

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in](#) 

Introduction to automated testing

[← Previous](#)[↑ Overview: Cross browser testing](#)[Next →](#)

Manually running tests on several browsers and devices, several times per day, can get tedious, and time consuming. To handle this efficiently, you should become familiar with automation tools. In this article, we look at what is available, how to use task runners, and how to use the basics of commercial browser test automation apps such as, Sauce Labs and Browser Stack.

Prerequisites:	Familiarity with the core HTML, CSS, and JavaScript languages; an idea of the high level principles of cross browser testing.
Objective:	To provide an understanding of what automated testing entails, how it can make your life easier, and how to make use of some of the commercial products that make things easier.


Automation makes things easy

Throughout this module we have detailed loads of different ways in which you can test your websites and apps, and explained the sort of scope your cross-browser testing efforts should have in terms of what browsers to test, accessibility considerations, and more. Sounds like a lot of work, doesn't it?

We agree — testing all the things we've looked at in previous articles manually can be a real pain. Fortunately, there are tools to help us automate some of this pain away. There are two main ways in which we can automate the tests we've been talking about in this module:

1. Use a task runner such as [Grunt](#) or [Gulp](#), or [npm scripts](#) to run tests and clean up code during your build process. This is a great way to perform tasks like linting and minifying code, adding in CSS prefixes or transpiling nascent JavaScript features for maximum cross-browser reach, and so on.
2. Use a browser automation system like [Selenium](#) to run specific tests on installed browsers and return results, alerting you to failures in browsers as they crop up. Commercial cross-browser testing apps like [Sauce Labs](#) and [Browser Stack](#) are based on Selenium, but allow you to access their set up remotely using a simple interface, saving you the hassle of setting up your own testing system.

We will look at how to set up your own Selenium-based testing system in the next article. In this article, we'll look at how to set up a task runner, and use the basic functionality of commercial systems like the ones mentioned above.

 **Note:** the above two categories are not mutually exclusive. It is possible to set up a task runner to access a service like Sauce Labs via an API, run cross browser tests, and return results. We will look at this below as well.

Using a task runner to automate testing tools

As we said above, you can drastically speed up common tasks such as linting and minifying code by using a task runner to run everything you need to run automatically at a certain point in your build process. For example, this could be every time you save a file, or at some other point. Inside this section we'll look at how to automate task running with Node and Gulp, a beginner-friendly option.

Setting up Node and npm


Most tools these days are based on [Node.js](https://nodejs.org/), so you'll need to install it from nodejs.org:

1. Download the installer for your system from the above site. (If you already have Node and npm installed, jump to point 4)
2. Install it like you would any other program. Note that Node comes with [Node Package Manager \(npm\)](#), which allows you to easily install packages, share your own packages with others, and run useful scripts on your projects.
3. Once the install completes, test that node is installed by typing the following into the terminal, which returns the installed versions of Node and npm:

```
1 | node -v
2 | npm -v
```

4. If you've got Node/npm already installed, you should update them to their latest versions. To update Node, the most reliable way is to download and install an updated installer package from their website (see link above). To update npm, use the following command in your terminal:

```
1 | npm install npm@latest -g
```

 **Note:** If the above command fails with permissions errors, [Fixing npm permissions](#) should sort you out.

To start using Node/npm-based packages on your projects, you need to set up your project directories as npm projects. This is easy to do.

For example, let's first create a test directory to allow us to play without fear of breaking anything.

1. Create a new directory somewhere sensible with using your file manager UI, or by navigating to the location you want and running the following command:

```
1 | mkdir node-test
```

2. To make this directory an npm project, you just need to go inside your test directory and initialize it, with the following:

```
1 | cd node-test
2 | npm init
```

3. This second command will ask you many questions to find out the information required to set up the project; you can just select the defaults for now.
4. Once all the questions have been asked, it will ask you if the information entered is OK. type yes and press Enter/Return and npm will generate a `package.json` file in your directory.

This file is basically a config file for the project. You can customize it later, but for now it'll look something like this:

```
1  {
2    "name": "node-test",
3    "version": "1.0.0",
4    "description": "Test for npm projects",
5    "main": "index.js",
6    "scripts": {
7      "test": "test"
8    },
9    "author": "Chris Mills",
10   "license": "MIT"
11 }
```

With this, you are ready to move on.

Setting up Gulp automation

Let's look at setting up Gulp and using it to automate some testing tools.

1. To begin with, create a test npm project using the procedure detailed at the bottom of the previous section.
2. Next, you'll need some sample HTML, CSS and JavaScript content to test your system on — make copies of our sample [index.html](#), [main.js](#), and [style.css](#) files in a subfolder with the name `src` inside your project folder. You can try your own test content if you like, but bear in mind that such tools won't work on internal JS/CSS — you need external files.
3. First, install gulp globally (meaning, it will be available across all projects) using the following command:

```
1 | npm install --global gulp-cli
```

4. Next, run the following command inside your npm project directory root to set up gulp as a dependency of your project:

```
1 | npm install --save-dev gulp
```

5. Now create a new file inside your project directory called `gulpfile.js`. This is the file that will run all our tasks. Inside this file, put the following:

```
1 | var gulp = require('gulp');
2 |
3 | gulp.task('default', function() {
4 |     console.log('Gulp running');
5 | });
```

This requires the `gulp` module we installed earlier, and then runs a basic task that does nothing except for printing a message to the terminal — this is useful for letting us know that Gulp is working. Each gulp task is written in the same basic format — gulp's `task()` method is run, and given two parameters — the name of the task, and a callback function containing the actual code to run to complete the task.

6. You can run your gulp task with the following commands — try this now:

```
1 | gulp
```

Adding some real tasks to Gulp

To add some real tasks to Gulp, we need to think about what we want to do. A reasonable set of basic functionalities to run on our project is as follows:

- `html-tidy`, `css-lint`, and `js-hint` to lint and report/fix common HTML/CSS/JS errors (see [gulp-htmltidy](#), [gulp-csslint](#), [gulp-jshint](#)).
- `Autoprefixer` to scan our CSS and add vendor prefixes only where needed (see [gulp-autoprefixer](#)).
- `babel` to transpile any new JavaScript syntax features to traditional syntax that works in older browsers (see [gulp-babel](#)).

See the links above for full instructions on the different gulp packages we are using.

To use each plugin, you need to first install it via npm, then require any dependencies at the top of the `gulpfile.js` file, then add your test(s) to the bottom of it, and finally add the name of your task inside the `default` task.

Before you go any further, update the default task to this:


```
1 | gulp.task('default', [ ]);
```

Inside the array goes the names of all the tasks you want Gulp to run, once you run the `gulp` command on the command line.

html-tidy

1. Install using the following line:

```
1 | npm install --save-dev gulp-htmltidy
```

 **Note:** `--save-dev` adds the package as a dependency to your project. If you look in your project's `package.json` file, you'll see an entry for it as, it has been added to the `devDependencies` property.

2. Add the following dependencies to `gulpfile.js`:

```
1 | var htmltidy = require('gulp-htmltidy');
```

3. Add the following test to the bottom of `gulpfile.js`:

```
1 | gulp.task('html', function() {  
2 |     return gulp.src('src/index.html')  
3 |         .pipe(htmltidy())  
4 |         .pipe(gulp.dest('build'));  
5 | });
```

4. Add `'html'` as an item inside the array in the `default` task.

Here we are grabbing our development `index.html` file — `gulp.src()` which allows us to grab a source file to do something with.

We next use the `pipe()` function to pass that source to another command to do something else with. We can chain as many of these together as we want. We first run `htmltidy()` on the source, which goes through and fixes errors in our file. The second `pipe()` function writes the output HTML file to the `build` directory.

In the input version of the file, you may have noticed that we put an empty `<p>` element; `htmltidy` has removed this by the time the output file has been created.

Autoprefixer and css-lint

1. Install using the following lines:

```
1 | npm install --save-dev gulp-autoprefixer
2 | npm install --save-dev gulp-csslint
```

2. Add the following dependencies to `gulpfile.js`:

```
1 | var autoprefixer = require('gulp-autoprefixer');
2 | var csslint = require('gulp-csslint');
```

3. Add the following test to the bottom of `gulpfile.js`:

```
1 | gulp.task('css', function() {
2 |     return gulp.src('src/style.css')
3 |         .pipe(csslint())
4 |         .pipe(csslint.formatter('compact'))
5 |         .pipe(autoprefixer({
6 |             browsers: ['last 5 versions'],
7 |             cascade: false
8 |         })))
9 |     .pipe(gulp.dest('build'));
10 | });
```

4. Add `'css'` as an item inside the array in the default task.

Here we grab our `style.css` file, run `csslint` on it (which outputs a list of any errors in your CSS to the terminal), then runs it through `autoprefixer` to add any prefixes needed to make nascent CSS features run in older browsers. At the end of the pipe chain, we output our modified prefixed CSS to the `build` directory. Note that this only works if `csslint` doesn't find any errors — try removing a curly brace from your CSS file and re-running `gulp` to see what output you get!

js-hint and babel

1. Install using the following lines:

```
1 | npm install --save-dev gulp-babel babel-preset-es2015
2 | npm install jshint gulp-jshint --save-dev
```

2. Add the following dependencies to `gulpfile.js`:

```
1 | var babel = require('gulp-babel');  
2 | var jshint = require('gulp-jshint');
```

3. Add the following test to the bottom of `gulpfile.js`:

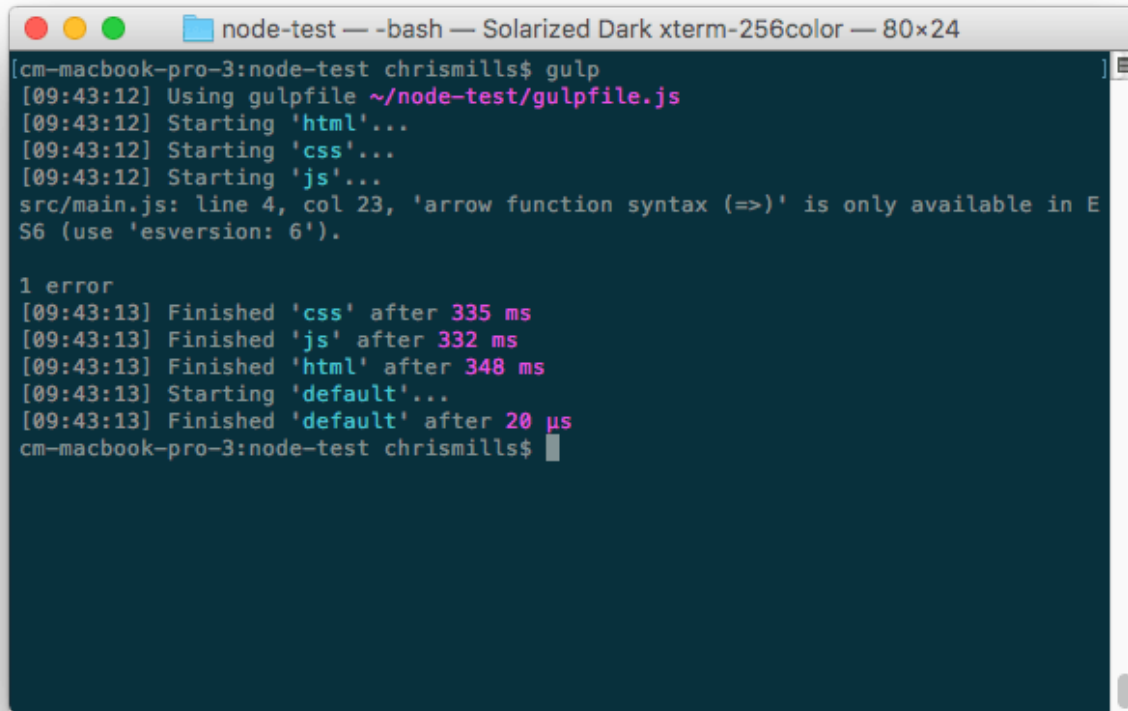
```
1 | gulp.task('js', function() {  
2 |     return gulp.src('src/main.js')  
3 |         .pipe(jshint())  
4 |         .pipe(jshint.reporter('default'))  
5 |         .pipe(babel({  
6 |             presets: ['es2015']  
7 |         })))  
8 |         .pipe(gulp.dest('build'));  
9 | });
```

4. Add `'js'` as an item inside the array in the `default` task.

Here we grab our `main.js` file, run `jshint` on it and output the results to the terminal using `jshint.reporter`; we then pass the file to `babel`, which converts it to old style syntax and outputs the result into the `build` directory. Our original code included a fat arrow function, which `babel` has modified into an old style function.

Further ideas

Once all this is all set up, you can run the `gulp` command inside your project directory, and you should get an output like this:

A terminal window titled 'node-test — -bash — Solarized Dark xterm-256color — 80x24'. The prompt is 'cm-macbook-pro-3:node-test chrismills\$'. The user enters 'gulp'. The output shows Gulp using a gulpfile at '~/node-test/gulpfile.js', starting tasks for 'html', 'css', and 'js', and then a 'default' task. An error message is displayed: 'src/main.js: line 4, col 23, 'arrow function syntax (=>)' is only available in ES6 (use 'esversion: 6')'. The build finishes with timing information: 'css' (335 ms), 'js' (332 ms), 'html' (348 ms), and 'default' (20 μs). The prompt returns to 'cm-macbook-pro-3:node-test chrismills\$'.

You can then try out the files output by your automated tasks by looking at them inside the build directory, and loading `build/index.html` in your web browser.

If you get errors, check that you've added all the dependencies and the tests as shown above; also try commenting out the HTML/CSS/JavaScript code sections and then rerunning gulp to see if you can isolate what the problem is.

Gulp comes with a `watch()` function that you can use to watch your files and run tests whenever you save a file. For example, try adding the following to the bottom of your `gulpfile.js`:

```
1 | gulp.task('watch', function(){
2 |   gulp.watch('src/*.html', ['html']);
3 |   gulp.watch('src/*.css', ['css']);
4 |   gulp.watch('src/*.js', ['js']);
5 | });
```

Now try entering the `gulp watch` command into your terminal. Gulp will now watch your directory, and run the appropriate tasks whenever you save a change to an HTML, CSS, or JavaScript file.

📄 **Note:** The `*` character is a wildcard character — here we're saying "run these tasks when any files of these types are saved. You could also use wildcards in your main tasks, for example `gulp.src('src/*.css')` would grab all your CSS files and then run piped tasks on them.

📄 **Note:** One problem with our watch command above is that our CSSLint/Autoprefixer combination throws full-blown errors when a CSS error is encountered, which stops the watch working. You'll have to restart the watch once a CSS error is encountered, or find another way to do this.

There's a lot more you can do with Gulp. The [Gulp plugin directory](#) has literally thousands of plugins to search through.

Other task runners

There are many other task runners available. We certainly aren't trying to say that Gulp is the best solution out there, but it works for us and it is fairly accessible to beginners. You could also try using other solutions:


- Grunt works in a very similar way to Gulp, except that it relies on tasks specified in a config file, rather than using written JavaScript. See [Getting started with Grunt for more details](#).
- You can also run tasks directly using npm scripts located inside your `package.json` file, without needing to install any kind of extra task runner system. This works on the premise that things like Gulp plugins are basically wrappers around command line tools. So, if you can work out how to run the tools using the command line, you can then run them using npm scripts. It is a bit trickier to work with, but can be rewarding for those who are strong with their command line skills. [Why npm scripts?](#) provides a good introduction with a good deal of further information.

Using commercial testing services to speed up browser testing

Now let's look at commercial 3rd party browser testing services and what they can do for us.

The basic premise with such applications is that the company that runs each one has a huge server farm that can run many different tests. When you use this service, you provide a URL of the page you want to test along with information, such as what browsers you want it tested in. The app then configures a new VM with the OS and browser you specified, and returns the test results in the form of screenshots, videos, logfiles, text, etc.

You can then step up a gear, using an API to access functionality programmatically, which means that such apps can be combined with task runners, such as your own local Selenium environments and others, to create automated tests.

 **Note:** There are other commercial browser testing systems available but in this article, we'll focus on Sauce Labs and BrowserStack. We're not saying that these are necessarily the best tools available, but they are good ones that are simple for beginners to get up and running with.

Sauce Labs

Test your code on Firefox for free on more than 800 browser/OS combos with our partner Sauce Labs.

[Try Sauce Labs now](#)

Getting started with Sauce Labs

Let's get started with a Sauce Labs Trial.

1. Create a [Sauce Labs trial account](#).
2. Sign in. This should happen automatically after you verify your e-mail address.

The basics: Manual tests

The [Sauce Labs dashboard](#) has a lot of options available on it. For now, make sure you are on the *Manual Tests* tab.










1. Click *Start a new manual session*.
2. In the next screen, type in the URL of a page you want to test (use <http://mdn.github.io/learning-area/javascript/building-blocks/events/show-video-box-fixed.html>, for example), then choose a browser/OS combination you want to test by using the different buttons and lists. There is a lot of choice, as you'll see!

New Session

URL

<http://mdn.github.io/learning-area/javascript/building-blocks/events/show-video-box-fixed.htm>

BROWSERS

iPad 2 Simulator

iPad Air 2 Simulator

iPad Air Simulator

iPad Pro Simulator

iPad Retina Simulator

iPad Simulator

iPhone 4s Simulator

iPhone 5 Simulator

iPhone 5s Simulator

iPhone 6 Plus Simulator

iPhone 6 Simulator

iPhone 6s Plus Simulator

iOS 9.3

iOS 9.2

iOS 9.1

iOS 9.0

iOS 8.4

iOS 8.3

iOS 8.2

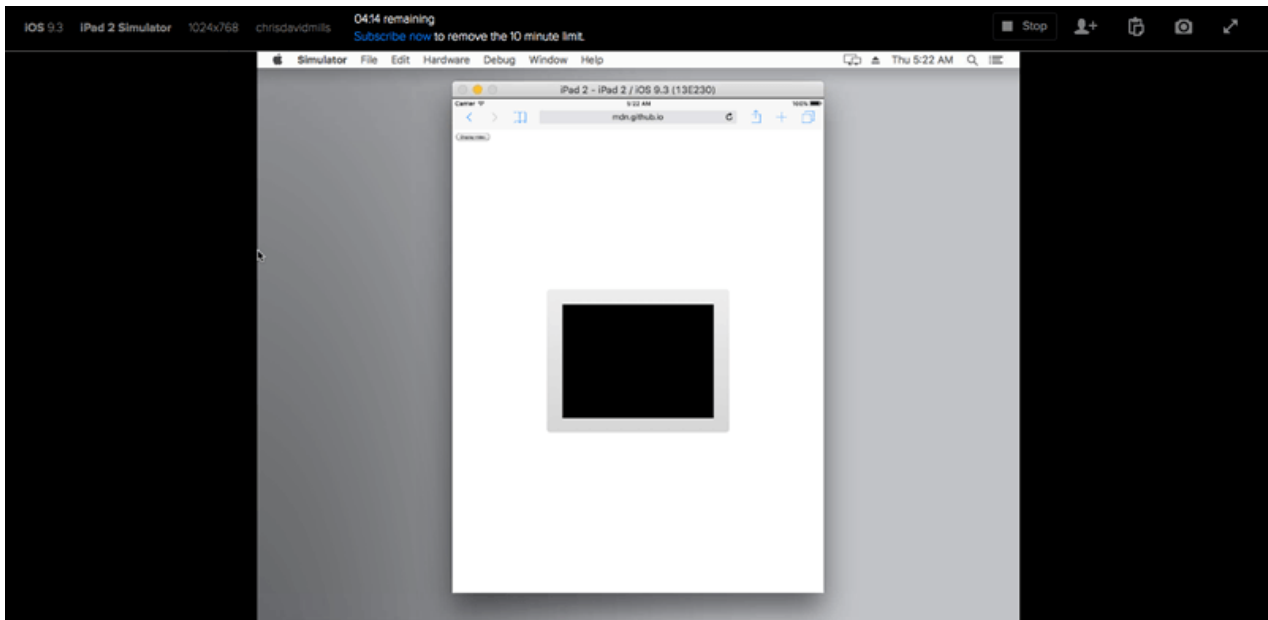
iOS 8.1

SAUCE CONNECT PROXY ?

Select tunnel (optional)

Start Session →

- When you click Start session, a loading screen will then appear, which spins up a virtual machine running the combination you chose.
- When loading has finished, you can then start to remotely test the web site running in the chosen browser.



5. From here you can see the layout as it would look in the browser you are testing, move the mouse around and try clicking buttons, etc. The top menu allows you to:

- Stop the session
- Give someone else a URL so they can observe the test remotely.
- Copy text/notes to a remote clipboard.
- Take a screenshot.
- Test in full screen mode.

Once you stop the session, you'll return to the Manual Tests tab, where you'll see an entry for each of the previous manual sessions you started. Clicking on one of these entries shows more data for the session. In here you can download any screenshots you took , watch a video of the session, and view data logs for the session for example.

Note: This is already very useful, and way more convenient than having to set up all these emulators and virtual machines by yourself.

Advanced: The Sauce Labs API

Sauce Labs has a [restful API](#) that allows you to programmatically retrieve details of your account and existing tests, and annotate tests with further details, such as their pass/fail state which isn't recordable by manual testing alone. For example, you might want to run one of your own Selenium tests remotely using Sauce Labs, to test a certain browser/OS combination, and then pass the test results back to Sauce Labs.

It has several clients available to allow you to make calls to the API using your favourite environment, be it PHP, Java, Node.js, etc.

Let's have a brief look at how we'd access the API using Node.js and [node-saucelabs](#).

1. First, set up a new npm project to test this out, as detailed in [Setting up Node and npm](#). Use a different directory name than before, like `sauce-test` for example.
2. Install the Node Sauce Labs wrapper using the following command:

```
1 | npm install saucelabs
```

3. Create a new file inside your project root called `call_sauce.js`. give it the following contents:

```
1  var SauceLabs = require('saucelabs');
2
3  var myAccount = new SauceLabs({
4    username: "your-sauce-username",
5    password: "your-sauce-api-key"
6  });
7
8  myAccount.getAccountDetails(function (err, res) {
9    console.log(res);
10   myAccount.getServiceStatus(function (err, res) {
11     // Status of the Sauce Labs services
12     console.log(res);
13     myAccount.getJobs(function (err, jobs) {
14       // Get a list of all your jobs
15       for (var k in jobs) {
16         if ( jobs.hasOwnProperty( k )) {
17           myAccount.showJob(jobs[k].id, function (err, res) {
18             var str = res.id + ": Status: " + res.status;
19             if (res.error) {
20               str += "\033[31m Error: " + res.error + " \033[0m";
21             }
22             console.log(str);
23           });
24         }
25       }
26     });
27   });
28 });
```

4. You'll need to fill in your Sauce Labs username and API key in the indicated places. These can be retrieved from your [User Settings](#) page. Fill these in now.

5. Make sure everything is saved, and run your file like so:

```
1 | node call_sauce
```

Advanced: Automated tests

We'll cover actually running automated Sauce Lab tests in the next article.

BrowserStack

Getting started with BrowserStack

Let's get started with a BrowserStack Trial.

1. Create a [BrowserStack](#) trial account.
2. Sign in. This should happen automatically after you verify your e-mail address.
3. When you first sign in, you should be on the Live testing page; if not, click the *Live* link in the top nav menu.
4. If you are on Firefox or Chrome, you'll be prompted to Install a browser extension in a dialog titled "Enable Local Testing" — click the *Install* button to proceed. If you are on other browsers you'll still be able to use some of the features (generally via Flash), but you won't get the full experience.

The basics: Manual tests

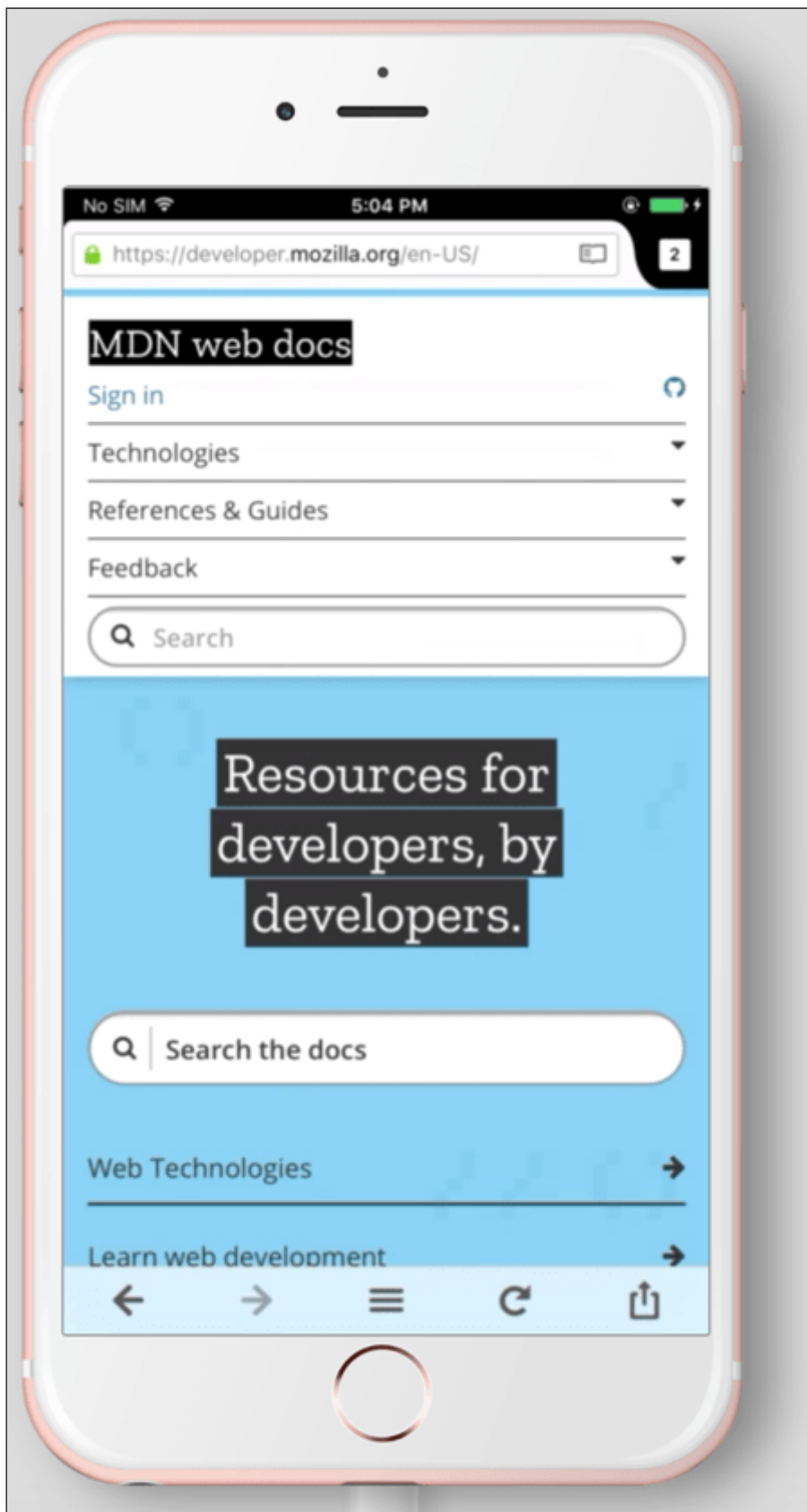
The BrowserStack Live dashboard allows you to choose what device and browser you want to test on — Platforms in the left column, devices on the right. When you mouse over or click on each device, you get a choice of browsers available on that device.

	iPhone		iPad	
Android	iPhone 6S Plus	9	iPad Air 2	8
iOS	iPhone 6S	9	iPad Air	7
	iPhone 6 Plus	8	iPad 4	7
Windows Phone	iPhone 6	8	iPad Mini 3	8
Windows	iPhone 5S	7	iPad Mini 2	7
Mac	iPhone 7 Plus	10.3	iPad Pro	10.3
Sierra	iPhone 7	10.3	iPad Pro	9.3
El Capitan	iPhone SE	10.3	iPad Air 2	9.3
Yosemite	iPhone 5	6	iPad Air	8.3
Mavericks	iPhone 4S	6	iPad Mini 4	9.3
Mountain Lion	iPhone 4S	5.1	iPad Mini 2	8.3
Lion	iPhone 4	4	iPad Mini	7
Snow Leopard	iPhone 3GS	3		

Test on physical devices!
Look for this icon.

Got it

Clicking on one of those browser icons will load up your choice of platform/device/browser — choose one now, and give it a try.

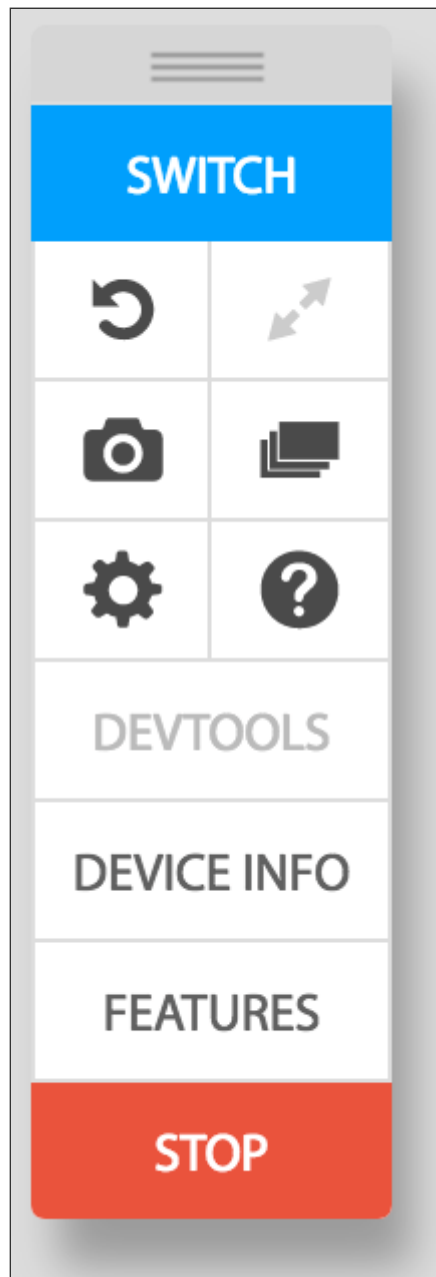


Note: The blue device icon next to some of the mobile device choices signals that you will be testing on a real device; choices without that icon will be run on an emulator.

You'll find that you can enter URLs into the address bar, and use the other controls like you'd expect on a real device. You can even do things like copy and paste from the device

to your clipboard, scroll up and down by dragging with the mouse, or use appropriate gestures (e.g. pinch/zoom, two fingers to scroll) on the touchpads of supporting devices (e.g. Macbook). Note that not all features are available on all devices.


You'll also see a menu that allows you to control the session.



The features here are as follows:

- *Switch* — Change to another platform/device/browser combination.
- Orientation (looks like a Reload icon) — Switch orientation between portrait and landscape.
- Fit to screen (looks like a full screen icon) — Fill the testing areas as much as possible with the device.
- Capture a bug (looks like a camera) — Takes a screenshot, then allows you to annotate and save it.

- Issue tracker (looks like a deck of cards) — View previously captured bugs/screenshots.
- Settings (cog icon) — Allows you to alter general settings for the session.
- Help (question mark) — Accesses help/support functions.
- *Devtools* — Allows you to use your browser's devtools to directly debug or manipulate the page being shown in the test browser. This currently only works when testing the Safari browser on iOS devices.
- *Device info* — Displays information about the testing device.
- *Features* — Shows you what features the current configuration supports, e.g. copy to clipboard, gesture support, etc.
- *Stop* — Ends the session.


 **Note:** This is already very useful, and way more convenient than having to set up all these emulators and virtual machines by yourself.

Other basic features

If you go back to the main BrowserStack page, you'll find a couple of other useful basic features under the *More* menu option:

- *Responsive*: Enter a URL and press *Generate*, and BrowserStack will load that URL on multiple devices with different viewport sizes. Within each device you can further adjust settings like monitor size, to get a good idea of how your site's layout works across different form factors.
- *Screenshots*: Enter a URL, choose the browsers/devices/platforms you are interested in, then press *Generate screenshots* — Browserstack will take screenshots of your site in all those different browsers then make them available to you to view and download.

Advanced: The BrowserStack API

BrowserStack also has a  [restful API](#) that allows you to programmatically retrieve details of your account plan, sessions, builds, etc.

It has several clients available to allow you to make calls to the API using your favourite environment, be it PHP, Java, Node.js, etc.

Let's have a brief look at how we'd access the API using Node.js.

1. First, set up a new npm project to test this out, as detailed in [Setting up Node and npm](#). Use a different directory name than before, like `bstack-test` for example.
2. Create a new file inside your project root called `call_bstack.js`. give it the following contents:

```
1  var request = require("request");
2
3  var bsUser = "BROWSERSTACK_USERNAME";
4  var bsKey = "BROWSERSTACK_ACCESS_KEY";
5  var baseUrl = "https://" + bsUser + ":" + bsKey + "@www.browserst
6
7  function getPlanDetails(){
8      request({uri: baseUrl + "plan.json"}, function(err, res, body)
9          console.log(JSON.parse(body));
10     });
11     /* Response:
12     {
13         automate_plan: <string>,
14         parallel_sessions_running: <int>,
15         team_parallel_sessions_max_allowed: <int>,
16         parallel_sessions_max_allowed: <int>,
17         queued_sessions: <int>,
18         queued_sessions_max_allowed: <int>
19     }
20     */
21 }
22
23 getPlanDetails();
```

3. You'll need to fill in your BrowserStack username and API key in the indicated places. These can be retrieved from your [BrowserStack automation dashboard](#). Fill these in now.
4. Make sure everything is saved, and run your file like so:

```
1 | node call_bstack
```

Below we've also provided some other ready-made functions you might find useful when working with the BrowserStack restful API.

```
1  function getBuilds(){
2      request({uri: baseUrl + "builds.json"}, function(err, res, body){
```

```
3         console.log(JSON.parse(body));
4     });
5     /* Response:
6     [
7         {
8             automation_build: {
9                 name: <string>,
10                duration: <int>,
11                status: <string>,
12                hashed_id: <string>
13            }
14        },
15        {
16            automation_build: {
17                name: <string>,
18                duration: <int>,
19                status: <string>,
20                hashed_id: <string>
21            }
22        },
23        ...
24    ]
25    */
26 };
27
28 function getSessionInBuild(build){
29     var buildId = build.automation_build.hashed_id;
30     request({uri: baseUrl + "builds/" + buildId + "/sessions.json"}, funct
31         console.log(JSON.parse(body));
32     });
33     /* Response:
34     [
35         {
36             automation_session: {
37                 name: <string>,
38                 duration: <int>,
39                 os: <string>,
40                 os_version: <string>,
41                 browser_version: <string>,
42                 browser: <string>,
43                 device: <string>,
44                 status: <string>,
45                 hashed_id: <string>,
```

```
46         reason: <string>,  
47         build_name: <string>,  
48         project_name: <string>,  
49         logs: <string>,  
50         browser_url: <string>,  
51         public_url: <string>,  
52         video_url: <string>,  
53         browser_console_logs_url: <string>,  
54         har_logs_url: <string>  
55     }  
56 },  
57 {  
58     automation_session: {  
59         name: <string>,  
60         duration: <int>,  
61         os: <string>,  
62         os_version: <string>,  
63         browser_version: <string>,  
64         browser: <string>,  
65         device: <string>,  
66         status: <string>,  
67         hashed_id: <string>,  
68         reason: <string>,  
69         build_name: <string>,  
70         project_name: <string>,  
71         logs: <string>,  
72         browser_url: <string>,  
73         public_url: <string>,  
74         video_url: <string>,  
75         browser_console_logs_url: <string>,  
76         har_logs_url: <string>  
77     }  
78 },  
79 ...  
80 ]  
81 */  
82 }  
83  
84 function getSessionDetails(session){  
85     var sessionId = session.automation_session.hashed_id;  
86     request({uri: baseUrl + "sessions/" + sessionId + ".json"}, function(e  
87         console.log(JSON.parse(body));  
88     });
```

```
89      /* Response:
90      {
91          automation_session: {
92              name: <string>,
93              duration: <int>,
94              os: <string>,
95              os_version: <string>,
96              browser_version: <string>,
97              browser: <string>,
98              device: <string>,
99              status: <string>,
100             hashed_id: <string>,
101             reason: <string>,
102             build_name: <string>,
103             project_name: <string>,
104             logs: <string>,
105             browser_url: <string>,
106             public_url: <string>,
107             video_url: <string>,
108             browser_console_logs_url: <string>,
109             har_logs_url: <string>
110         }
111     }
112     */
113 }
```

Advanced: Automated tests

We'll cover actually running automated BrowserStack tests in the next article.

Summary

This was quite a ride, but I'm sure you can start to see the benefit in having automation tools do a lot of the heavy lifting for you in terms of testing.

In the next article, we'll look at setting up our own local automation system using Selenium, and how to combine that with services such as Sauce Labs and BrowserStack

[← Previous](#)[↑ Overview: Cross browser testing](#)[Next →](#)

In this module

- Introduction to cross browser testing
 - Strategies for carrying out testing
 - Handling common HTML and CSS problems
 - Handling common JavaScript problems
 - Handling common accessibility problems
 - Implementing feature detection
 - Introduction to automated testing
 - Setting up your own test automation environment
-