

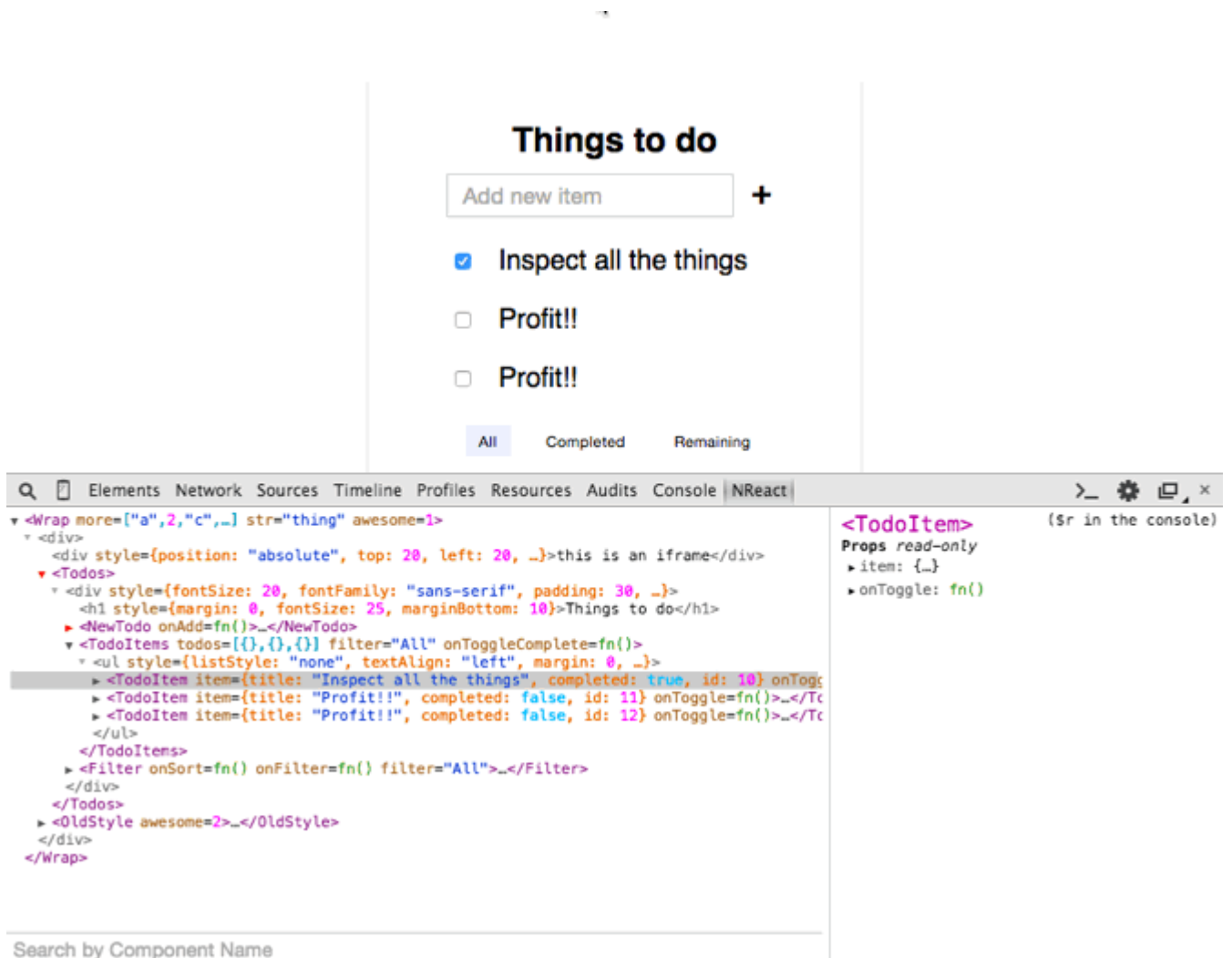
# Module 4

## Debugging React

### Using React Developer Tools

React Developer Tools lets you inspect the React component hierarchy, including component props and state.

It exists both as a browser extension (for [Chrome](#) and [Firefox](#)), and as a [standalone app](#) (works with other environments including Safari, IE, and React Native).



# Installation

---

## Pre-packaged

The official extensions represent the current stable release.

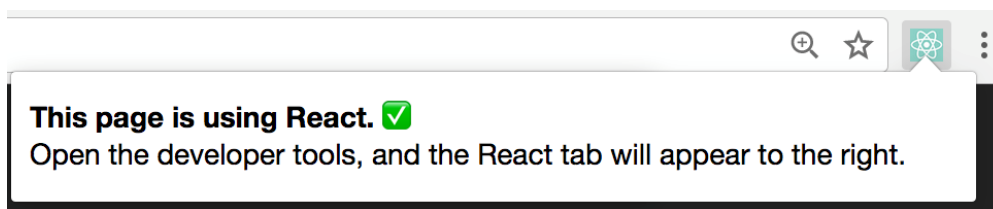
- [Chrome extension](#)
- [Firefox extension](#)
- [Standalone app \(Safari, React Native, etc\)](#)

Opera users can [enable Chrome extensions](#) and then install the [Chrome extension](#).

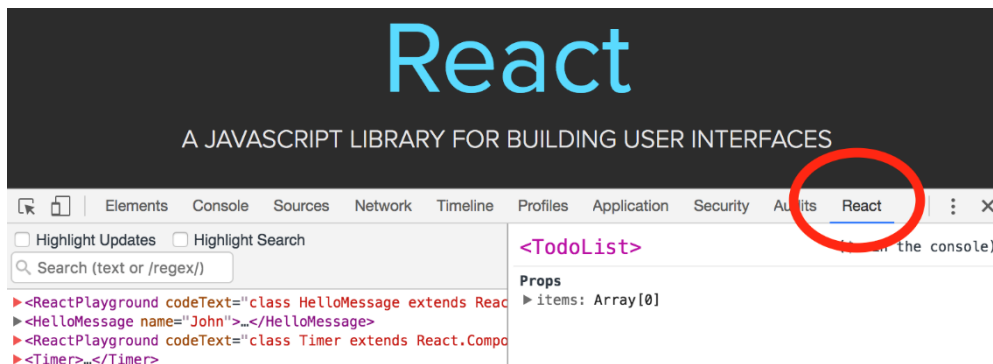
## Usage

---

The extension icon will light up on the websites using React:



On such websites, you will see a tab called React in Chrome Developer Tools:



A quick way to bring up the DevTools is to right-click on the page and press Inspect.

## Tree View

- Arrow keys or hjkl for navigation
- Right click a component to show in elements pane, scroll into view, show source, etc.
- Differently-colored collapser means the component has state/context

```
▼ <Wrap>
  ▼ <div>
    <div style={position: "absolute", top: 20, left: 20, ...}>this is an iframe</div>
    ▼ <Todos>
      ▼ <div style={fontSize: 20, fontFamily: "sans-serif", padding: 30, ...}>
        <h1 style={margin: 0, fontSize: 25, marginBottom: 10}>Things to do</h1>
        ▶ <NewTodo onAdd=bound onAdd()>...</NewTodo>
        ▼ <TodoItems todos=[{...}, {...}, {...}] filter="All" onToggleComplete=bound toggleComplete()> == $r
          ▼ <ul style={listStyle: "none", textAlign: "left", margin: 0, ...}>
            ▶ <TodoItem key="10" item={title: "Inspect all the things", completed: true, id: 10} onToggl
            ▶ <TodoItem key="11" item={title: "Profit!!", completed: false, id: 11} onToggle=onToggle()>
            ▶ <TodoItem key="12" item={title: "Profit!!", completed: false, id: 12} onToggle=onToggle()>
```

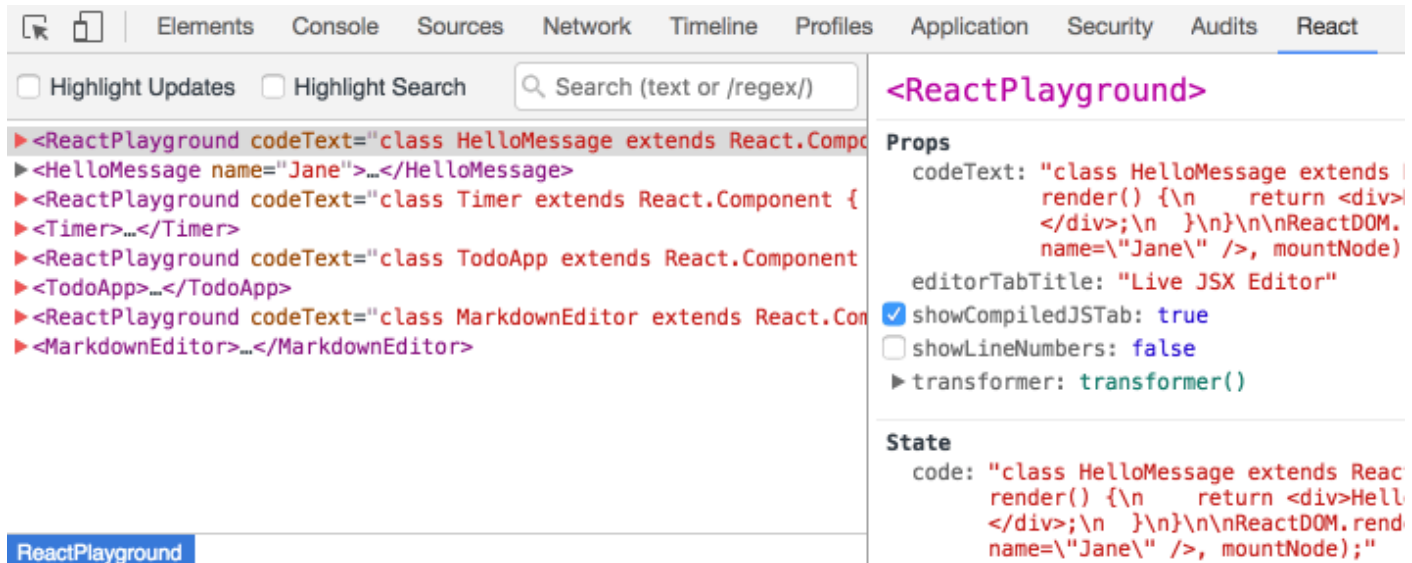
## Side Pane

- Right-click to store as global variable
- Updates are highlighted

```
<NewTodo> ($r in the console)
Props
  ▶ onAdd: fn()
State
  text: "Text"
```

## Search Bar

- Use the search bar to find components by name



## Handy Tips

### Finding Component by a DOM Node

If you inspect a React element on the page using the regular **Elements** tab, then switch over to the **React** tab, that element will be automatically selected in the React tree.

### Finding DOM Node by a Component

You can right-click any React element in the **React** tab, and choose "Find the DOM node". This will bring you to the corresponding DOM node in the **Elements** tab.

### Displaying Element Source

You may include the [transform-react-jsx-source](#) Babel plugin to see the source file and line number of React elements. This information appears in the bottom of the right panel when available. Don't forget to disable it in production! (Tip: if you use [Create React App](#) it is already enabled in development.)

### Usage with React Native and Safari

There is a [standalone version](#) that works with other environments such as React Native and Safari.

# Using Visual Studio Code

## SHOULD I SWITCH TO VS CODE? AND WHY?

If you're looking for suggestions for whether to use it or not, let me say **yes**, you should switch to it from whatever other editor you are using now.

This editor builds on top of decades of editor experience from Microsoft.

The code of the editor is completely Open Source, and there's no payment required to use it.

It uses [Electron](#) as its base, which enables it to be cross platform and work on Mac, Windows and Linux. It's built using Node.js, and you can extend it using JavaScript (which makes it a win for all us JavaScript developers).

It's **fast**, easily the fastest editor I've used after Sublime Text.

It has won the enthusiasm of the community: there are thousands of **extensions**, some official, and some made by the community, and it's [winning surveys](#).

Microsoft releases an update every month. Frequent updates foster innovation and Microsoft is listening to its users, while keeping the platform as stable as possible.

## DEBUGGER

The fourth icon in the toolbar opens the JavaScript debugger. This deserves a section on its own. In the meantime check out [the official docs](#).

### How to get started in 6 steps

1. Download the [latest release of VS Code](#) and install the [Chrome debugger](#)
2. Create your React app using [create-react-app](#)
3. Use the following config for your launch.jsonfile to configure the VS Code debugger and put it inside .vscode in your root folder.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Chrome",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceRoot}/src"
    }
  ]
}
```

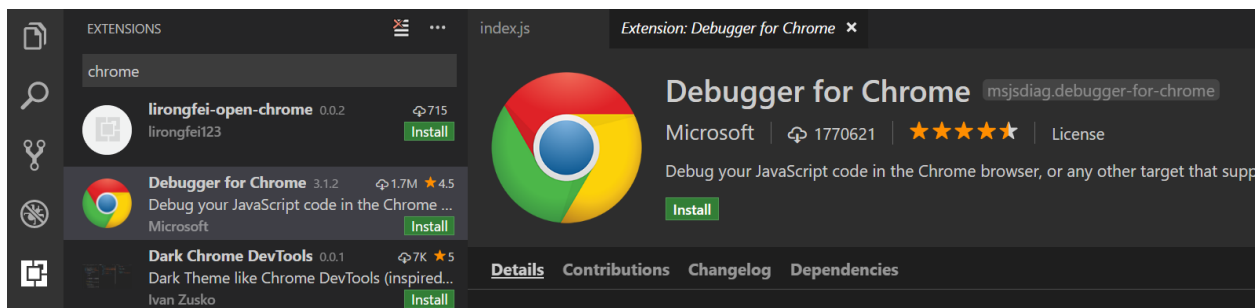
4. Start your React app by running npm start in your favorite terminal
5. Start debugging in VS Code by pressing F5 or by clicking the green debug icon

## Detailed steps - Debugging React

To debug the client side React code, we'll need to install the [Debugger for Chrome](#) extension.

Note: This section assumes you have the Chrome browser installed. Microsoft also publishes a version of this extension for their [Edge](#) browser.

Open the Extensions view (Ctrl+Shift+X) and type 'chrome' in the search box. You'll see several extensions which reference Chrome.



Press the **Install** button for **Debugger for Chrome**.

## Set a breakpoint

To set a breakpoint in `index.js`, click on the gutter to the left of the line numbers. This will set a breakpoint which will be visible as a red circle.

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4  import registerServiceWorker from './registerServiceWorker';
5  import './index.css';
6
7  var element = React.createElement('h1', {className: 'greeting'}, 'Hello, world!');
8  ReactDOM.render(element, document.getElementById('root'));
9  registerServiceWorker();
10
```

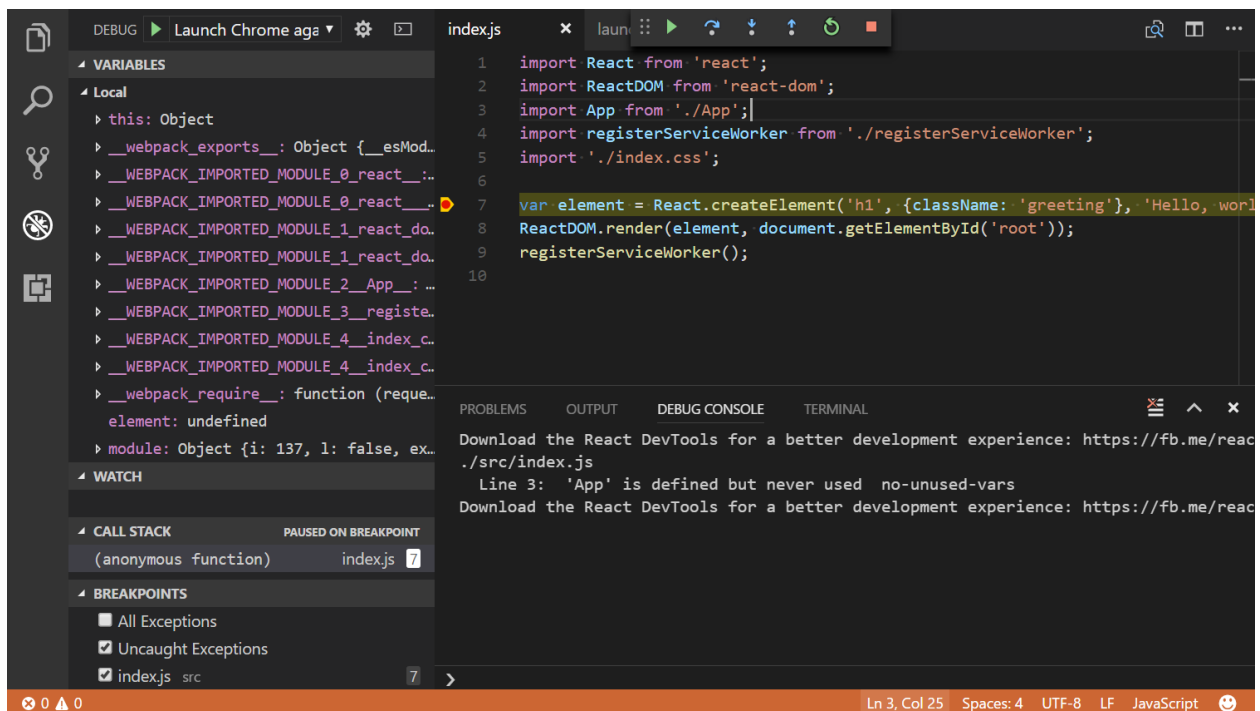
## Configure the Chrome debugger

We need to initially configure the [debugger](#). To do so, go to the Debug view (`Ctrl+Shift+D`) and click on the gear button to create a `launch.json` debugger configuration file. Choose **Chrome** from the **Select Environment** drop-down list. This will create a `launch.json` file in a new `.vscode` folder in your project which includes a configuration to launch the website.

We need to make one change for our example: change the port of the `url` from `8080` to `3000`. Your `launch.json` should look like this:

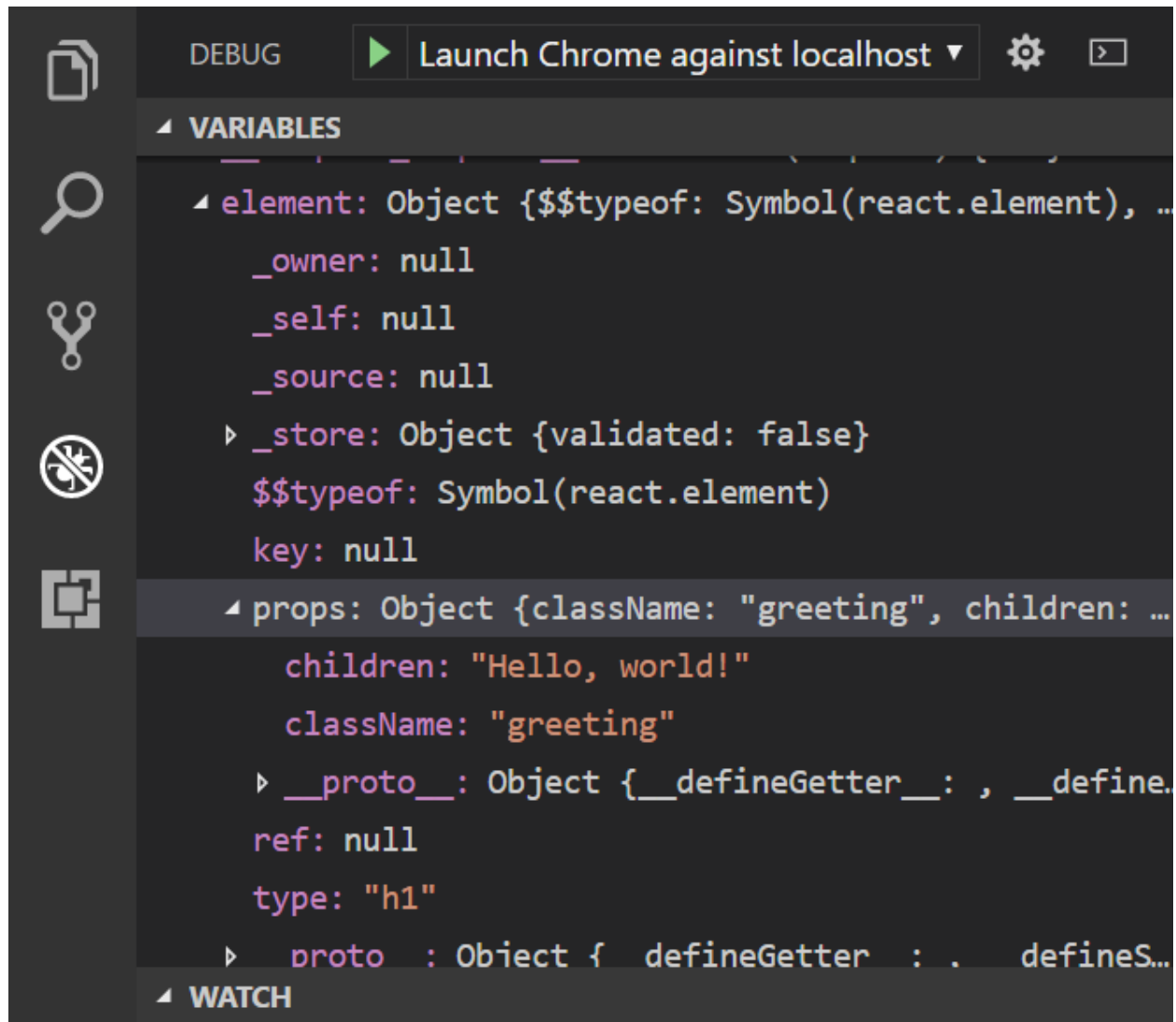
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

Ensure that your development server is running ("npm start"). Then press **F5** or the green arrow to launch the debugger and open a new browser instance. The source code where the breakpoint is set runs on startup before the debugger was attached so we won't hit the breakpoint until we refresh the web page. Refresh the page and you should hit your breakpoint.





You can step through your source code ([F10](#)), inspect variables such as `element`, and see the call stack of the client side React application.



The **Debugger for Chrome** extension README has lots of information on other configurations, working with sourcemaps, and troubleshooting. You can review it directly within VS Code from the **Extensions** view by clicking on the extension item and opening the **Details** view.



The screenshot shows the Visual Studio Code interface with the Extensions view open. The left sidebar contains icons for Explorer, Search, Source Control, Run and Debug, and Extensions. The Extensions view is active, displaying a search bar and a list of extensions. The 'Debugger for Chrome' extension by Microsoft is highlighted. The main panel shows the details for this extension, including its icon, name, version (3.1.2), download count (1.7M), publisher (Microsoft), and a five-star rating. Below this, there are tabs for 'Details', 'Contributions', 'Changelog', and 'Dependencies'. The 'Details' tab is selected, showing a section titled 'Using the debugger' with instructions on how to use the extension. The instructions mention that when a launch config is set up, the user can debug their project by picking a launch config from a dropdown on the Run and Debug view or pressing F5 to start. A 'Configuration' section follows, explaining that the extension operates in two modes: launching an instance of Chrome navigated to the user's app, or debugging a local application. It also mentions that the user can configure these modes using a `.vscode/launch.json` file, which can be created manually or by VS Code if it doesn't exist yet.

EXTENSIONS

Search Extensions in Marketplace

**Debugger for Chrome** 3.1.2 1.7M  
Debug your JavaScript code in the Ch...  
Microsoft

**Debugger for Chrome** msjsdiag.debugger-for-chrome  
Microsoft | 1770637 | ★★★★★ | License  
Debug your JavaScript code in the Chrome browser, or any other target that s

Disable Uninstall

Details Contributions Changelog Dependencies

### Using the debugger

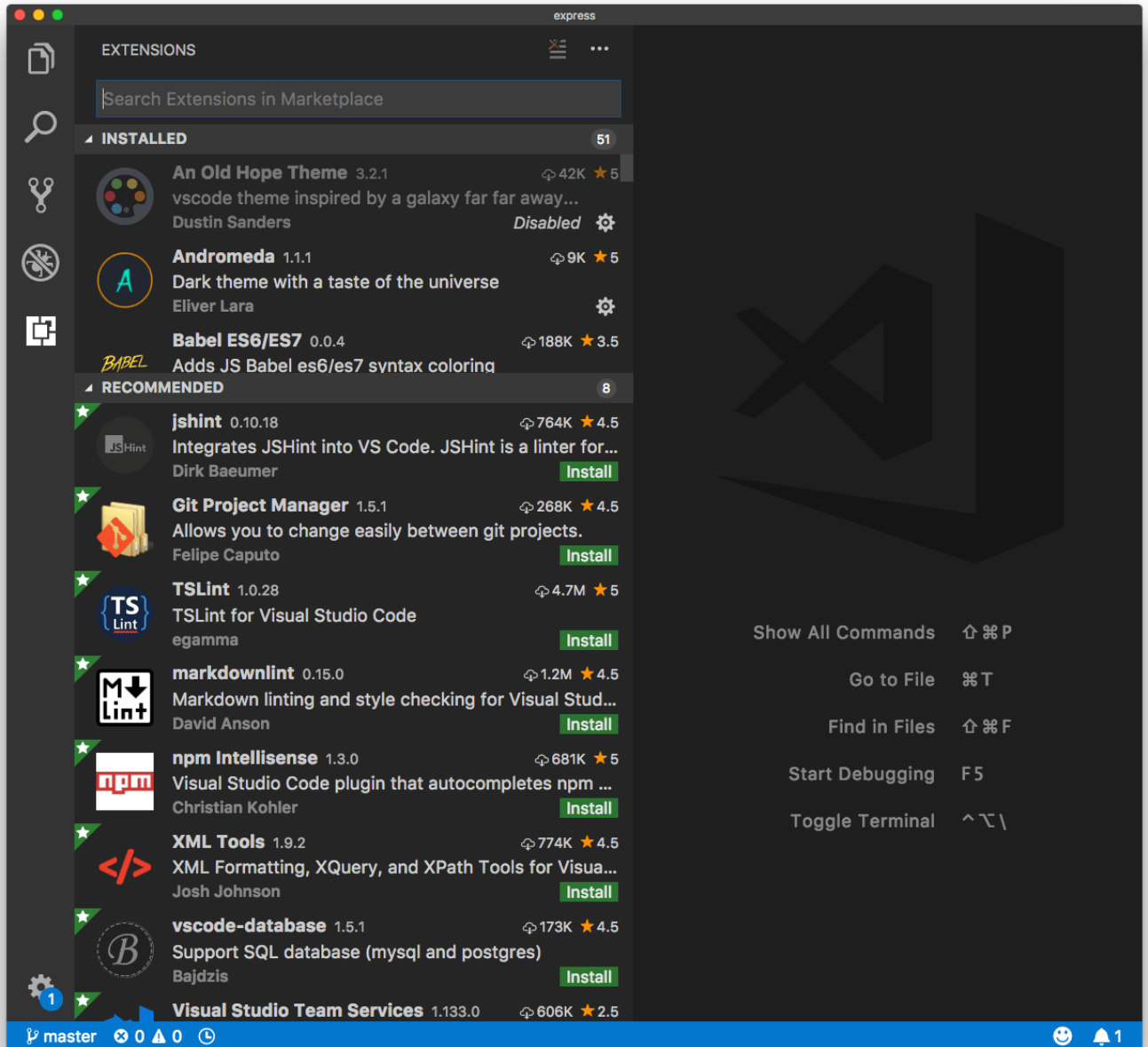
When your launch config is set up, you can debug your project! Pick a launch config from the dropdown on the Run and Debug view or F5 to start.

#### Configuration

The extension operates in two modes - it can launch an instance of Chrome navigated to your app, or it can attach to a running application. Just like when using the Node debugger, you configure these modes with a `.vscode/launch.json` file in the root of your project. You can create this file manually, or Code will create one for you if you try to run your project, and it doesn't exist yet.

# EXTENSIONS

The fifth icon brings us to extensions.



Extensions are one killer feature of VS Code.

They can provide so much value that you'll surely end up using tons of them.

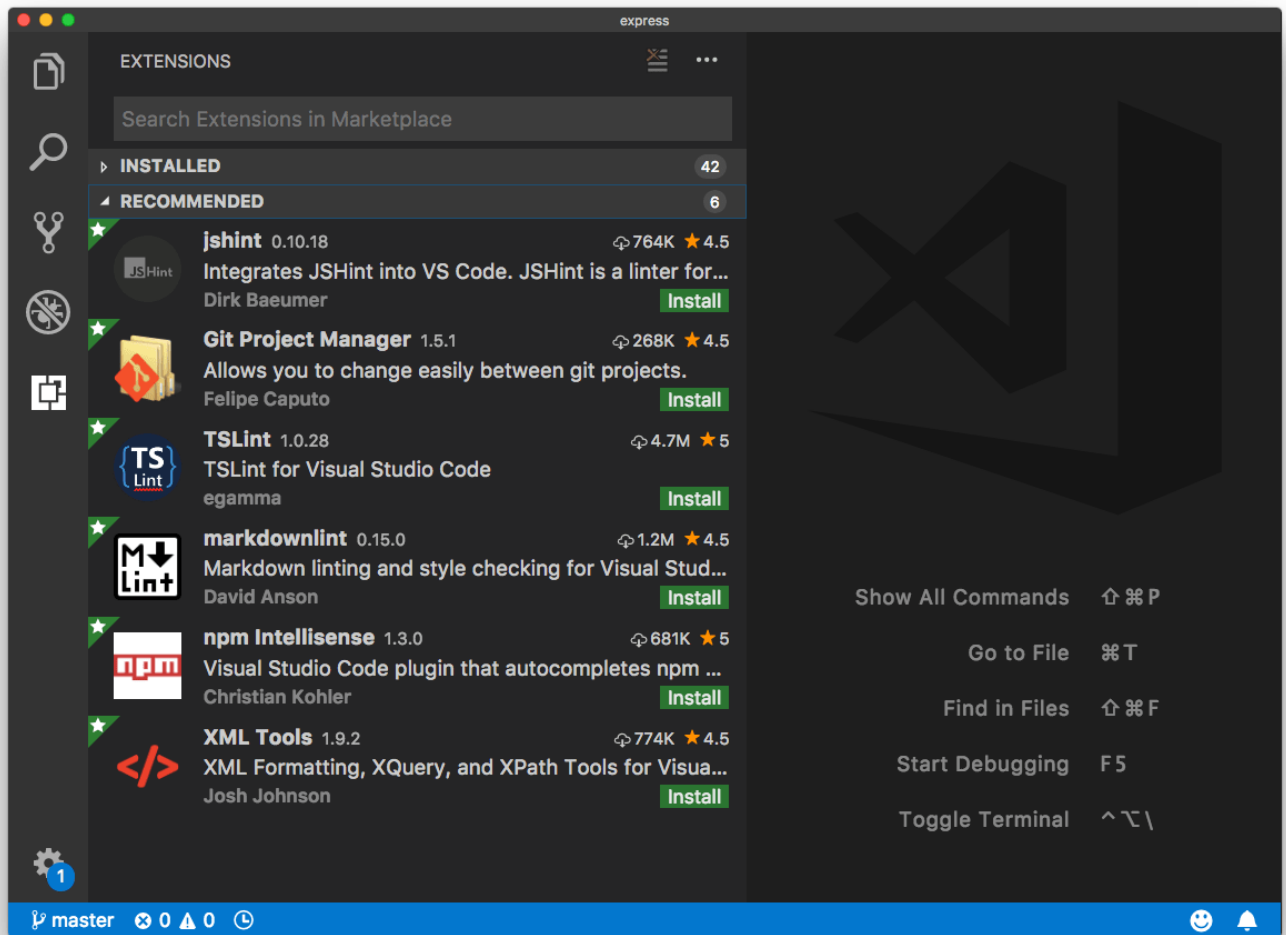
I have lots of extensions installed.

One thing to remember is that every extension you install is going to impact (more or less) the performance of your editor.

You can disable an extension you install, and enable only when you need it.

You can also disable an extension for a specific workspace (we'll talk about workspaces later). For example, you don't want to enable the JavaScript extensions in a Go project.

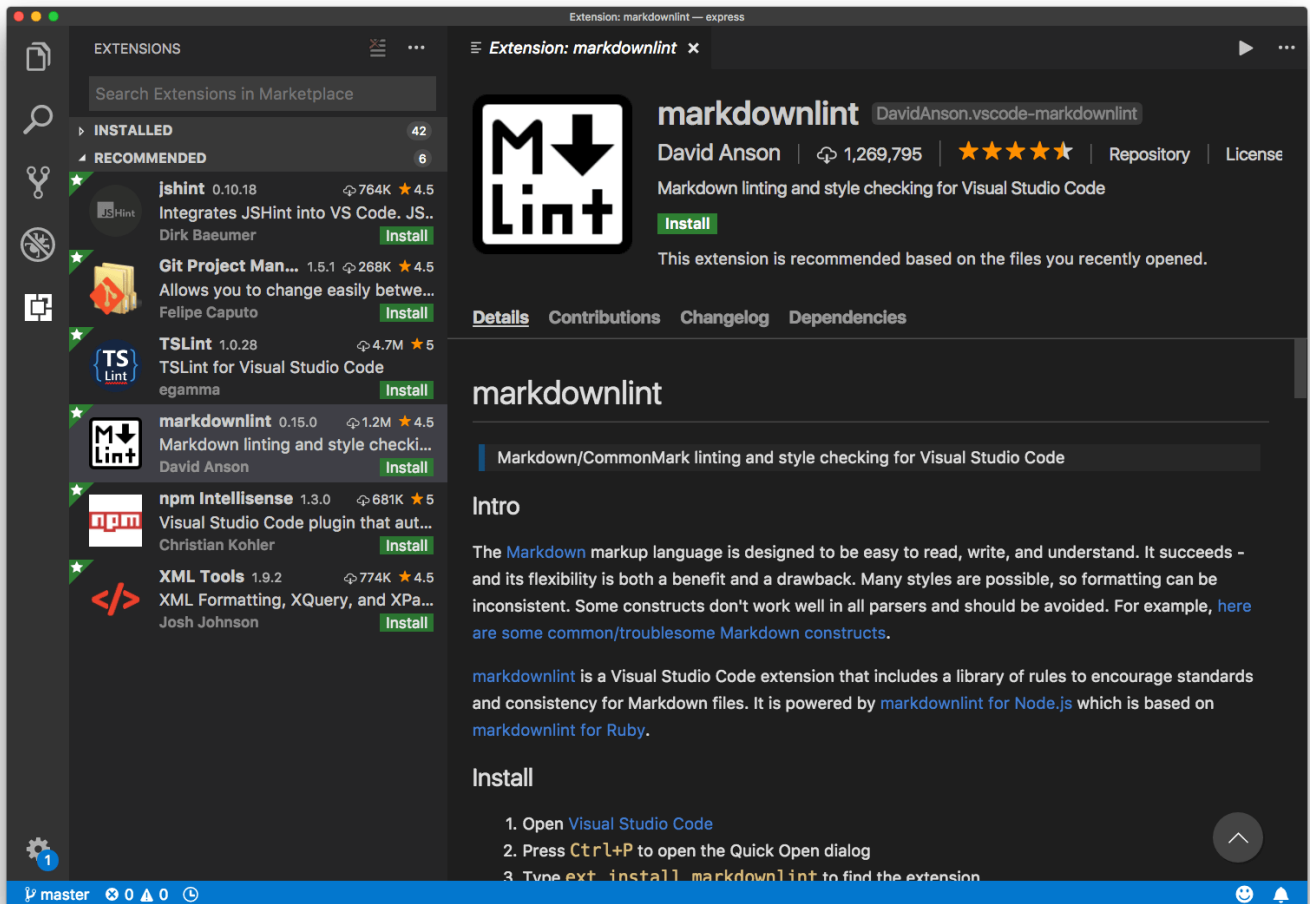
There is a list of recommended extensions, which include all the most popular tools.



Since I edit lots of markdown file, VS Code suggests me the `markdownlint` extension, which provides linting and syntax checking for Markdown files.

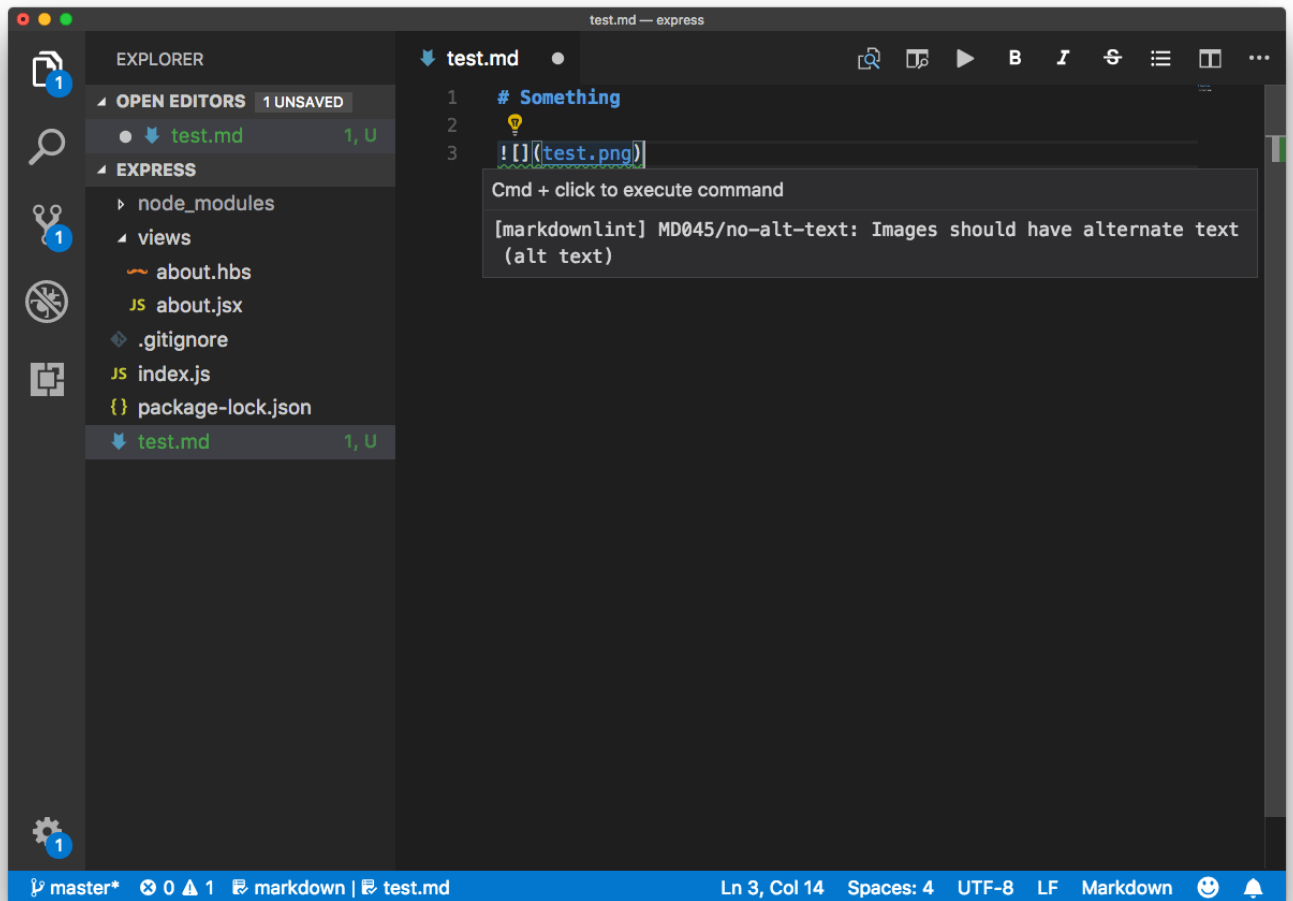
As an example, let's install it.

First, I inspect the number of views. It's 1.2M, so many! And the reviews are positive (4.5/5). Clicking the extension name opens the details on the right.



Pressing the green Install button starts the installation process, which is straightforward. It does everything for you, and you just need to click the “Reload” button to activate it, which basically reboots the editor window.

Done! Let's test it by creating a markdown file with an error, like a missing altattribute on an image. It successfully tells us so:

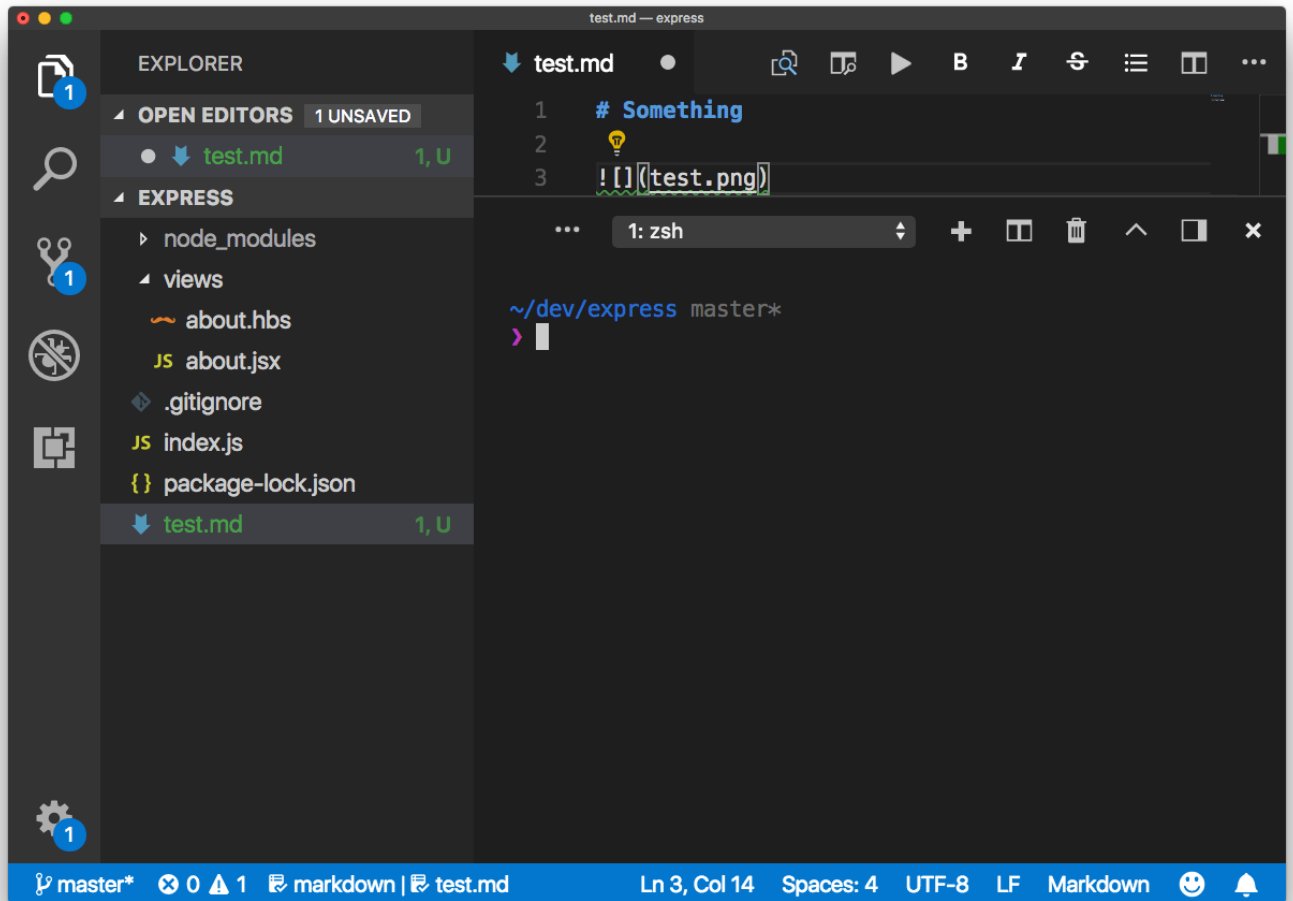


Down below I introduce some popular extensions you don't want to miss, and the ones I use the most.

## THE TERMINAL

VS Code has an integrated terminal.

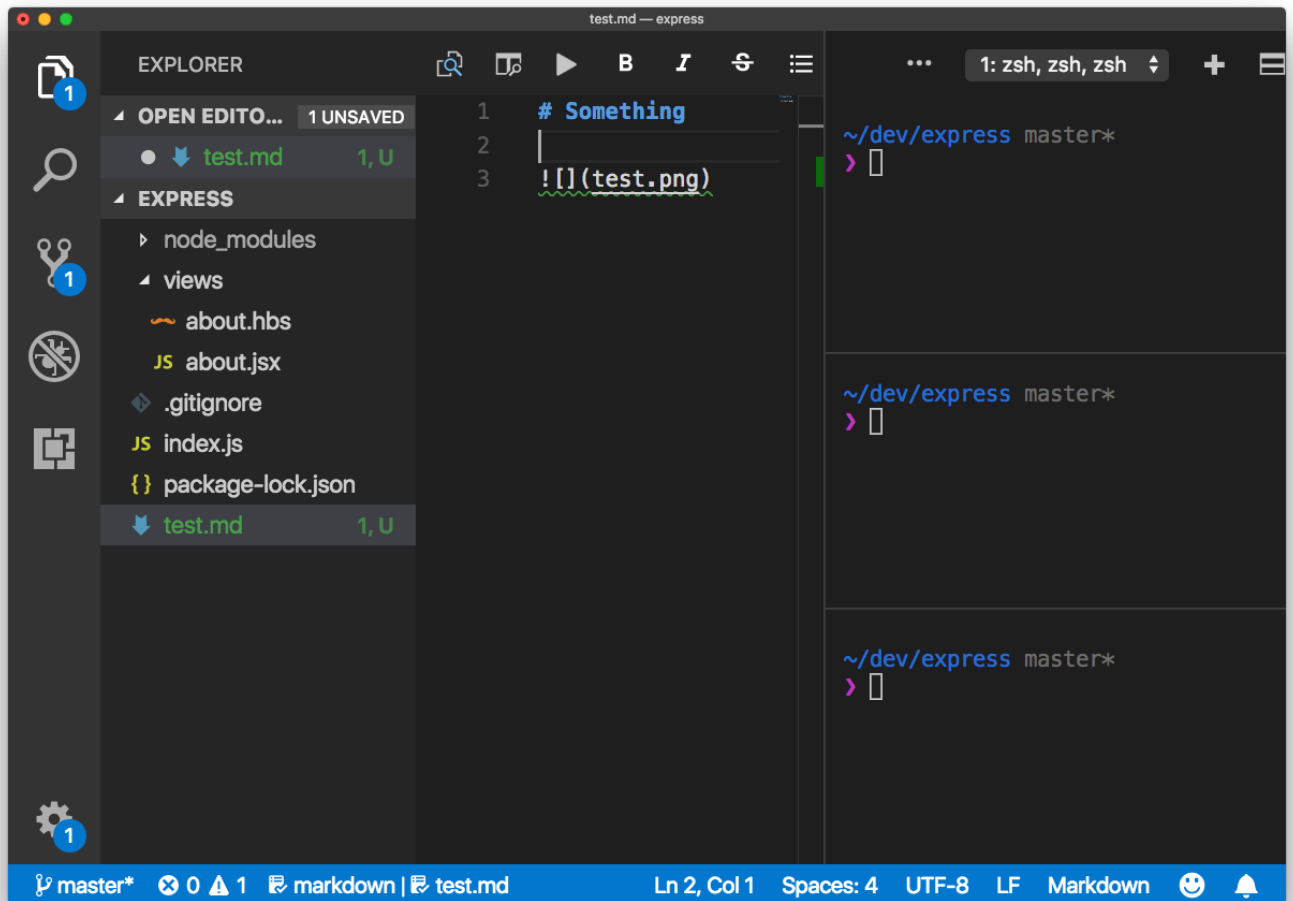
You can activate it from the menu **View** ► **Integrated Terminal**, or using **CMD+`** and it will open with your default shell.



This is very convenient because in modern web development you almost always have some **npm** or **yarn** process running in the background.



You can create more than one terminal tab, and show them one next to the other, and also stack them to the right rather than in the bottom of the window:

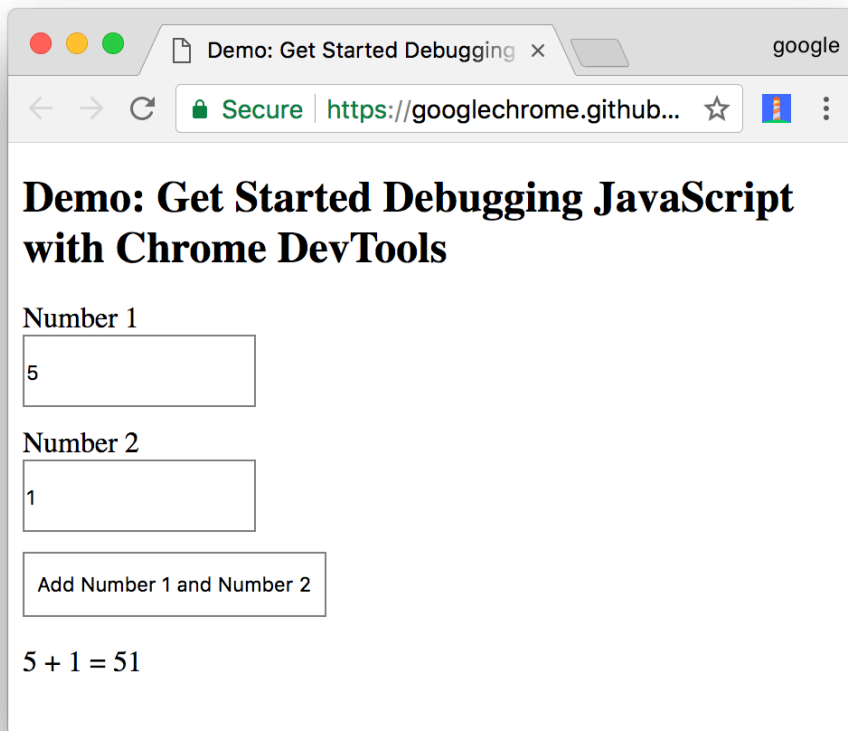


# Using the Chrome F12 Debugger

## Step 1: Reproduce the bug

Finding a series of actions that consistently reproduces a bug is always the first step to debugging.

1. Click **Open Demo**. The demo opens in a new tab.
2. Enter 5 in the **Number 1** text box.
3. Enter 1 in the **Number 2** text box.
4. Click **Add Number 1 and Number 2**. The label below the button says  $5 + 1 = 51$ . The result should be 6. This is the bug you're going to fix.

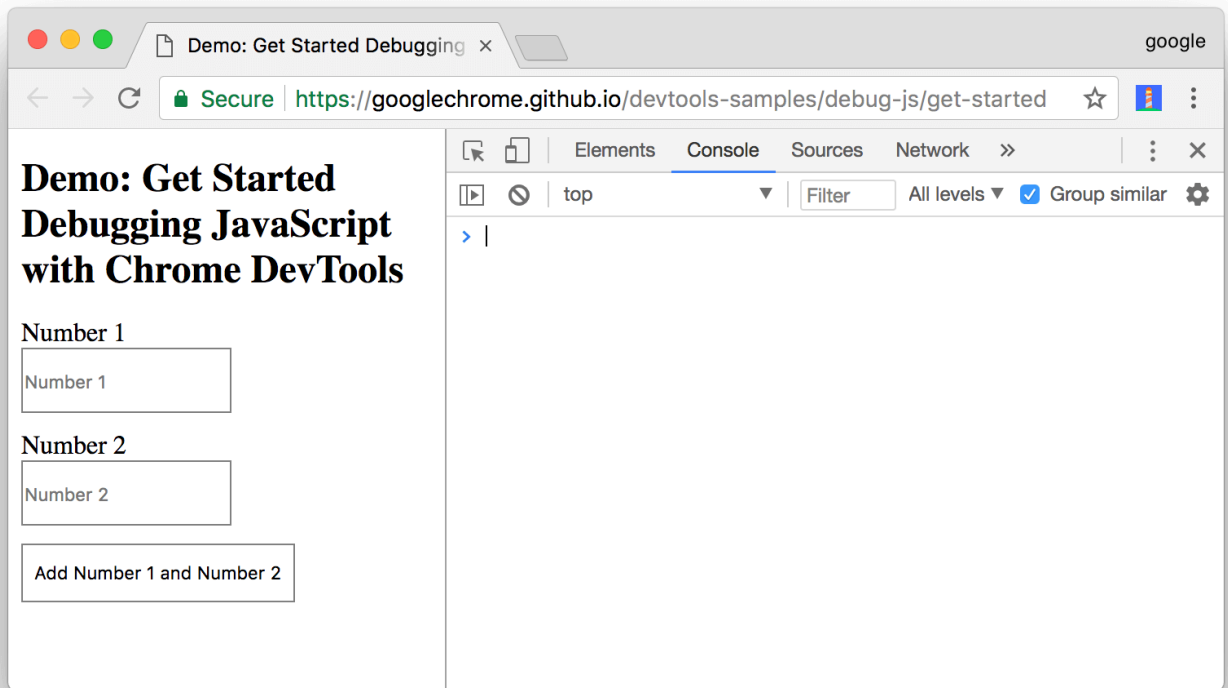


**Figure 1.** The result of  $5 + 1$  is 51. It should be 6.

## Step 2: Get familiar with the Sources panel UI

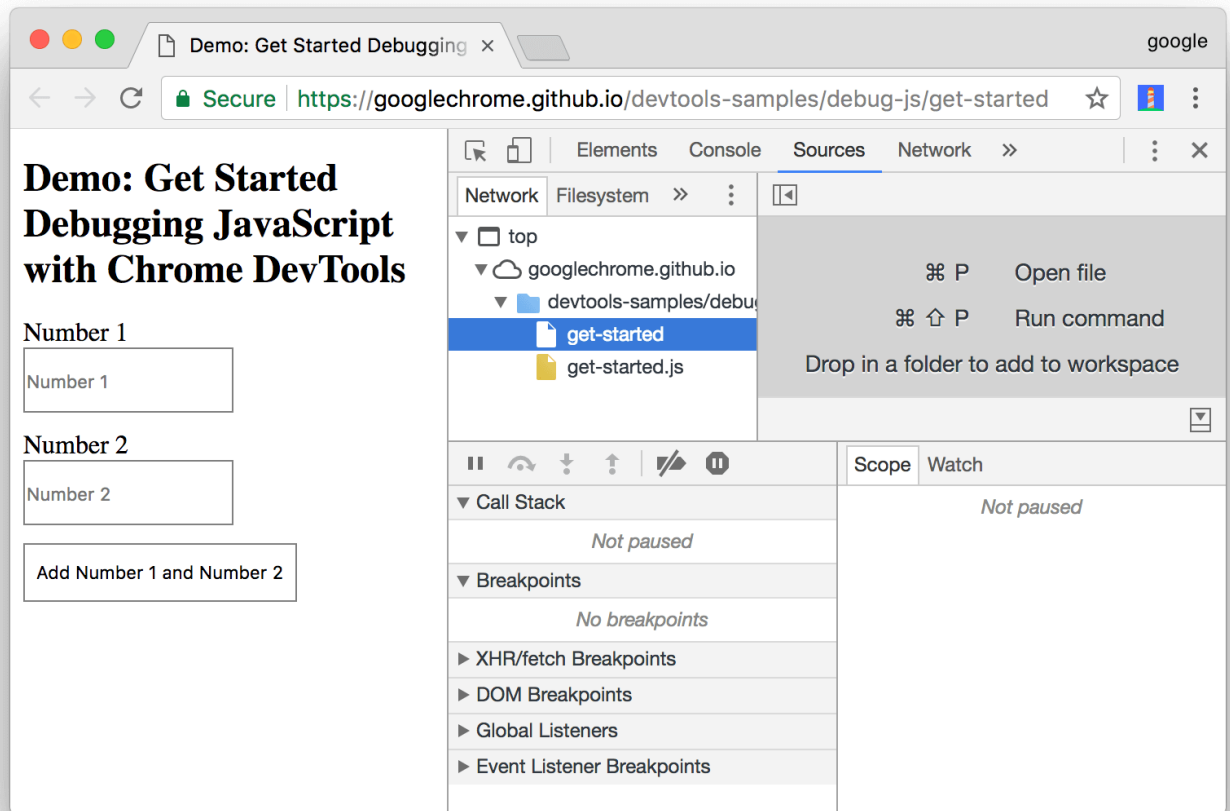
DevTools provides a lot of different tools for different tasks, such as changing CSS, profiling page load performance, and monitoring network requests. The **Sources** panel is where you debug JavaScript.

1. Open DevTools by pressing **Command+Option+I** (Mac) or **Control+Shift+I** (Windows, Linux). This shortcut opens the **Console** panel.



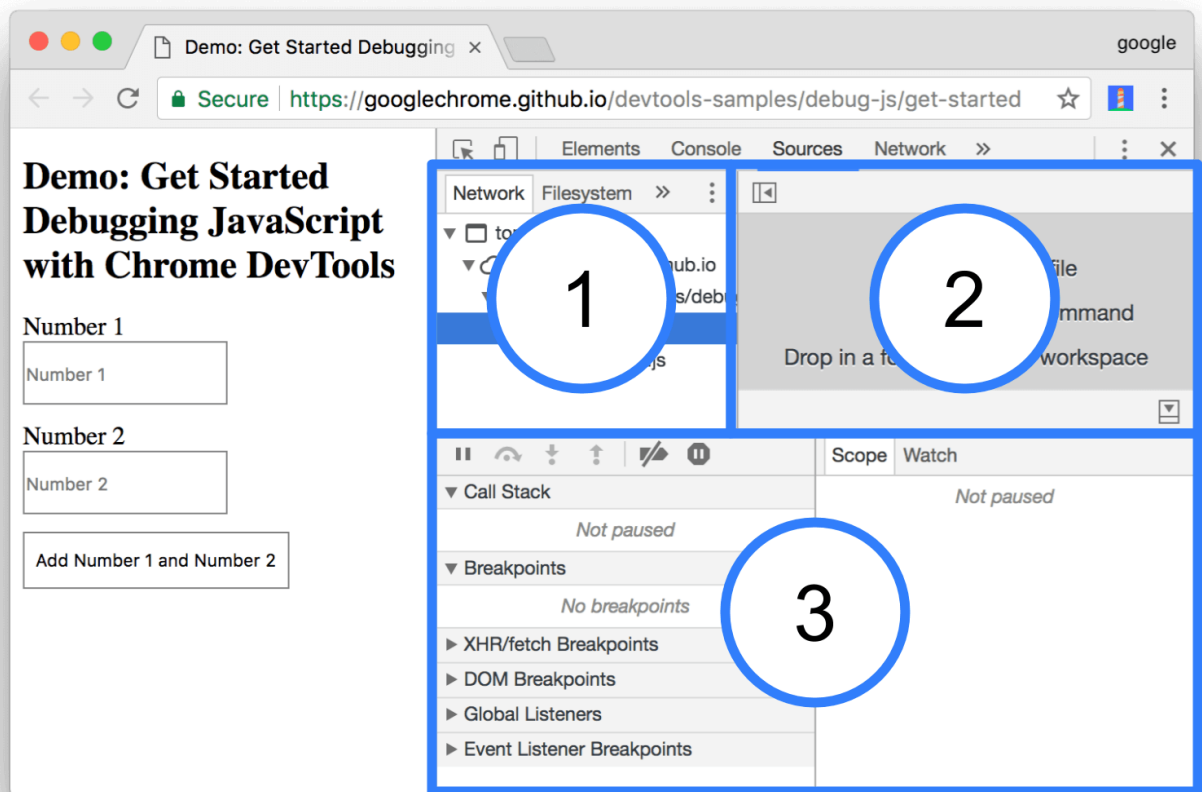
**Figure 2.** The **Console** panel

2. Click the **Sources** tab.



**Figure 3.** The **Sources** panel

The **Sources** panel UI has 3 parts:



**Figure 4.** The 3 parts of the **Sources** panel UI

1. The **File Navigator** pane. Every file that the page requests is listed here.
2. The **Code Editor** pane. After selecting a file in the **File Navigator** pane, the contents of that file are displayed here.
3. The **JavaScript Debugging** pane. Various tools for inspecting the page's JavaScript. If your DevTools window is wide, this pane is displayed to the right of the **Code Editor** pane.

### Step 3: Pause the code with a breakpoint

A common method for debugging a problem like this is to insert a lot of `console.log()` statements into the code, in order to inspect values as the script executes. For example:


```
function updateLabel() {  
  var addend1 = getNumber1();  
  console.log('addend1:', addend1);  
  var addend2 = getNumber2();  
  console.log('addend2:', addend2);  
  var sum = addend1 + addend2;  
  console.log('sum:', sum);  
  label.textContent = addend1 + ' + ' + addend2 + ' = ' + sum;  
}
```

The `console.log()` method may get the job done, but **breakpoints** can get it done faster. A breakpoint lets you pause your code in the middle of its execution, and examine all values at that moment in time. Breakpoints have a few advantages over the `console.log()` method:

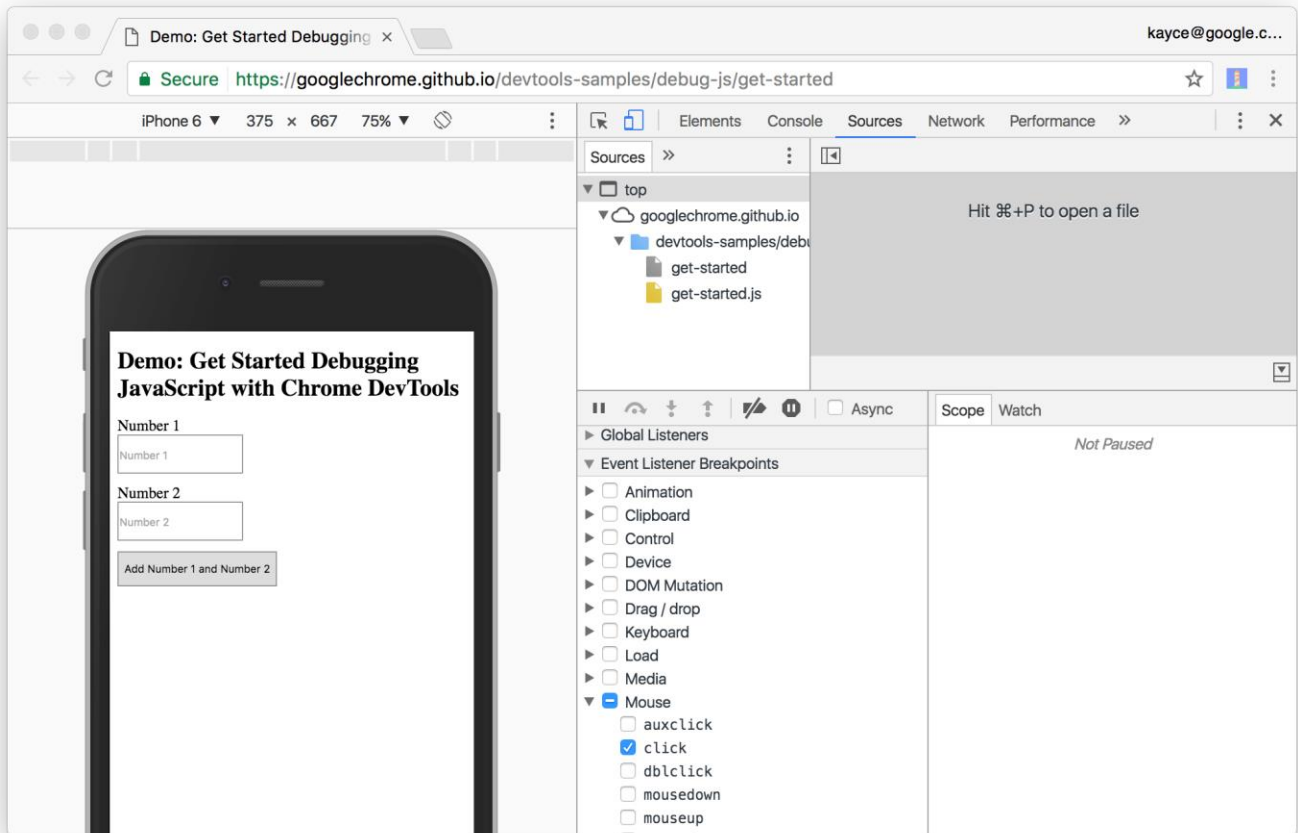
- With `console.log()`, you need to manually open the source code, find the relevant code, insert the `console.log()` statements, and then reload the page in order to see the messages in the Console. With breakpoints, you can pause on the relevant code without even knowing how the code is structured.
- In your `console.log()` statements you need to explicitly specify each value that you want to inspect. With breakpoints, DevTools shows you the values of all variables at that moment in time. Sometimes there are variables affecting your code that you're not even aware of.

In short, breakpoints can help you find and fix bugs faster than the `console.log()` method.

If you take a step back and think about how the app works, you can make an educated guess that the incorrect sum ( $5 + 1 = 51$ ) gets computed in the `click` event listener that's associated to the **Add Number 1 and Number 2** button. Therefore, you probably want to pause the code around the time that the `click` listener executes. **Event Listener Breakpoints** let you do exactly that:

1. In the **JavaScript Debugging** pane, click **Event Listener Breakpoints** to expand the section. DevTools reveals a list of expandable event categories, such as **Animation** and **Clipboard**.
2. Next to the **Mouse** event category, click **Expand** . DevTools reveals a list of mouse events, such as **click** and **mousedown**. Each event has a checkbox next to it.

3. Check the **click** checkbox. DevTools is now set up to automatically pause when *any* click event listener executes.



**Figure 5.** The **click** checkbox is enabled

4. Back on the demo, click **Add Number 1 and Number 2** again. DevTools pauses the demo and highlights a line of code in the **Sources** panel. DevTools should be paused on this line of code:

```
function onClick() {
```

If you're paused on a different line of code, press **Resume Script Execution**  until you're paused on the correct line.

**Note:** If you paused on a different line, you have a browser extension that registers a **click** event listener on every page that you visit. You were paused in the extension's **click** listener. If you use Incognito Mode to [browse in private](#), which disables all extensions, you can see that you pause on the correct line of code every time.

**Event Listener Breakpoints** are just one of many types of breakpoints available in DevTools. It's worth memorizing all the different types, because each type ultimately helps you debug different scenarios as quickly as possible. See [Pause Your Code With Breakpoints](#) to learn when and how to use each type.

## Step 4: Step through the code

One common cause of bugs is when a script executes in the wrong order. Stepping through your code enables you to walk through your code's execution, one line at a time, and figure out exactly where it's executing in a different order than you expected. Try it now:

1. On the **Sources** panel of DevTools, click **Step into next function call**  to step through the execution of the `onClick()` function, one line at a time. DevTools highlights the following line of code:

```
if (inputsAreEmpty()) {
```

2. Click **Step over next function call** . DevTools executes `inputsAreEmpty()` without stepping into it. Notice how DevTools skips a few lines of code. This is because `inputsAreEmpty()` evaluated to false, so the `if` statement's block of code didn't execute.

That's the basic idea of stepping through code. If you look at the code in `get-started.js`, you can see that the bug is probably somewhere in the `updateLabel()` function. Rather than stepping through every line of code, you can use another type of breakpoint to pause the code closer to the probable location of the bug.

## Step 5: Set a line-of-code breakpoint


Line-of-code breakpoints are the most common type of breakpoint. When you've got a specific line of code that you want to pause on, use a line-of-code breakpoint:

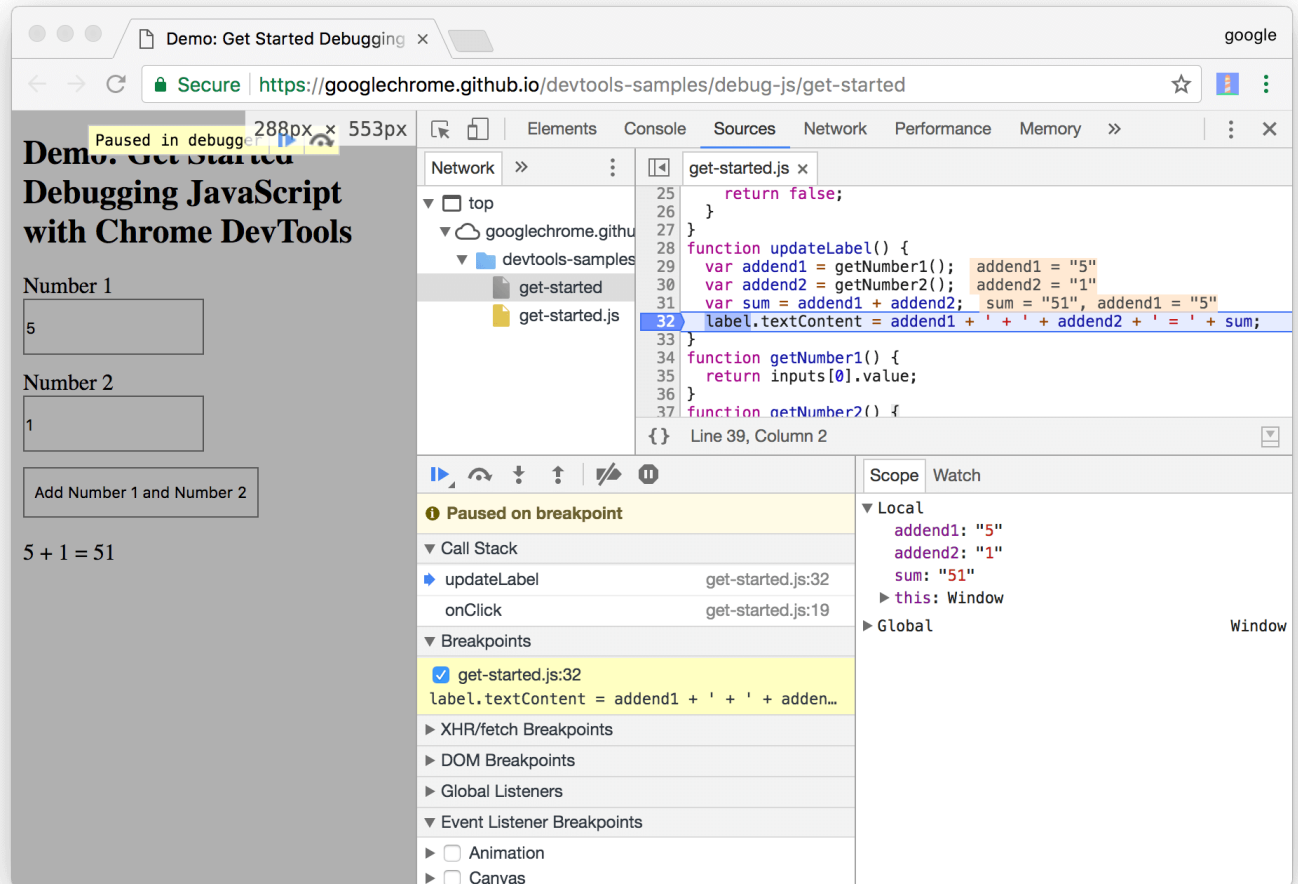
1. Look at the last line of code in `updateLabel()`:

```
label.textContent = addend1 + ' + ' + addend2 + ' = ' + sum;
```

2. To the left of the code you can see the line number of this particular line of code, which is **32**. Click on **32**. DevTools puts a blue icon on top of **32**. This means that there is a line-of-code breakpoint on this line. DevTools now always pauses before this line of code is executed.



3. Click **Resume script execution** . The script continues executing until it reaches line 32. On lines 29, 30, and 31, DevTools prints out the values of `addend1`, `addend2`, and `sum` to the right of each line's semi-colon.



**Figure 6.** DevTools pauses on the line-of-code breakpoint on line 32

## Step 6: Check variable values

The values of `addend1`, `addend2`, and `sum` look suspicious. They're wrapped in quotes, which means that they're strings. This is a good hypothesis for the explaining the cause of the bug. Now it's time to gather more information. DevTools provides a lot of tools for examining variable values.

### Method 1: The Scope pane

When you're paused on a line of code, the **Scope** pane shows you what local and global variables are currently defined, along with the value of each variable. It also shows closure variables, when applicable. Double-click a variable value to edit it. When you're not paused on a line of code, the **Scope** pane is empty.

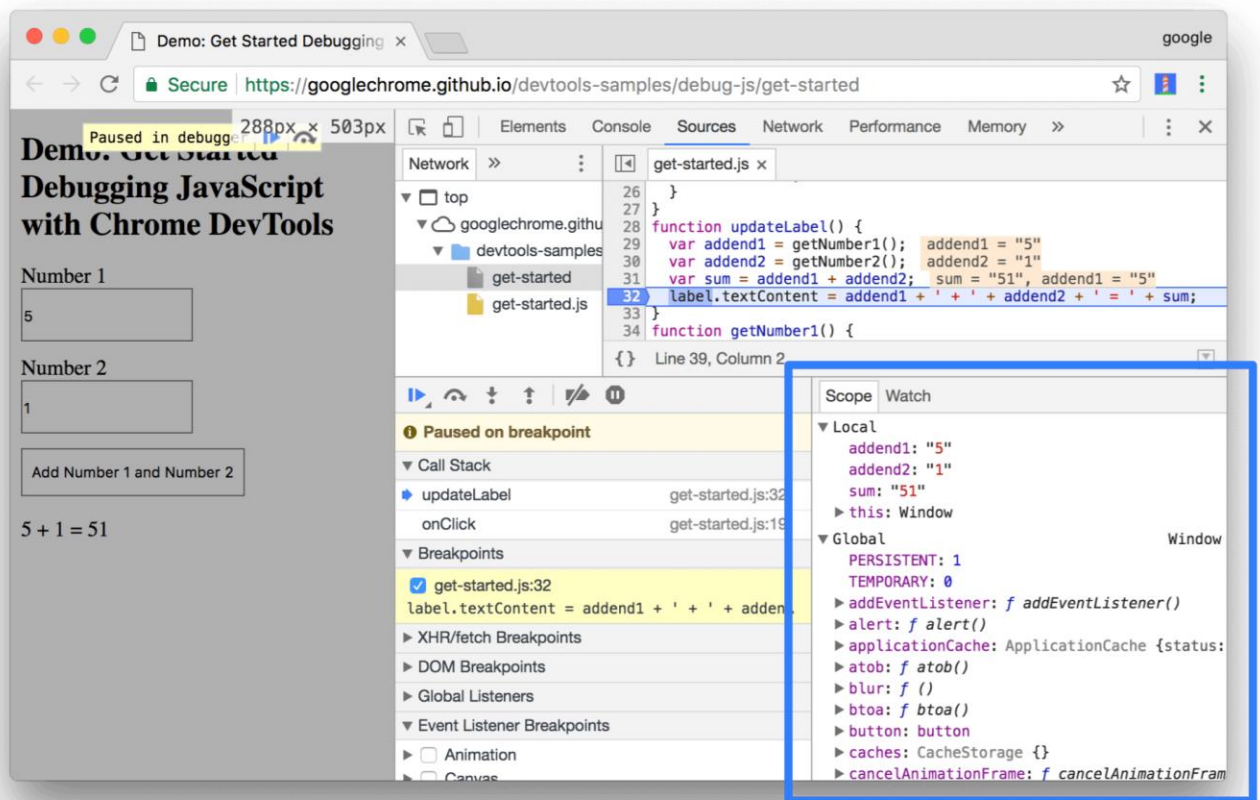

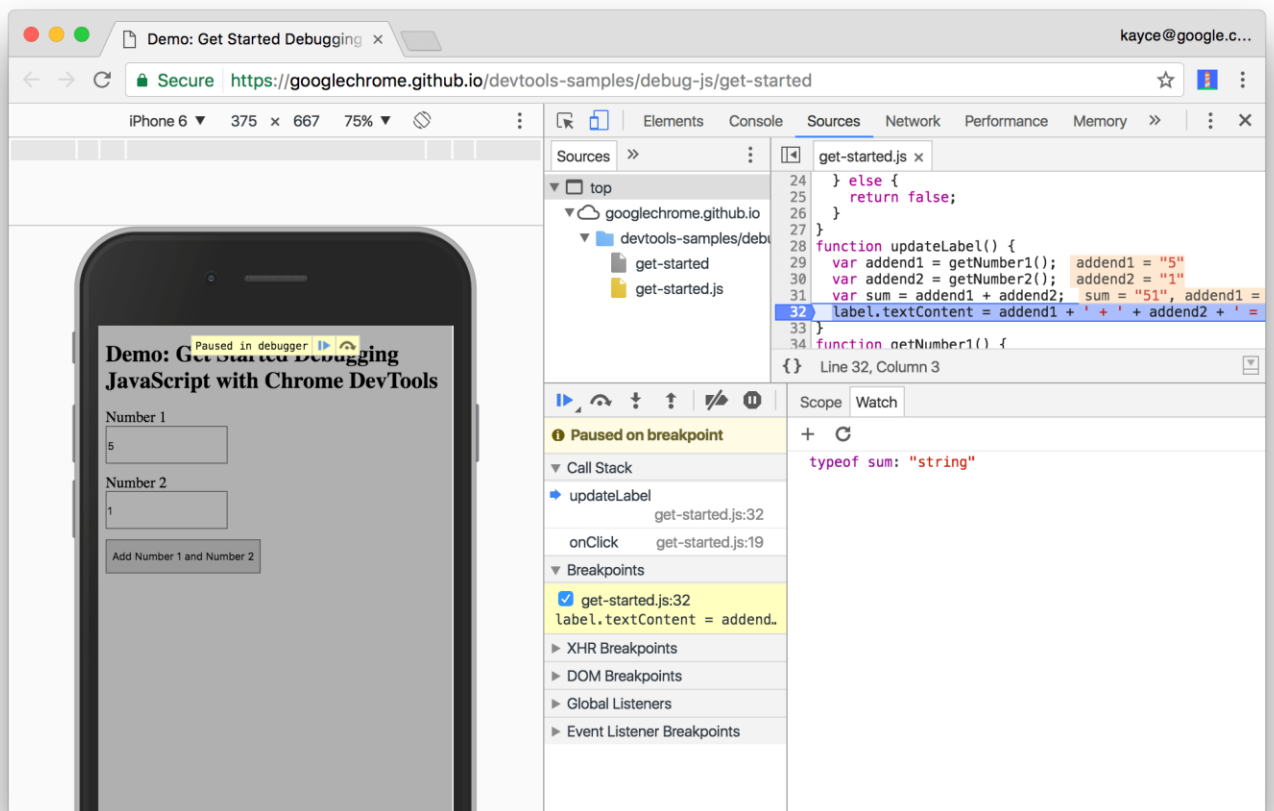


Figure 7. The **Scope** pane

### Method 2: Watch Expressions

The **Watch Expressions** tab lets you monitor the values of variables over time. As the name implies, Watch Expressions aren't just limited to variables. You can store any valid JavaScript expression in a Watch Expression. Try it now:

1. Click the **Watch** tab.
2. Click **Add Expression** .
3. Type `typeof sum`.
4. Press Enter. DevTools shows `typeof sum: "string"`. The value to the right of the colon is the result of your Watch Expression.



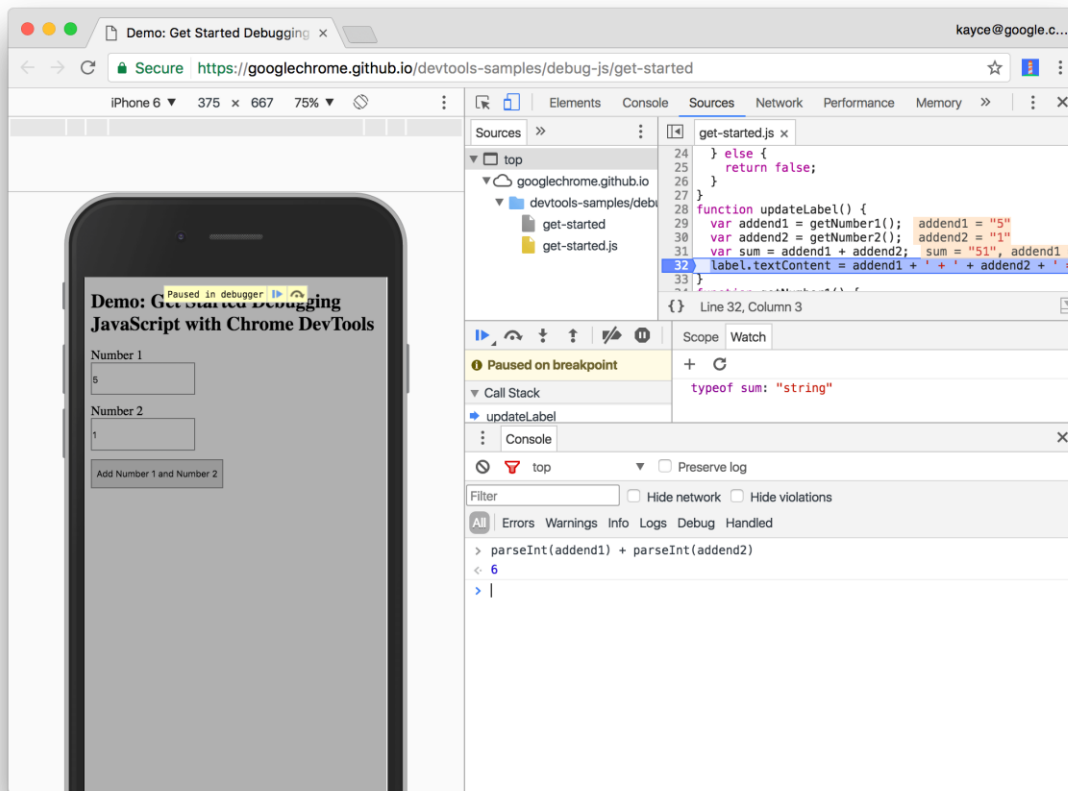
**Figure 8.** The Watch Expression pane (bottom-right), after creating the `typeof sum` Watch Expression. If your DevTools window is large, the Watch Expression pane is on the right, above the **Event Listener Breakpoints** pane.

As suspected, `sum` is being evaluated as a string, when it should be a number. You've now confirmed that this is the cause of the bug.

### Method 3: The Console

In addition to viewing `console.log()` messages, you can also use the Console to evaluate arbitrary JavaScript statements. In terms of debugging, you can use the Console to test out potential fixes for bugs. Try it now:



1. If you don't have the Console drawer open, press `Escape` to open it. It opens at the bottom of your DevTools window.
2. In the Console, type `parseInt(addend1) + parseInt(addend2)`. This statement works because you are paused on a line of code where `addend1` and `addend2` are in scope.
3. Press `Enter`. DevTools evaluates the statement and prints out `6`, which is the result you expect the demo to produce.



**Figure 9.** The Console drawer, after evaluating `parseInt(addend1) + parseInt(addend2)`.

## Step 7: Apply a fix

You've found a fix for the bug. All that's left is to try out your fix by editing the code and re-running the demo. You don't need to leave DevTools to apply the fix. You can edit JavaScript code directly within the DevTools UI. Try it now:

1. Click **Resume script execution** .
2. In the **Code Editor**, replace line 31, `var sum = addend1 + addend2`, with `var sum = parseInt(addend1) + parseInt(addend2)`.
3. Press `Command+S` (Mac) or `Control+S` (Windows, Linux) to save your change.
4. Click **Deactivate breakpoints** . It changes blue to indicate that it's active. While this is set, DevTools ignores any breakpoints you've set.
5. Try out the demo with different values. The demo now calculates correctly.

**Caution:** This workflow only applies a fix to the code that is running in your browser. It won't fix the code for all users that visit your page. To do that, you need to fix the code that's on your servers.

## Next steps

Congratulations! You now know how to make the most of Chrome DevTools when debugging JavaScript. The tools and methods you learned in this section can save you countless hours.

DevTools offers many other ways, including:

- Conditional breakpoints that are only triggered when the condition that you provide is true.
- Breakpoints on caught or uncaught exceptions.
- XHR breakpoints that are triggered when the requested URL matches a substring that you provide.

See [Pause Your Code With Breakpoints](#) to learn when and how to use each type.

There's a couple of code stepping controls that weren't explained in this section. See [Step over line of code](#) to learn more.

# Using Webpack with Create React App

In most of our earlier projects, we loaded React with script tags in our apps' index.html files:

```
<script src='vendor/react.js'></script>
<script src='vendor/react-dom.js'></script>
```

Because we've been using ES6, we've also been loading the Babel library with script tags:

```
<script src='vendor/babel-standalone.js'></script>
```

With this setup we've been able to load in any ES6 JavaScript file we wanted in index.html, specifying that its type is text/babel:

```
<script type='text/babel' src='./client.js'></script>
```

Babel would handle the loading of the file, transpiling our ES6 JavaScript to browser-ready ES5 JavaScript.

We began with this setup strategy because it's the simplest. You can begin writing React components in ES6 with little setup.

However, this approach has limitations. For our purposes, the most pressing limitation is the lack of support for JavaScript modules.

## JavaScript modules

We saw modules in earlier apps. For instance, the time tracking app had a Client module. That module's file defined a few functions, like getTimers(). It then set window.client to an object that "exposed" each function as a property. That object looked like this:

```
// `window.client` was set to this object
// Each property is a function
{ getTimers,
  createTimer,
  updateTimer,
  startTimer,
  stopTimer,
  deleteTimer,
};
```

This Client module only exposed these functions. These are the Client module's public methods. The file public/js/client.js also contained other function definitions, like checkStatus(), which verifies that the server returned a 2xx response code. While each of the public methods uses checkStatus() internally, checkStatus() is kept private. It is only accessible from within the module.

That's the idea behind a module in software. You have some self-contained component of a software system that is responsible for some discrete functionality. The module exposes a limited interface to the rest of the system, ideally the minimum viable interface the rest of the system needs to effectively use the module.

In React, we can think of each of our individual components as their own modules. Each component is responsible for some discrete part of our interface. React components might contain their own state or perform complex operations, but the interface for all of them is the same: they accept inputs (props) and output their DOM representation (render). Users of a React component need not know any of the internal details.

In order for our React components to be truly modular, we'd ideally have them live in their own files. In the upper scope of that file, the component might define a styles object or helper functions that only the component uses. But we want our component-module to only expose the component itself.

Until ES6, modules were not natively supported in JavaScript. Developers would use a variety of different techniques to make modular JavaScript. Some solutions only work in the browser, relying on the browser environment (like the presence of window). Others only work in Node.js.

Browsers don't yet support ES6 modules. But ES6 modules are the future. The syntax is intuitive, we avoid bizarre tactics employed in ES5, and they work both in and outside of the browser. Because of this, the React community has quickly adopted ES6 modules.



If you look at `time_tracking_app/public/js/client.js`, you'll get an idea of how strange the techniques for creating ES5 JavaScript modules are.

However, due to the complexity of module systems, we can't simply use ES6's import/export syntax and expect it to "just work" in the browser, even with Babel. More tooling is needed.

For this reason and more, the JavaScript community has widely adopted JavaScript bundlers. As we'll see, JavaScript bundlers allow us to write modular ES6 JavaScript that works seamlessly in the browser. But that's not all. Bundlers pack numerous advantages. Bundlers provide a strategy for both organizing and distributing web apps. They have powerful toolchains for both iterating in development and producing production-optimized builds.

While there are several options for JavaScript bundlers, the React community's favorite is Webpack.

However, bundlers like Webpack come with a significant trade-off: They add complexity to the setup of your web application. Initial configuration can be difficult and you ultimately end up with an app that has more moving pieces.

In response to setup and configuration woes, the community has created loads of boilerplates and libraries developers can use to get started with more advanced React apps. But the React core team recognized that as long as there wasn't a core team sanctioned solution, the community was likely to remain splintered. The first steps for a bundler-powered React setup can be confusing for novice and experienced developers alike.

The React core team responded by producing the Create React App project.

## Create React App

The `create-react-app` library provides a command you can use to initiate a new Webpack-powered React app:

```
$ create-react-app my-app-name
```

The library will configure a “black box” Webpack setup for you. It provides you with the benefits of a Webpack setup while abstracting away the configuration details.

Create React App is a great way to get started with a Webpack-React app using standard conventions. Therefore, we’ll use it in all of our forthcoming Webpack-React apps.

In this module, we’ll:

- See what a React component looks like when represented as an ES6 module
- Examine the setup of an app managed by Create React App
- Take a close look at how Webpack works
- Explore some of the numerous advantages that Webpack provides for both development and production use
- Peek under the hood of Create React App
- Figure out how to get a Webpack-React app to work alongside an API



The idea of a “black box” controlling the inner-workings of your app might be scary. This is a valid concern. Later in the module, we’ll explore a feature of Create React App, `eject`, which should hopefully assuage some of this fear.



# Exploring Create React App

Let's install Create React App and then use it to initialize a Webpack-React app. We can install it globally from the command line using the `-g` flag. You can run this command anywhere on your system:

```
$ npm i -g create-react-app@1.4.1
```



The `@1.4.1` above is used to specify a version number. We recommend using this version as it is the same version we tested the code in this course with.

Now, anywhere on your system, you can run the `create-react-app` command to initiate the setup for a new Webpack-powered React app.

Let's create a new app. We'll do this inside of the code download that came with the course. From the root of the code folder, change into the directory for this module:

```
$ cd webpack
```

That directory already has three folders:

```
$ ls
```

```
es6-modules/  
food-lookup/  
heart-webpack-complete/
```

The completed version of the code for this next section is available in `heart-webpack-complete`.

Run the following command to initiate a new React app in a folder called `heart-webpack`:

```
$ create-react-app heart-webpack --scripts-version=1.0.14
```

This will create the boilerplate for the new app and install the app's dependencies. This might take a while.

The `--scripts-version` flag above is important. We want to ensure your `react-scripts` version is the same as the one we're using here in the course. We'll see what the `react-scripts` package is in a moment.

When Create React App is finished, cd into the new directory:

```
$ cd heart-webpack
```

```
$ ls
```

```
README.md  
node_modules/  
package.json  
public/  
src/
```

Inside src/ is a sample React app that Create React App has provided for demonstration purposes. Inside of public/ is an index.html, which we'll look at first.

## public/index.html

Opening public/index.html in a text editor:

```
<!doctype html>  
<html lang="en"> <head>  
  <meta charset="utf-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1">  
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico"> <!--  
    ... comment omitted ...  
  -->  
  <title>React App</title> </head>  
  <body>  
    <div id="root"></div>  
    <!--  
      ... comment omitted ...  
    -->  
  </body>  
</html>
```

The stark difference from the index.html we've used in previous apps: there are no script tags here. That means this file is not loading any external JavaScript files. We'll see why this is soon.

## package.json

Looking inside of the project's package.json, we see a few dependencies and some script definitions:

webpack/heart-webpack-complete/package.json

---

```
{
  "name": "heart-webpack",
  "version": "0.1.0",
  "private": true,
  "devDependencies": {
    "react-scripts": "1.1.1"
  },
  "dependencies": {
    "react": "16.3.0",
    "react-dom": "16.3.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

---

Let's break it down.

### react-scripts

package.json specifies a single development dependency, react-scripts:

webpack/heart-webpack-complete/package.json

---

```
"devDependencies": {
  "react-scripts": "1.1.1"
},
```

---

Create React App is just a boilerplate generator. That command produced the folder structure of our new React app, inserted a sample app, and specified our package.json. It's actually the reactscripts package that makes everything work.

react-scripts specifies all of our app's development dependencies, like Webpack and Babel. Furthermore, it contains scripts that “glue” all of these dependencies together in a conventional manner.



Create React App is just a boilerplate generator. The react-scripts package, specified in package.json, is the engine that will make everything work.



Even though react-scripts is the engine, throughout the module we'll continue to refer to the overall project as Create React App.

## react and react-dom

Under dependencies, we see react and react-dom listed:

webpack/heart-webpack-complete/package.json

---

```
"dependencies": {  
  "react": "16.3.0",  
  "react-dom": "16.3.0"  
},
```

---

Webpack gives us the ability to use npm packages in the browser. We can specify external libraries that we'd like to use in package.json. This is incredibly helpful. Not only do we now have easy access to a vast library of packages. We also get to use npm to manage all the libraries that our app uses. We'll see in a bit how this all works.

## Scripts

package.json specifies four commands under scripts. Each executes a command with reactscripts. Over this module and the next we'll cover each of these commands in depth, but at a high-level:

- **start:** Boots the Webpack development HTTP server. This server will handle requests from our web browser.
- **build:** For use in production, this command creates an optimized, static bundle of all our assets.
- **test:** Executes the app's test suite, if present.
- **eject:** Moves the innards of react-scripts into your project's directory. This enables you to abandon the configuration that react-scripts provides, tweaking the configuration to your liking.

For those weary of the black box that react-scripts provides, that last command is comforting. You have an escape hatch should your project “outgrow” react-scripts or should you need some special configuration.



In a package.json, you can specify which packages are necessary in which environment. Note that react-scripts is specified under devDependencies.

When you run `npm i`, npm will check the environment variable `NODE_ENV` to see if it's installing packages in a production environment. In production, npm only installs packages listed under dependencies (in our case, react and react-dom). In development, npm installs all packages. This speeds the process in production, foregoing the installation of unneeded packages like linters or testing libraries.

Given this, you might wonder: Why is react-scripts listed as a development dependency? How will the app work in a production environment without it? We'll see why this is after taking a look at how Webpack prepares production builds.

### src/

Inside of src/, we see some JavaScript files:

```
$ ls src
```

```
App.css  
App.js  
App.test.js  
index.css  
index.js  
logo.svg
```

Create React App has created a boilerplate React app to demonstrate how files can be organized. This app has a single component, App, which lives inside App.js.

## App.js

Looking inside src/App.js:

webpack/heart-webpack-complete/src/App.js

---

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';
class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload. </p>
        </div>
      );
    }
  }
}
export default App;
```

---

There are a few noteworthy features here.

### The import statements

We import React and Component at the top of the file:

webpack/heart-webpack-complete/src/App.js

---

```
import React, { Component } from 'react';
```

---

This is the ES6 module import syntax. Webpack will infer that by 'react' we are referring to the npm package specified in our package.json.

The next two imports may have surprised you:

```
webpack/heart-webpack-complete/src/App.js
```

---

```
import logo from './logo.svg';  
import './App.css';
```

---

We're using import on files that aren't JavaScript! Webpack has you specify all your dependencies using this syntax. We'll see later how this comes into play. Because the paths are relative (they are preceded with `./`), Webpack knows we're referring to local files and not npm packages.

**App** is an ES6 module

The App component itself is simple and does not employ state or props. Its return method is just markup, which we'll see rendered in a moment.

What's special about the App component is that it's an ES6 module. Our App component lives inside its own dedicated App.js. At the top of this file, it specifies its dependencies and at the bottom it specifies its export:

```
webpack/heart-webpack-complete/src/App.js
```

---

```
export default App;
```

---

Our React component is entirely self-contained in this module. Any additional libraries, styles, and images could be specified at the top. Any developer could open this file and quickly reason what dependencies this component has. We could define helper functions that are private to the component, inaccessible to the outside.

Furthermore, recall that there is another file in `src/` related to App besides App.css: App.test.js. So, we have three files corresponding to our component: The component itself (an ES6 module), a dedicated stylesheet, and a dedicated test file.

Create React App has suggested a powerful organization paradigm for our React app. While perhaps not obvious in our single-component app, you can imagine how this modular component model is intended to scale well as the number of components grows to the hundreds or thousands.

We know where our modular component is defined. But we're missing a critical piece: Where is the component written to the DOM?

The answer lies inside src/index.js.

## index.js

Open src/index.js now:

```
webpack/heart-webpack-complete/src/index.js
```

---

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

---

Stepping through this file, we first import both react and react-dom. Because we specified App as the default export from App.js, we can import it here. Note again that the relative path (./App) signals to Webpack that we're referring to a local file, not an npm package.

At this point, we can use our App component just as we have in the past. We make a call to ReactDOM.render(), rendering the component to the DOM on the root div. This div tag is the one and only div present in index.html.

This layout is certainly more complicated than the one we used in our first couple of projects. Instead of just rendering App right below where we define it, we have another file that we're importing App into and making the ReactDOM.render() call in. Again, this setup is intended to keep our code modular. App.js is restricted to only defining a React component. It does not carry the additional responsibility of rendering that component. Following from this pattern, we could comfortably import and render this component anywhere in our app.

We now know where the ReactDOM.render() call is located. But the way this new setup works is still opaque. index.html does not appear to load in any JavaScript. How do our JavaScript modules make it to the browser?

Let's boot the app and then explore how everything fits together.



Why do we import React at the top of the file? It doesn't apparently get referenced anywhere.

React actually is referenced later in the file, we just can't see it because of a layer of indirection. We're referencing App using JSX. So this line in JSX:

```
<App />
```

Is actually this underneath the JSX abstraction:

```
React.createElement(App, null);
```



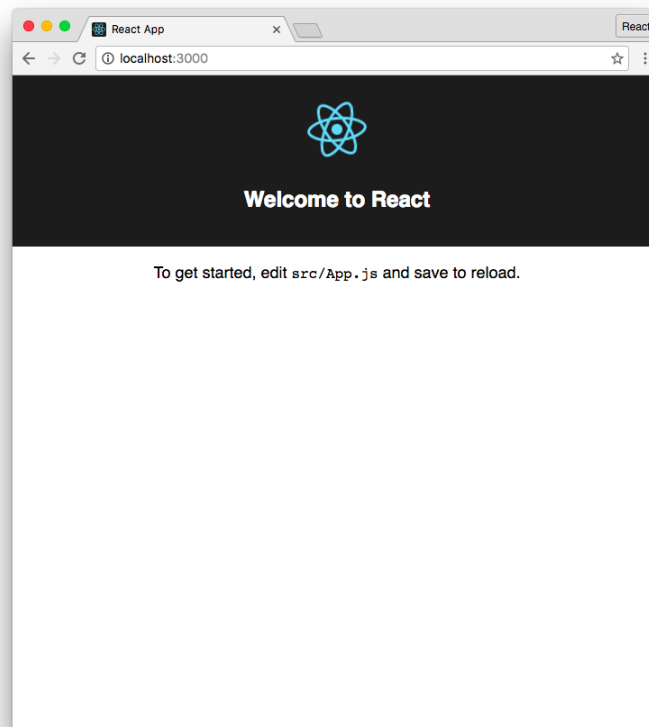
## Booting the app

From the root of heart-webpack, run the start command:

```
$ npm start
```

This boots the Webpack development server. We dig into the details of this server momentarily.

Visiting <http://localhost:3000/>, we see the interface for the sample app that Create React App has provided:



### The sample app

The App component is clearly present on the page. We see both the logo and text that the component specifies. How did it get there?

Let's view the source code behind this page. In both Chrome and Firefox, you can type `viewsource:http://localhost:3000/` into the address bar to do so:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="/favicon.ico">
    <!--
      ... comment omitted ...
    -->
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <!--
      ... comment omitted ...
    -->
    <script type="text/javascript" src="/static/js/bundle.js"></script></body>
</html>
```

This `index.html` looks the same as the one we looked at earlier, save for one key difference: There is a **script** tag appended to the bottom of the **body**. This script tag references a `bundle.js`. As we'll see, the App component from `App.js` and the `ReactDOM.render()` call from `index.js` both live inside of that file.

The Webpack development server inserted this line into our **index.html**. To understand what `bundle.js` is, let's dig into how Webpack works.



This script defaults the server's port to 3000. However, if it detects that 3000 is occupied, it will choose another. The script will tell you where the server is running, so check the console if it appears that it is not on `http://localhost:3000/`.



If you're running OS X, this script will automatically open a browser window pointing to `http://localhost:3000/`.

# Webpack basics

In our first app (the voting app), we used the library http-server to serve our static assets, like index.html, our JavaScript files, and our images.

In the second app (the timers app), we used a small Node server to serve our static assets. We defined a server in server.js which both provided a set of API endpoints and served all assets under public/. Our API server and our static asset server were one in the same.

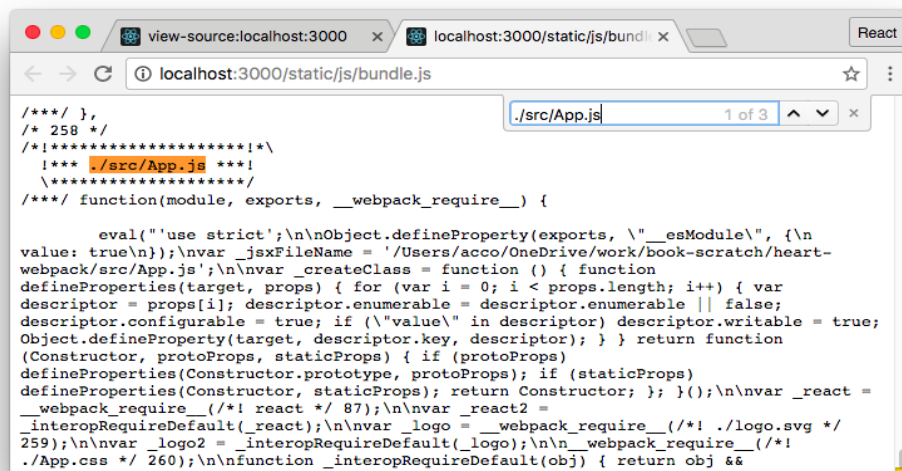
With Create React App, our static assets are served by the Webpack development server that is booted when we run npm start. At the moment, we're not working with an API.

As we saw, the original index.html did not contain any references to the React app. Webpack inserted a reference to bundle.js in index.html before serving it to our browser. If you look around on disk, bundle.js doesn't exist anywhere. The Webpack development server produces this file on the fly and keeps it in memory. When our browser makes a request to localhost:3000/, Webpack is serving its modified version of index.html and bundle.js from memory.

From the page view-source:http://localhost:3000/, you can click on /static/js/bundle.js to open that file in your browser. It will probably take a few seconds to open. It's a gigantic file.

bundle.js contains all the JavaScript code that our app needs to run. Not only does it contain the entire source for App.js — it contains the entire source for the React library!

You can search for the string ./src/App.js in this file. Webpack demarcates each separate file it has included with a special comment. What you'll find is quite messy:



If you do a little hunting, you can see recognizable bits and pieces of App.js amidst the chaos. This is indeed our component. But it looks nothing like it.

Webpack has performed some transformation on all the included JavaScript. Notably, it used Babel to transpile our ES6 code to an ES5-compatible format.

If you look at the comment header for App.js, it has a number. In the screenshot above, that number was 258:

```
/* 258 */
/*!*****!\ !***
./src/App.js ***!
\*****/
```



Your module IDs might be different than the ones here in the text.

The module itself is encapsulated inside of a function that looks like this:

```
function(module, exports, __webpack_require__) { // The
  chaotic `App.js` code here
}
```

Each module of our web app is encapsulated inside of a function with this signature. Webpack has given each of our app's modules this function container as well as a module ID (in the case of App.js, 258).

But “module” here is not limited to JavaScript modules.

Remember how we imported the logo in App.js, like this:

```
webpack/heart-webpack-complete/src/App.js
```

---

```
import logo from './logo.svg';
```

---

And then in the component's markup it was used to set the src on an img tag:

```
webpack/heart-webpack-complete/src/App.js
```

---

```
<img src={logo} className="App-logo" alt="logo" />
```

---

Here's what the variable declaration of logo looks like inside the chaos of the App.js Webpack module:

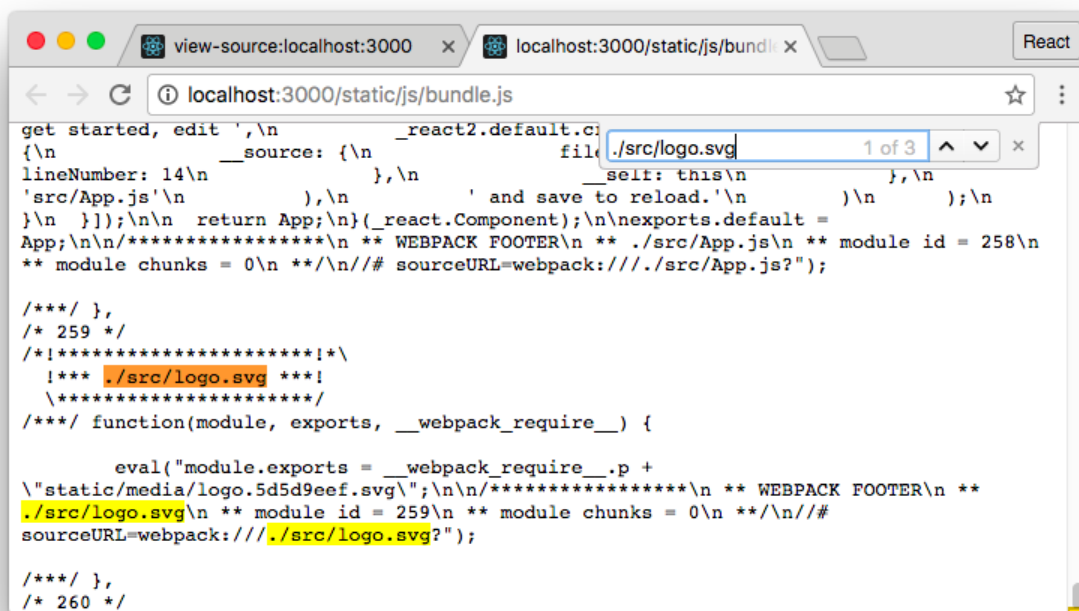
```
var _logo = __webpack_require__(/*! ./logo.svg */ 259);
```

This looks quite strange, mostly due to the in-line comment that Webpack provides for debugging purposes. Removing that comment:

```
var _logo = __webpack_require__(259);
```

Instead of an import statement, we have plain old ES5 code. What is this doing though?

To find the answer, search for `./src/logo.svg` in this file. (It should appear directly below `App.js`). The SVG is represented inside `bundle.js`, too!



Looking at the header for this module:

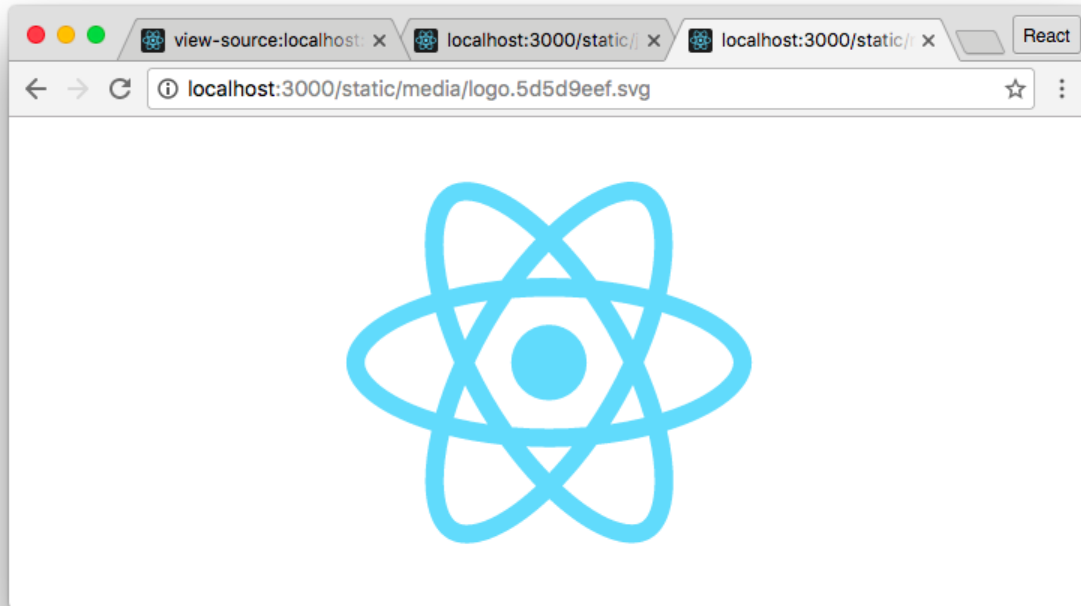
```
/* 259 */
/*!*****!*\ !***
./src/logo.svg ***!
\*****/
```

Note that its module ID is 259, the same integer passed to `__webpack_require__()` above.

Webpack treats everything as a module, including image assets like `logo.svg`. We can get an idea of what's going on by picking out a path in the mess of the `logo.svg` module. Your path might be different, but it will look like this: `static/media/logo.5d5d9eef.svg`

If you open a new browser tab and plug in this address:  
<http://localhost:3000/static/media/logo.5d5d9eef.svg>

You should get the React logo:



So Webpack created a Webpack module for logo.svg by defining a function. While the implementation details of this function are opaque, we know it refers to the path to the SVG on the Webpack development server. Because of this modular paradigm, it was able to intelligently compile a statement like this:

```
import logo from './logo.svg';
```

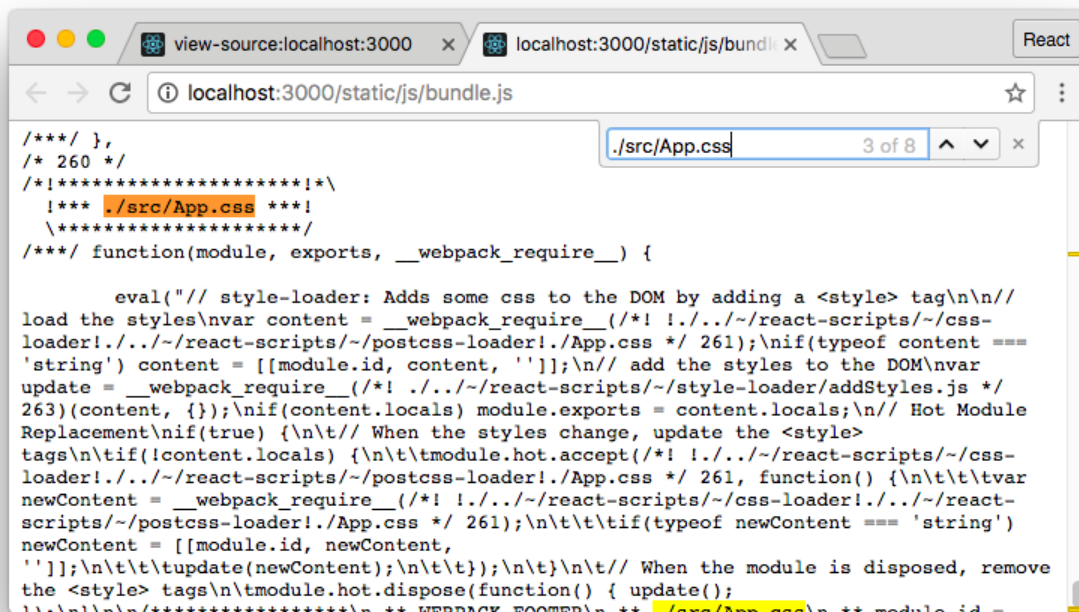
Into this ES5 statement:

```
var _logo = __webpack_require__(259);
```

`__webpack_require__()` is Webpack's special module loader. This call refers to the Webpack module corresponding to logo.svg, number 259. That module returns the string path to the logo's location on the Webpack development server, `static/media/logo.5d5d9eef.svg`:

```
var _logo = __webpack_require__(259);  
console.log(_logo);  
// -> "static/media/logo.5d5d9eef.svg"
```

What about our CSS assets? Yep, everything is a module in Webpack. Search for the string `./src/App.css`:



Webpack's `index.html` didn't include any references to CSS. That's because Webpack is including our CSS here via `bundle.js`. When our app loads, this cryptic Webpack module function dumps the contents of `App.css` into style tags on the page.

So we know what is happening: Webpack has rolled up every conceivable “module” for our app into `bundle.js`. You might be asking: Why?

The first motivation is universal to JavaScript bundlers. Webpack has converted all our ES6 modules into its own bespoke ES5-compatible module syntax.

Furthermore, Webpack, like other bundlers, consolidated all our JavaScript modules into a single file. While it could keep JavaScript modules in separate files, having a single file maximizes performance. The initiation and conclusion of each file transfer over HTTP adds overhead. Bundling up hundreds or thousands of smaller files into one bigger one produces a notable speed-up.

Webpack takes this module paradigm further than other bundlers, however. As we saw, it applies the same modular treatment to image assets, CSS, and npm packages (like React and ReactDOM). This modular paradigm unleashes a lot of power. We touch on aspects of that power throughout the rest of this module.

With our nascent understanding of how Webpack works, let's turn our attention back to the sample app. We'll make some modifications and see first-hand how the Webpack development process works.

## Making modifications to the sample app

We've been checking out the bundle.js produced by the Webpack development server in our browser. Recall that to boot this server, we ran the following command:

```
$ npm start
```

As we saw, this command was defined in package.json:

```
"start": "react-scripts start",
```

What exactly is going on here?

The react-scripts package defines a start script. Think of this start script as a special interface to Webpack that contains some features and conventions provided by Create React App. At a highlevel, the start script:

- Sets up the Webpack configuration
- Provides some nice formatting and coloring for Webpack's console output
- Launches a web browser if you're on OS X

Let's take a look at what the development cycle of our Webpack-powered React app looks like.



## Hot reloading

If the server is not already running, go ahead and run the start command to boot it:

```
$ npm start
```

Again, our app launches at `http://localhost:3000/`. The Webpack development server is listening on this port and serves the development bundle when our server makes a request.

One compelling development feature Webpack gives us is hot reloading. Hot reloading enables certain files in a web app to be hot-swapped on the fly whenever changes are detected without requiring a full page reload.

At the moment, Create React App only sets up hot reloading for CSS. This is because the Reactspecific hot reloader is not considered stable enough for the default setup.

Hot reloading for CSS is wonderful. With your browser window open, make an edit to `App.css` and witness as the app updates without a refresh.

For example, you can change the speed at which the logo spins. Here, we changed it from 20s to 1s:

```
.App-logo { animation: App-logo-spin infinite 1s  
  linear; height: 80px;  
}
```

Or you can change the color of the header's text. Here, we changed it from white to purple:

```
.App-header { background-  
  color: #222; height: 150px;  
  padding: 20px; color:  
  purple;  
}
```

### How hot reloading works

Webpack includes client-side code to perform hot reloading inside `bundle.js`. The Webpack client maintains an open socket with the server. Whenever the bundle is modified, the client is notified via this websocket. The client then makes a request to the server, asking for a patch to the bundle. Instead of fetching the whole bundle, the server will just send the client the code that client needs to execute to “hot swap” the asset.

Webpack's modular paradigm makes hot reloading of assets possible. Recall that Webpack inserts CSS into the DOM inside `style` tags. To swap out a modified CSS asset, the client removes the previous `style` tags and inserts the new one. The browser renders the modification for the user, all without a page reload.

## Auto-reloading

Even though hot reloading is not supported for our JavaScript files, Webpack will still auto-reload the page whenever it detects changes.

With our browser window still open, let's make a minor edit to `src/App.js`. We'll change the text in the `p` tag:

```
<p className="App-intro">
```

```
  I just made a change to <code>src/App.js</code>!
```

```
</p>
```

Save the file. You'll note the page refreshes shortly after you save and your change is reflected.

Because Webpack is at heart a platform for JavaScript development and deployment, there is an ever-growing ecosystem of plug-ins and tools for Webpack-powered apps.

For development, hot- and auto-reloading are two of the most compelling plug-ins that come configured with Create React App. In the later section on eject ("Ejecting"), we'll point to the Create React App configuration file that sets up Webpack for development so that you can see the rest.

For deployment, Create React App has configured Webpack with a variety of plug-ins that produce a production-level optimized build. We'll take a look at the production build process next.

## Creating a production build

So far, we've been using the Webpack development server. In our investigation, we saw that this server produces a modified `index.html` which loads a `bundle.js`. Webpack produces and serves this file from memory — nothing is written to disk.

For production, we want Webpack to write a bundle to disk. We'll end up with a production-optimized build of our HTML, CSS, and JavaScript. We could then serve these assets using whatever HTTP server we wanted. To share our app with the world, we'd just need to upload this build to an asset host, like Amazon's S3.

Let's take a look at what a production build looks like.

Quit the server if it's running with `CTRL+C`. From the command line, run the build command that we saw in `package.json` earlier:

```
$ npm run build
```

When this finishes, you'll note a new folder is created in the project's root: build. cd into that directory and see what's inside:

```
$ cd build
$ ls
favicon.ico
index.html
static/
```

If you look at this index.html, you'll note that Webpack has performed some additional processing that it did not perform in development. Most notably: there are no newlines. The entire file is on a single line. Newlines are not necessary in HTML and are just extra bytes. We don't need them in production.

Here's what that exact file looks like in a human readable format:

```
<!/DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta content="width=device-width,initial-scale=1" name="viewport">
    <link href="/favicon.ico?fd73a6eb" rel="shortcut icon">
    <title>React App</title>
    <link href="/static/css/main.9a0fe4f1.css" rel="stylesheet">
  </head>
  <body>
    <div id="root"></div>
    <script src="/static/js/main.590bf8bb.js" type="text/javascript"> </script>
  </body>
</html>
```

Instead of referencing a bundle.js, this index.html references a file in static/ which we'll look at momentarily. What's more, this production index.html now has a link tag to a CSS bundle. As we saw, in development Webpack inserts CSS via bundle.js. This feature enables hot reloading. In production, hot reloading capability is irrelevant. Therefore, Webpack deploys CSS normally.



Webpack versions assets. We can see above that our JavaScript bundle has a different name and is versioned (main.<version>.js).

Asset versioning is useful when dealing with browser caches in production. If a file is changed, the version of that file will be changed as well. Client browsers will be forced to fetch the latest version.

Note that your versions (or digests) for the files above may be different.

The static/ folder is organized as follows:

```
$ ls static
css/
js/
media/
```

Checking out the folders individually:

```
$ ls static/css
main.9a0fe4f1.css
main.9a0fe4f1.css.map
```

```
$ ls static/js
main.f7b2704e.js
main.f7b2704e.js.map
```

```
$ ls static/media
logo.5d5d9eef.svg
```

Feel free to open up both the .css file and the .js file in your text editor. We refrain from printing them here in the course due to their size.



Be careful about opening these files—you might crash your editor due to their size!

If you open the CSS file, you'll see it's just two lines: The first line is all of our app's CSS, stripped of all superfluous whitespace. We could have hundreds of different CSS files in our app and they would end up on this single line. The second line is a special comment declaring the location of the map file.

The JavaScript file is even more packed. In development, bundle.js has some structure. You can pick apart where the individual modules live. The production build does not have this structure.

What's more, our code has been both minified and uglified. If you're unfamiliar with minification or uglification, see the aside "[Minification, uglification, and source maps](#)."

Last, the media folder will contain all of our app's other static files, like images and videos. This app only has one image, the React logo SVG file.

Again, this bundle is entirely self-contained and ready to go. If we wanted to, we could install the same http-server package that we used in the first application and use it to serve this folder, like this:

```
http-server ./build -p 3000
```

Without the Webpack development server, you can imagine the development cycle would be a bit painful:

1. Modify the app
2. Run `npm run build` to generate the Webpack bundle
3. Boot/restart the HTTP server

This is why there is no way to "build" anything other than a bundle intended for production. The Webpack server services our needs for development.

## Minification, uglification, and source maps

For production environments, we can significantly reduce the size of JavaScript files by converting them from a human-readable format to a more compact one that behaves exactly the same. The basic strategy is stripping all excess characters, like spaces. This process is called minification.

Uglification (or obfuscation) is the process of deliberately modifying JavaScript files so that they are harder for humans to read. Again, the actual behavior of the app is unchanged. Ideally, this process slows down the ability for outside developers to understand your codebase.

Both the .css and .js files are accompanied by a companion file ending in .map. The .map file is a source map that provides debugging assistance for production builds. Because they've been minified and uglified, the CSS and JavaScript for a production app are difficult to work with. If you encounter a JavaScript bug on production, for example, your browser will direct you to a cryptic line of this obscure code.

Through a source map, you can map this puzzling area of the codebase back to its original, unbuilt form. For more info on source maps and how to use them, see the blog post "[Introduction to JavaScript Source Maps](#)<sup>a</sup>."

---

<sup>a</sup><http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/>

# Ejecting

When first introducing Create React App at the beginning of this module, we noted that the project provided a mechanism for “ejecting” your app.

This is comforting. You might find yourself in a position in the future where you would like further control over your React-Webpack setup. An eject will copy all the scripts and configuration encapsulated in react-scripts into your project’s directory. It opens up the “black box,” handing full control of your app back over to you.

Performing an eject is also a nice way to strip some of the “magic” from Create React App. We’ll perform an eject in this section and take a quick look around.



There is no backing out from an eject. Be careful when using this command. Should you decide to eject in the future, make sure your app is checked in to source control.

If you’ve been adding to the app inside heart-webpack, you might consider duplicating that directory before proceeding. For example, you can do this:

```
cp -r heart-webpack heart-webpack-ejected
```

The node\_modules folder does not behave well when it’s moved wholesale like this, so you’ll need to remove node\_modules and re-install:

```
cd heart-webpack-ejected  
rm -rf node_modules  
npm i
```

You can then perform the steps in this section inside of heart-webpack-ejected and preserve heart-webpack.

## Buckle up

From the root of heart-webpack, run the eject command:

```
$ npm run eject
```

Confirm you’d like to eject by typing y and hitting enter.

After all the files are copied from react-scripts into your directory, npm install will run. This is because, as we’ll see, all the dependencies for react-scripts have been dumped into our package.json.

When the npm install is finished, take a look at our project's directory:

```
$ ls
README.md
build/
config/
node_modules/
package.json
public/
scripts/
src/
```

We have two new folders: config/ and scripts/. If you looked inside src/ you'd note that, as expected, it is unchanged.

Take a look at package.json. There are loads of dependencies. Some of these dependencies are necessary, like Babel and React. Others — like eslint and whatwg-fetch — are more “nice-to-haves.” This reflects the ethos of the Create React App project: an opinionated starter kit for the React developer.

Check out scripts/ next:

```
$ ls scripts
build.js
start.js
test.js
```

When we ran npm start and npm run build earlier, we were executing the scripts start.js and build.js, respectively. We won't look at these files here in the course, but feel free to peruse them. While complicated, they are well-annotated with comments. Simply reading through the comments can give you a good idea of what each of these scripts are doing (and what they are giving you “for free”).

Finally, check out config/ next:

```
$ ls config
env.js
jest/
paths.js
polyfills.js
webpack.config.dev.js
webpack.config.prod.js
```

react-scripts provided sensible defaults for the tools that it provides. In package.json, it specifies configuration for Babel. Here, it specifies configuration for Webpack and Jest (the testing library we use in the next module).

Of particular noteworthiness are the configuration files for Webpack. Again, we won't dive into those here. But these files are well-commented. Reading through the comments can give you a good idea of what the Webpack development and production pipelines look like and what plugins are used. In the future, if you're ever curious about how react-scripts has configured Webpack in development or production, you can refer to the comments inside these files.

Hopefully seeing the "guts" of react-scripts reduces a bit of its mysticism. Testing out eject as we have here gives you an idea of what the process looks like to abandon react-scripts should you need to in the future.

So far in this module, we've covered the fundamentals of Webpack and Create React App's interface to it. Specifically, we've seen:

- How the interface for Create React App works
- The general layout for a Webpack-powered React app
- How Webpack works (and some of the power it provides)
- How Create React App and Webpack help us generate production-optimized builds
- What an ejected Create React App project looks like

There's one essential element our demo Webpack-React app is missing, however.

In our second project (the timers app) we had a React app that interfaced with an API. The node server both served our static assets (HTML/CSS/JS) and provided a set of API endpoints that we used to persist data about our running timers.

As we've seen in this module, when using Webpack via Create React App we boot a Webpack development server. This server is responsible for serving our static assets.

What if we wanted our React app to interface with an API? We'd still want the Webpack development server to serve our static assets. Therefore, we can imagine we'd boot our API and Webpack servers separately. Our challenge then is getting the two to cooperate.