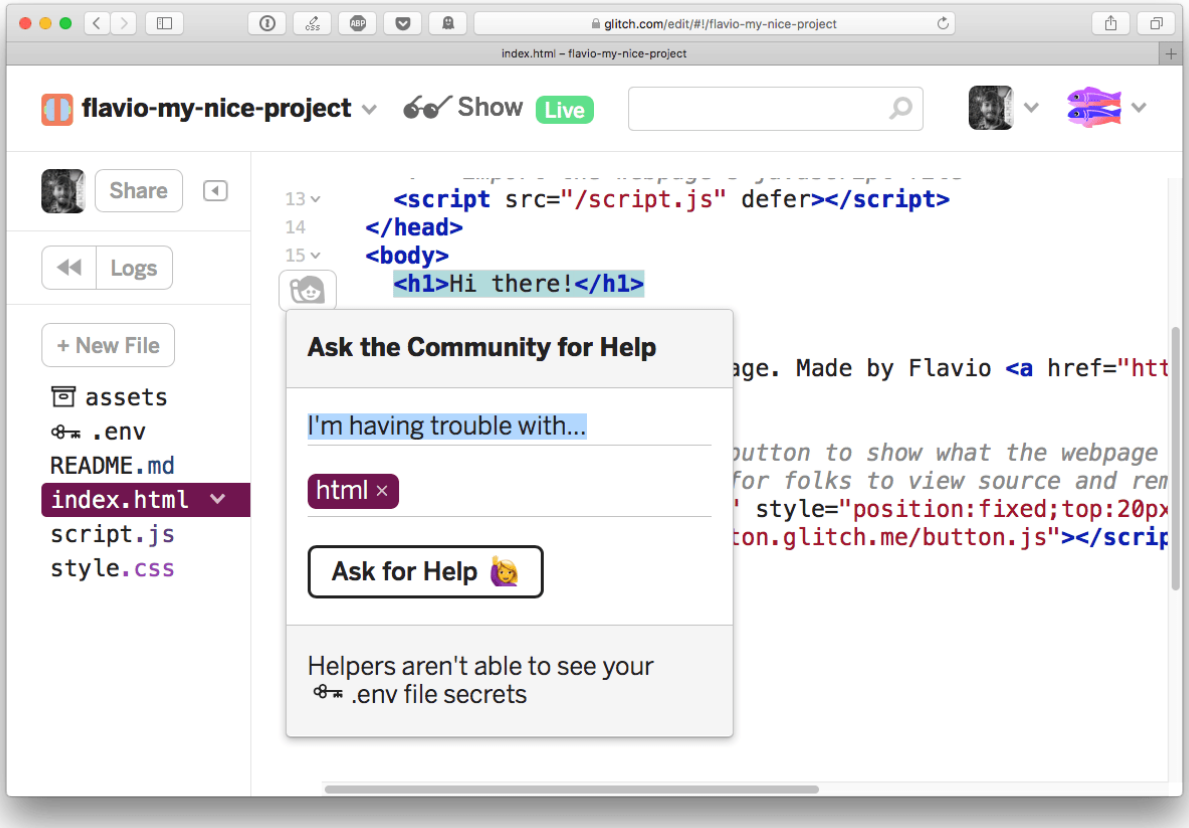
You can see their changes as they make it. It's pretty cool!

Asking for help

Linked to this collaboration feature, there's a great one: you can ask help from anyone in the world, just by selecting some text in the page, and click the raised hand icon:



This opens a panel where you can add a language tag, and a brief description of what you need:

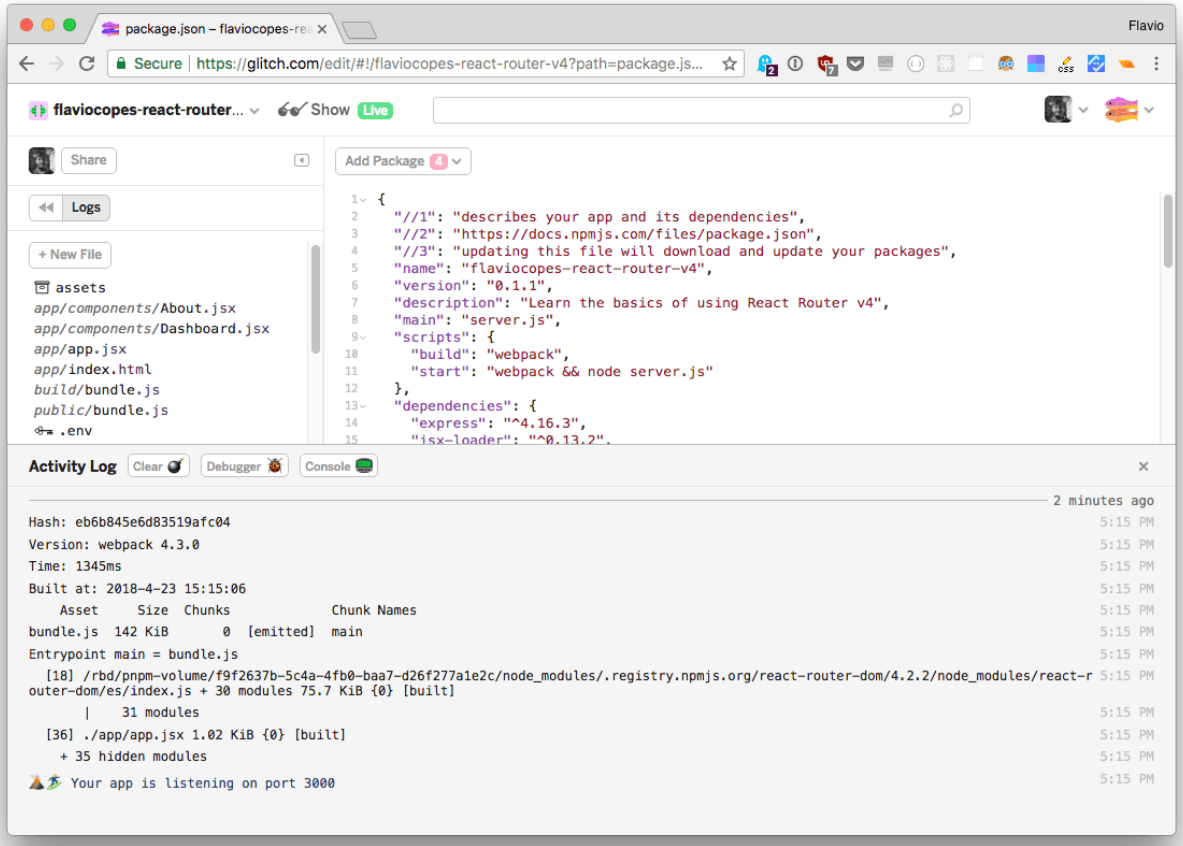


Once done, your request will be shown in the Glitch homepage for anyone to pick up.

When a person jumps in to help, they see the line you highlighted, and I found that comments made a good way to communicate like a chat:

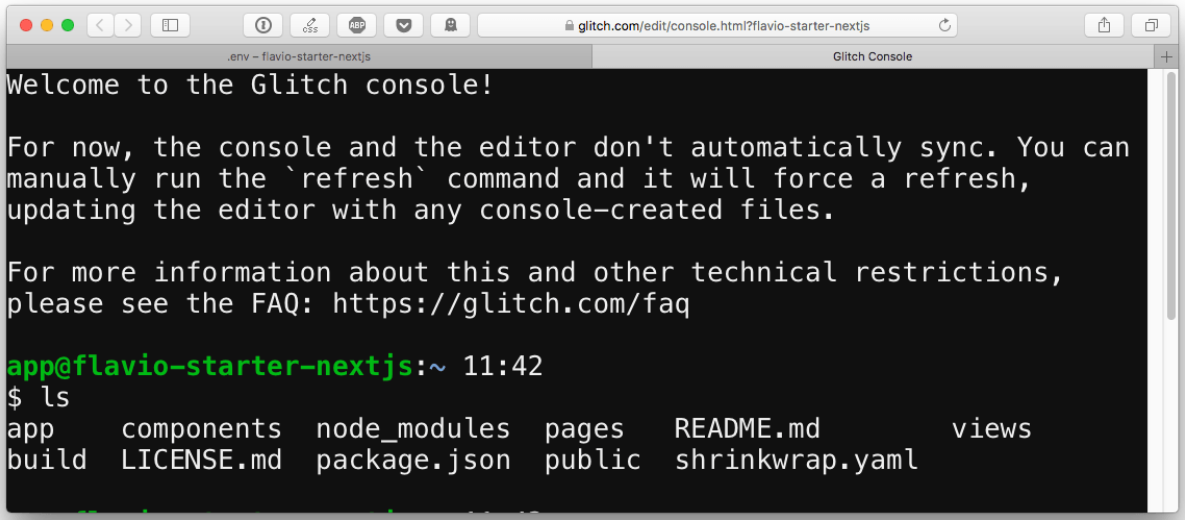
See the logs

Click **Logs** to have access to all the logs of the app:



Access the console

From the Logs panel, there is a **Console** button. Click it to open the interactive console in a separate tab in the browser:



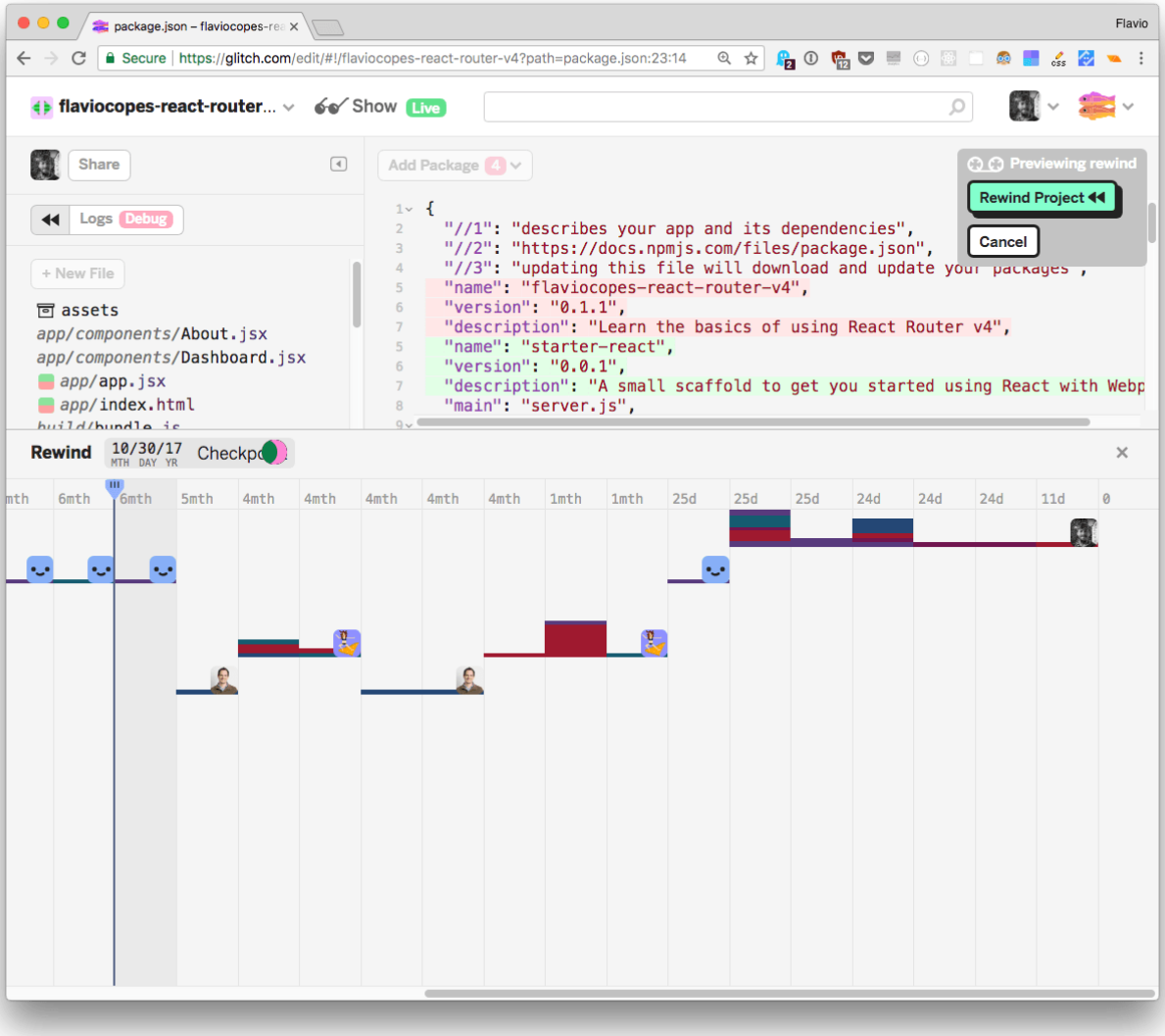
The debugger

Clicking the **Debugger** button in the Logs panel, an instance of the Chrome DevTools opens in another tab with a link to the debugger URL.

The changes history

A great feature is the ability to check all your changes in the project history.

It's a lot like how Git works - in fact, under the hoods it's Git powering this really easy to use interface, which opens clicking the ⏪ button:



How is Glitch different than Codepen or JSFiddle?

One big difference that separates Glitch from other tools is the ability to run server-side code.

Codepen and JSFiddle can only run frontend code, while a Glitch can even be used as a lightweight server for your apps - keeping the usage limits in mind.

For example I have set up an Express.js server that is triggered by a Webhook at specific times during the day to perform some duties. I don't need to worry about it running on another server, I just wrote it on Glitch and run directly from there.

That's it!

I hope you like my small tutorial on using Glitch, and I hope I explained most of the killer features of it.

More questions?

I suggest to just try it, and see if it clicks for you too.

The Glitch FAQ (https://glitch.com/faq) is a great place to start.

Have fun!

AIRTABLE API FOR DEVELOPERS

Airtable is an amazing tool. Discover why it's great for any developer to know about it and its API

Airtable is an amazing tool.

It’s a mix between a spreadsheet and a database.

As a developer you get to create a database with a very nice to use interface, with the ease of use and editing of a spreadsheet, and you can easily update your records even from a mobile app.

Perfect for prototypes

Airtable is much more than a glorified spreadsheet, however. It is a perfect tool for a developer looking to prototype or create an MVP of an application.

An MVP, or Minimum Viable Product, is one initial version of an application or product.

Most products fail not because of technical limitations or because “the stack did not scale”. They fail because either there is no need for them, or the maker does not have a clear way to market the product.

Creating an MVP minimizes the risks of spending months trying to build the perfect app, and then realizing no one wants it.

A great API

Airtable has an absolutely nice API to work with, which makes it easy to interface with your Airtable database programmatically.

This is what makes it 10x superior to a standard spreadsheet, when it comes to data handling AND making it easy to authenticate.

The API has a limit of 5 requests per second, which is not high, but still reasonable to work with for most scenarios.

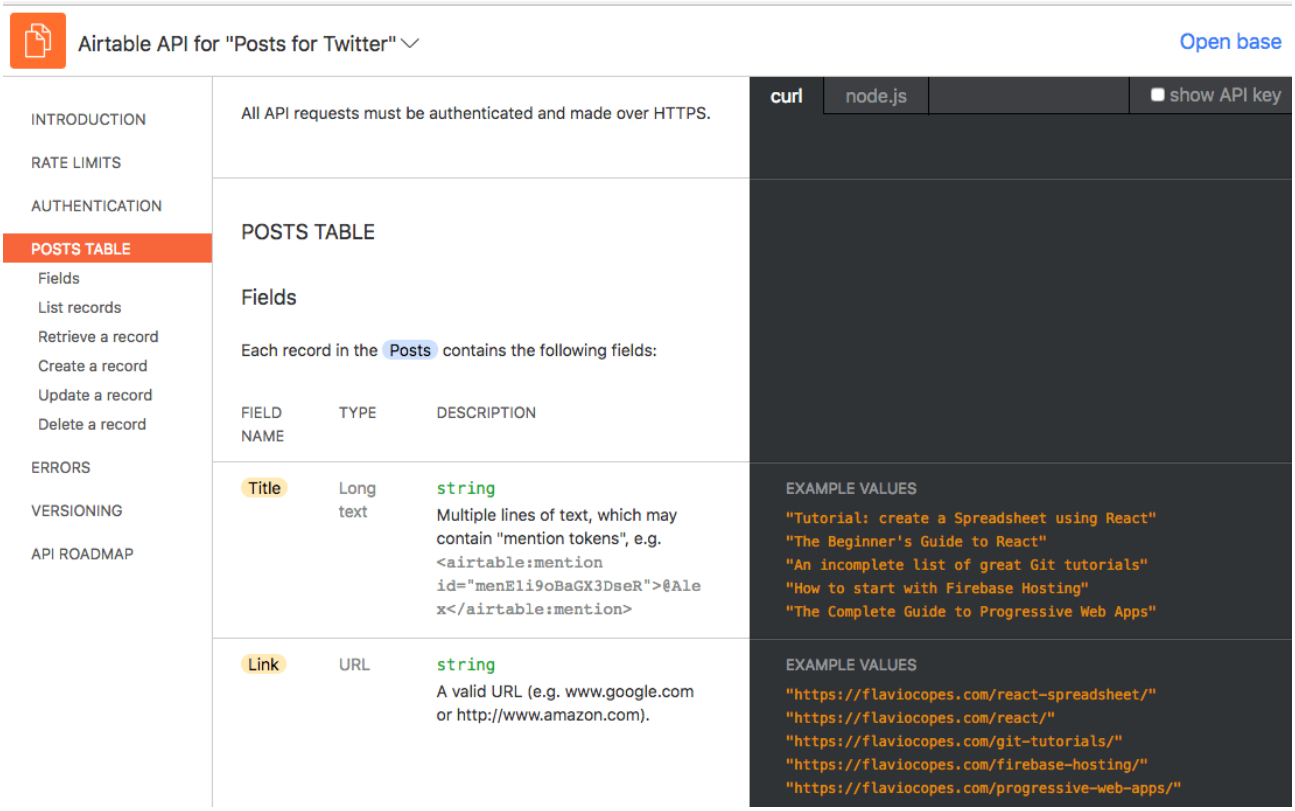
A great documentation for the API

As developers we spend a lot of time reading through docs and trying to figure out how things work.

An API is tricky because you need to interact with a service, and you want to both learn what the service exposes, and how can you use the API to do what you need.

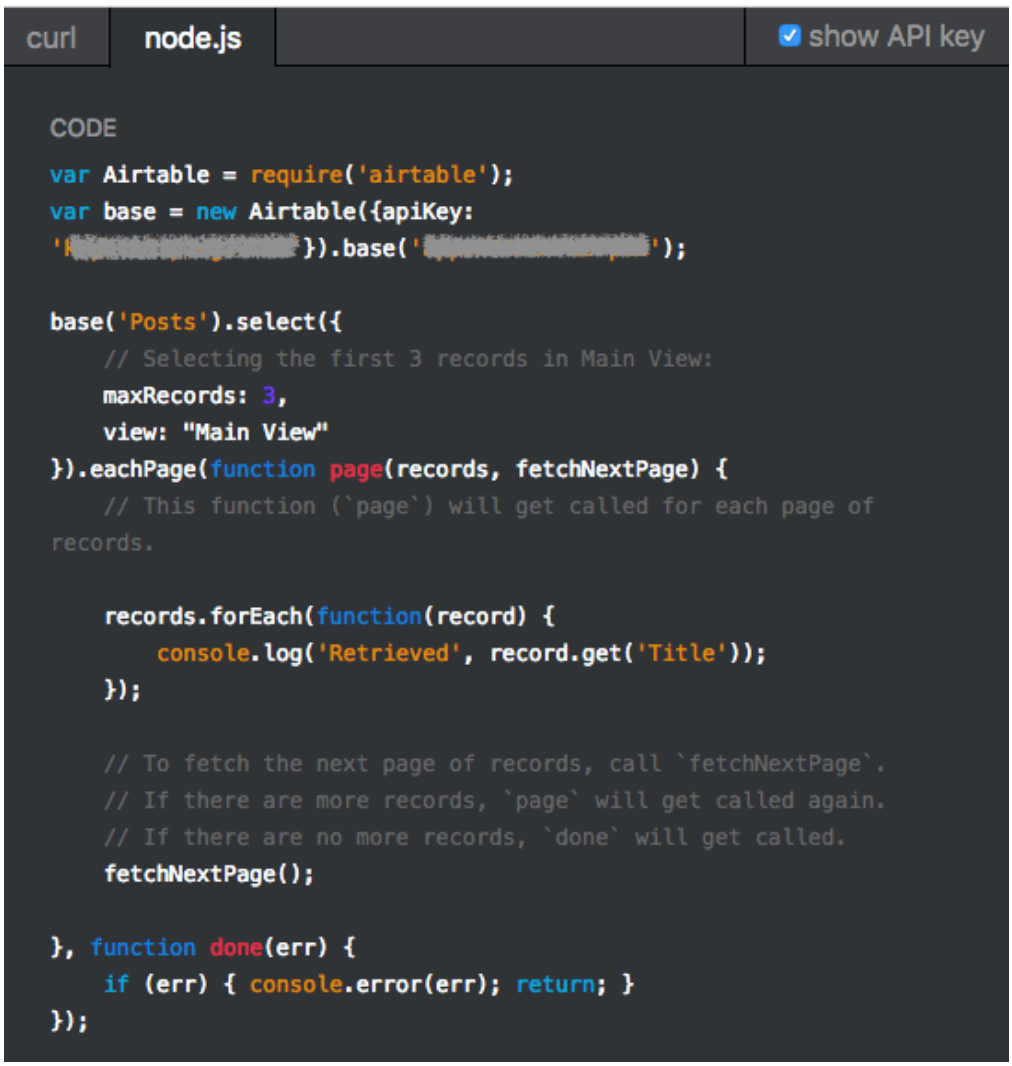
Airtable raises the bar for any API documentation out there. It puts your API keys, base IDs and table names directly in the examples, so you just need to copy and paste them into your codebase and you're ready to go.

Not just that, the examples in the API docs use the actual data in your table. In this image, notice how the fields example values are actual values I put in my table:



The API documentation offers examples using `curl`:

and their Node.js official client:



The official Node.js client

Airtable maintains the official Airtable.js (https://github.com/airtable/airtable.js) Node.js client library, a very easy to use way to access the Airtable data.

It's convenient because it offers a built-in logic to handle rate limits and retrying the requests when you exceed them.

Let's see a few common operations you can perform with the API, but first let's define a couple values we'll reference in the code:

- `API_KEY` : the Airtable API key
- `BASE_NAME` : the name of the base you'll work with
- `TABLE_NAME` : the name of the table in that base.
- `VIEW_NAME` : the name of the table view.

A base is a short term for *database*, and it can contain many tables.

A table has one or more views that organize the same data in a different way. There's always at least one view (see more on views (https://support.airtable.com/hc/en-us/articles/202624989-Guide-to-views))

Authenticate

You can set up the `AIRTABLE_API_KEY` environment variable, and Airbase.js will automatically use that, or explicitly add it into your code:

```
const Airtable = require('airtable')

Airtable.configure({
```

```
    apiKey: API_KEY
})
```

Initialize a base

```
const base = require('airtable').base(BASE_NAME)
```

or, if you already initialized the Airtable variable, use

```
const base = Airtable.base(BASE_NAME)
```

Reference a table

With a `base` object, you can now reference a table using

```
const table = base(TABLE_NAME)
```

Retrieve the table records

Any row inside a table is called a **record**.

Airtable returns a maximum of 100 records in each page of results. If you know you will never go over 100 items in a table, just use the `firstPage` method:

```
table.select({
    view: VIEW_NAME
```

```
}).firstPage((err, records) => {
    if (err) {
      console.error(err)
      return
    }

    //all records are in the `records` array, do something with it
})
```

If you have (or expect) more than 100 records, you need to paginate through them, using the `eachPage` method:

```
let records = []

// called for every page of records
const processPage = (partialRecords, fetchNextPage) => {
  records = [...records, ...partialRecords]
  fetchNextPage()
}

// called when all the records have been retrieved
const processRecords = (err) => {
  if (err) {
    console.error(err)
    return
  }

  //process the `records` array and do something with it
}

table.select({
    view: VIEW_NAME
}).eachPage(processPage, processRecords)
```

Inspecting the record content

Any record has a number of properties which you can inspect.

First, you can get its ID:

```
record.id

//or

record.getId()
```

and the time of creation:

```
record.createdTime
```

and you can get any of its properties, which you access through the column name:

```
record.get('Title')
record.get('Description')
record.get('Date')
```

Get a specific record

You can get a specific record by ID:

```
const record_id = //...

table.find(record_id, (err, record) => {
    if (err) {
      console.error(err)
      return
    }
```

```
    console.log(record)
})
```

Create a new record

You can add a new record

```
table.create({
  "Title": "Tutorial: create a Spreadsheet using React",
  "Link": "https://flaviocopes.com/react-spreadsheet/",
}, (err, record) => {
    if (err) {
      console.error(err)
      return
    }

    console.log(record.getId())
})
```

Update a record

You can update a single field of a record, and leave the other fields untouched, using `update`:

```
const record_id = //...

table.update(record_id, {
  "Title": "The modified title"
}, (err, record) => {
    if (err) {
      console.error(err)
      return
    }
```

```
    console.log(record.get('Title'))
})
```

Or, you can update some fields in a record and **clear out** the ones you did not touch, with `replace`:

```
const record_id = //...

table.replace(record_id, {
  "Title": "The modified title",
  "Description": "Another description"
}, (err, record) => {
    if (err) {
      console.error(err)
      return
    }

    console.log(record)
})
```

Delete a record

A record can be deleted using

```
const record_id = //...

table.destroy(record_id, (err, deletedRecord) => {
    if (err) {
      console.error(err)
      return
    }

    console.log('Deleted record', deletedRecord.id)
})
```

A TUTORIAL TO HOST YOUR STATIC SITE ON NETLIFY

Discover Netlify, a great hosting service ideal for static sites which has a nice free plan, free CDN and it's blazing fast

- Introducing Netlify
- Netlify and Hugo
- Advanced functionality offered by Netlify for Static Sites
- Previewing branches

I recently switched my blog hosting to Netlify (https://www.netlify.com) .

I did so while my previous hosting was having some issues that made my site unreachable for a few hours, and while I waited for it to get up online again, I created a replica of my site on Netlify.

Since this blog runs on Hugo (https://gohugo.io) , which is a Static Site Generator, I need very little time to move the blog files around. All I need is something that can serve HTML files, which is pretty much any hosting on the planet.

I started looking for the best platform for a static site, and a few stood out but I eventually tried Netlify, and I'm glad I did.

Introducing Netlify

There are a few things that made a great impression to me before trying it.

First, the **free plan is very generous** for free or commercial projects, with 100GB of free monthly bandwidth, and for a static site with just a few images here and there, it's a lot of space!

They include a global CDN, to make sure speed is not a concern even in continents far away from the central location servers.

You can point your DNS nameservers to Netlify and they will handle everything for you with a very nice interface to set up advanced needs.

They of course support having a custom domain and HTTPS.

Coming from Firebase, I expected a very programmer friendly way to manage deploys, but I found it even better with regards to handling each Static Site Generator.

Netlify and Hugo

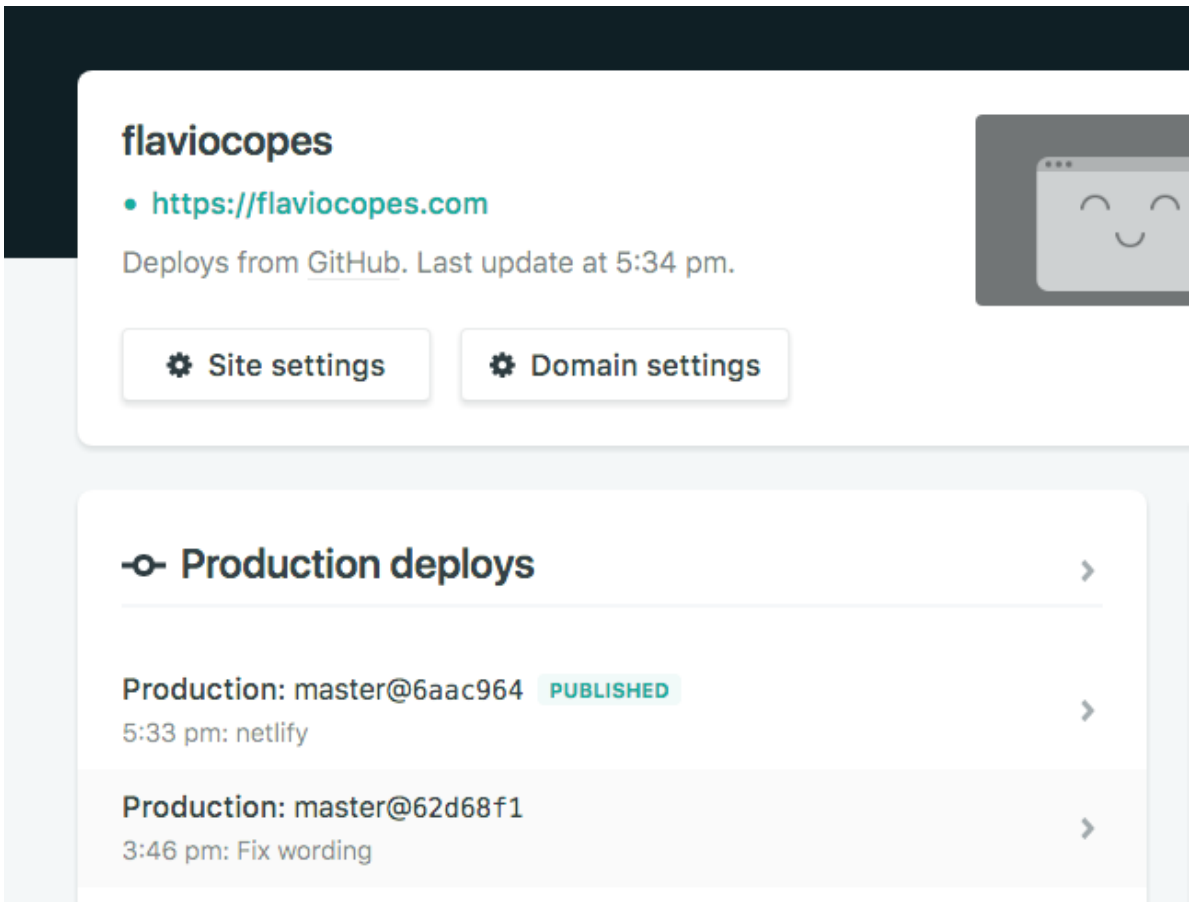
I use Hugo, and locally I run a server by using its built-in tool `hugo server`, which handles rebuilding all the HTML every time I make a

change, and it runs an HTTP server on port `1313` by default.

To generate the static site, I have to run `hugo`, and this creates a series of files in the `public/` folder.

I followed this method on Firebase: I ran `hugo` to create the files, then `firebase deploy`, configured to push my `public/` folder content to the Google servers.

In the case of Netlify however, I linked it to my private GitHub repository that hosts the site, and every time I push to the master branch, the one I told Netlify to sync with, Netlify initiates a new deploy, and the changes are live within seconds.



> *TIP: if you use Hugo on Netlify, make sure you set HUGO_VERSION in* `netlify.toml` *to the latest Hugo stable release, as the default version might be old and (at the time of writing) does not support recent features like post bundles*

If you think this is nothing new, you're right, since this is not hard to implement on your own server (I do so on other sites not hosted on Netlify), but here's something new: you can preview any GitHub (or GitLab, or BitBucket) branch / PR on a separate URL, all while your main site is live and running with the "stable" content.

Another cool feature is the ability to perform A/B testing on 2 different Git branches.

Advanced functionality offered by Netlify for Static Sites

Static sites have the obvious limitation of not being able to do any server-side operation, like the ones you'd expect from a traditional CMS for example.

This is an advantage (less security issues to care about) but also a limitation in the functionality you can implement.

A blog is nothing complex, maybe you want to add comments and they can be done using services like Disqus or others.

Or maybe you want to add a form and you do so by embedding forms generated on 3rd part applications, like Wufoo or Google Forms.

Netlify provides a suite of tools to handle Forms (https://www.netlify.com/docs/form-handling/#spam-filtering) , authenticate users and even deploy and manage Lambda functions (https://macarthur.me/posts/building-a-lambda-function-with-netlify/) .

Need to password protect a site before launching it? ✅

Need to handle CORS? ✅

Need to have 301 redirects? ✅

Need pre-rendering for your SPA? ✅

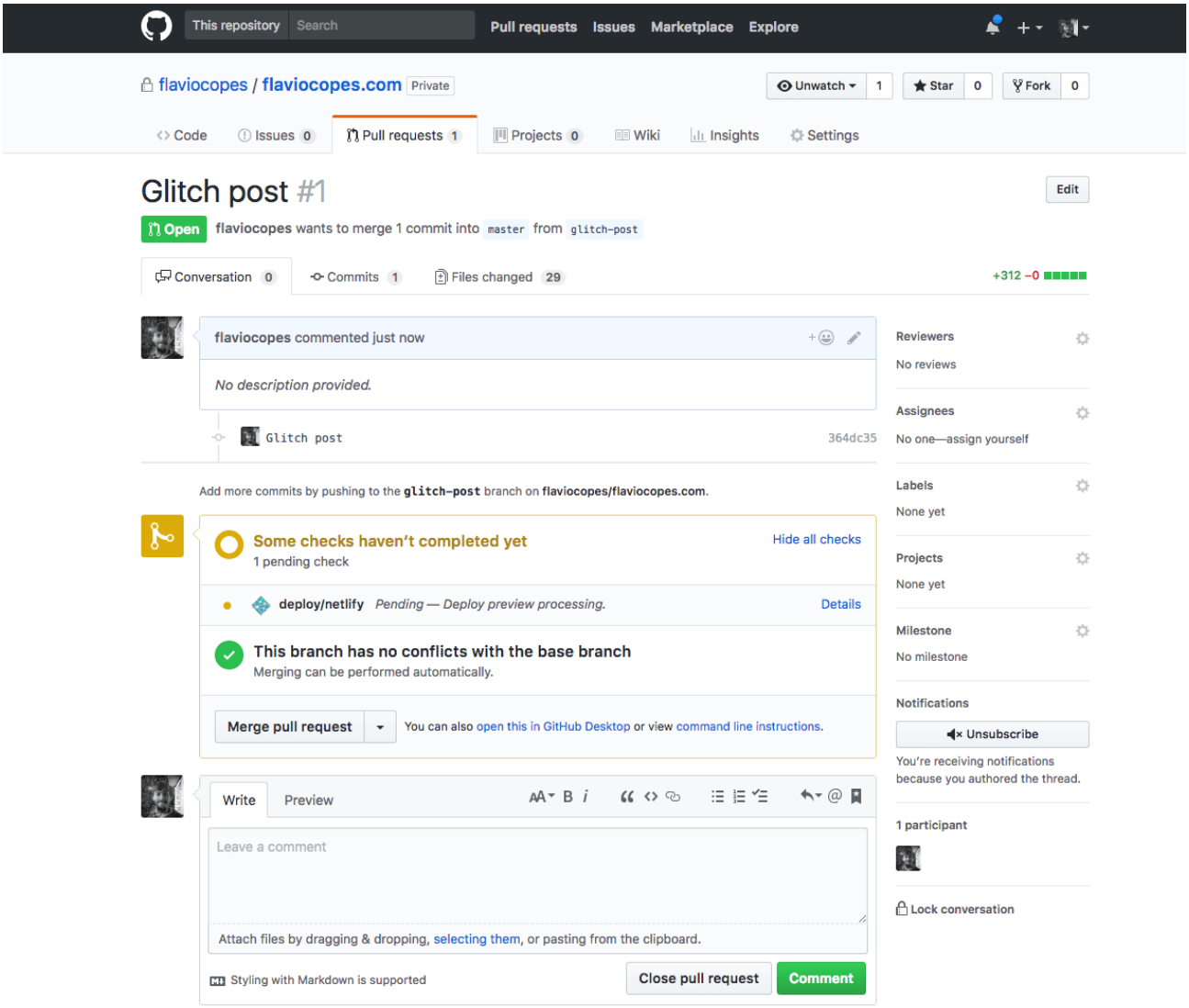
I just scratched the surface of the things you can do with Netlify without reaching out to 3rd part services, and I hope I gave you a reason to try it out.

Previewing branches

The GitHub integration works great with Pull Requests.

Every time you push a Pull Request, Netlify deploys that branch on a specific URL which you can share with your team, or to anyone that you want.

Here I made a Pull Request to preview a blog post, without making it available on my public blog:



Netlify immediately picked it up, and automatically deployed it 🎉

Clicking the link points you to the special URL that lets you preview the PR version of the site.