



Fast, easy and reliable testing for anything that runs in a browser

*Last updated: 27 May 2023*

## Contents

Cypress is not Selenium .....	5
Cypress Best Practices.....	6
PageObject example .....	7
Installing Node and Cypress.....	8
Install Node and the Node Package Manager (npm).....	8
Installing Cypress in Microsoft Windows.....	8
Upgrade Cypress to a newer version .....	8
Running Cypress interactively.....	9
Cypress folder structure.....	13
Cypress configuration files.....	13
Running Cypress from the command line or terminal.....	14
Some example execution options.....	15
Using browsers.....	15
Environment variables .....	15
Option 1: Using environment variables from “cypress.env.json” .....	15
Option 2: Using environment variables from “package.json” .....	16
Configuration Options.....	16
Configure Cypress .....	17
Retries .....	17
Writing Cypress tests .....	18
“.find()”, “.children()”, and “.eq(n)” methods.....	18
“.each()” method .....	19
Get the value of (HTML) properties .....	19
Individual timeout settings .....	19
Debugging commands .....	19
Running specific tests or spec files only.....	20
Skipping blocks (specs) or single tests .....	20
Interacting with Elements.....	21
.click().....	21
.dblclick().....	21
.rightclick() .....	21
.type().....	21
.clear() .....	21
.check().....	21

.uncheck().....	21
.select() for static drop-downs.....	21
.selectFile().....	22
.scrollIntoView ().....	22
.scrollTo ().....	22
.trigger().....	22
Overriding negative checks.....	23
Browser navigation.....	23
Assertions.....	24
Implicit and Explicit Assertions.....	24
Negative Assertions.....	24
Common Assertions.....	24
Length.....	24
Class.....	24
Value.....	24
Text Content.....	24
Visibility.....	24
Existence.....	25
State.....	25
CSS.....	25
Attribute.....	25
Multiple assertions.....	25
Hooks.....	26
Closures and conditional statements.....	27
Aliases.....	28
Table example.....	29
Stubbing, Spying, and controlling date and time (Clocks).....	30
Use “cy.request()” instead of “cy.visit()” for API requests.....	30
Example GET method.....	30
Example POST method.....	30
Replace “cy.request()” with “cy.api()”.....	31
Intercept.....	32
Stubbing with and without using Fixtures.....	33
Spying.....	33
Clocks.....	34
Data-driven tests.....	35

Reading data from a fixture file for data-driven tests .....	36
Reading data from a JSON file.....	36
Reading data from a Comma-Separated Values (CSV) file .....	37
Comma Separated Values (CSV) fixture example .....	38
“Cy.visit()” configuration options.....	39
Multi-domain testing with “cy.origin()” and “cy.session()” .....	40
Using “cy.origin()” and “cy.session()” .....	40
Browser tab handling (Workaround) .....	41
Frames.....	42
XPath support .....	43
Tags .....	44
Installing the “cypress-grep” plugin.....	44
Example use of tags .....	44
Filtering with cypress-grep.....	45
Cypress-grep use examples:.....	45
Behaviour-Driven Development (BDD) .....	46
Parallel test executions .....	48
GitHub Actions parallel execution example.....	48
Jenkins integration .....	49
Docker .....	49
Reporters .....	50
Setup the “cypress-mochawesome-reporter” .....	50
Generate the HTML report using the “cypress-mochawesome-reporter” .....	50
Delete old test results data before creating a new report .....	51
Intelligent Code Completion in Microsoft Visual Studio Code .....	52
Option 1: Add “triple slash directives” to every program code page.....	52
Option 2: Add a configuration file .....	52
Cypress Recorder .....	53
Cypress Scenario Recorder .....	54

## Cypress is not Selenium

Cypress is not just another repackaging of Selenium like so many other JavaScript testing frameworks, but it is a completely new and different approach. It uses very different technology.

Cypress tests run inside of the browser, which you can actually debug! Unlike Selenium, there are no object serialisations or JSON wire protocol communications. You have real, native access to everything in your application under test. This allows for direct access to the browser's development tools (<F12>), including all network recording and inspection options. It also allows for time travel through a recorded session to restore the browser status to a specific time point during the test execution. This makes debugging a lot easier than with Selenium.

Cypress is built around JavaScript and Node and supports them natively. This is unlike Selenium, where JavaScript is just one of many possible language bindings and where using JavaScript does not offer any advantage over any other language binding (such as Java, C#, Python, Ruby etc.). Unlike Selenium, Cypress is built around the concept of asynchronous communication that dominates modern web architectures like Single Page Applications (SPA's), which are often built with JavaScript frameworks like React or Angular. Although there are plenty of JavaScript frameworks for Selenium (like Nightwatch.js, Webdriver.io, Protractor etc.), these frameworks are still just repackaging and trying to work around the synchronous nature of Selenium.

Selenium has been designed in a time where web applications were mostly static with fixed page requests and page responses (stateless web applications). Cypress is about a decade newer than Selenium and designed around asynchronous communication (stateful web applications).

The sweet spot of Cypress is to be used as a tool to test your own JavaScript applications as you build them. It is built for developers and Quality Assurance (QA) engineers. Cypress is particularly suitable for fast visual low-level tests, such as Unit Tests (Cypress calls them Component Tests) and simple Integration Tests (Cypress calls them End-to-End Tests), while Selenium is often more suitable for complex business process based user interface tests, particularly when these tests need to run on a variety of browsers and mobile devices.

Cypress has a much smaller community of developers, tools, and frameworks than Selenium. It will certainly not replace Selenium any time soon, but it can and should be used where Selenium comes to its limits with modern and interactive web applications.

- + Tests execute faster in Cypress than in Selenium.
- + Tests execute more reliably (less flaky) in Cypress than in Selenium.
- + Cypress utilizes a single custom universal driver for all the browsers that it supports.
- + Cypress comes with built-in explicit retries to search for elements in the Document Object Model (DOM) and explicitly waits for events to happen before a test is considered to have failed.
- + Cypress allows for time travel. It takes snapshots as your tests run. In the Test Runner application, you can simply hover over commands in the Command Log to see exactly what happened at, before, and after each step. You can often directly inspect the elements.
- + Cypress allows for easy stubbing of API calls. This can make tests more reliable, and it also allows for tests to be implemented when the API's are not available yet, or cannot be used because of financial constraints, or impacts on production systems.

- + Screenshots are taken automatically on failure and videos of your entire test suite are recorded when run headlessly (in Continuous Integration).
- + Cypress offers a simple Cucumber/Gherkin plugin for Behaviour-Driven Development (BDD).
- Cypress currently only supports Google Chrome-family browsers (including Electron and Chromium-based browsers like Microsoft Edge), and Mozilla Firefox. Other browsers, such as Apple Safari might be supported in the future.
- Cypress does not support testing of native mobile apps. There is no equivalent to what Appium does for Selenium.
- Selenium offers a much better integration with cloud testing providers like Sauce Labs and BrowserStack.
- Cypress runs only in a single browser and only on a single browser tab, although this document provides a workaround for the single browser tab problem.
- Cypress runs in a browser and therefore uses JavaScript only. Selenium supports a wide range of programming languages, such as Java, Python, Ruby, C#, JavaScript, Perl, and PHP.
- In Cypress, tests are limited by browser security to only visit a single superdomain (single origin). However, there are some workarounds for this limitation.

Cypress is free and open source. Tests can be executed using the free Cypress Test Runner application or headlessly, as described in this document. This document only covers the free and open source parts of Cypress.

In addition to that, Cypress also offers an optional cloud-based “Dashboard Service” for recording and storing test results. This service offers a free tier and multiple paid-for tiers with higher numbers of allowed users and test results recordings. The paid “Dashboard Service” helps support the work that the Cypress teams does on the free and open source Cypress Test Runner.

## Cypress Best Practices

Cypress recommends using “data-\*” attributes to provide context to your selectors and insulate them from CSS or JavaScript changes.

1. Don't target elements based on JavaScript framework generated variable CSS attributes.
2. Don't target elements that may change their text content.
3. Add “data-\*” attributes to make it easy to target elements.

Cypress considers the use of “Page Objects” as recommended in Selenium as a no-go (anti-pattern). It recommends testing specs in isolation (without the object sharing of “Page Objects”), programmatically logging into your application (instead of using the user interface), and taking control of your application's state. This does not mean that the user interface login and account creation / reset functionalities should not be tested using the user interface at all, but it means that they should only be tested once, using the user interface in separate login tests. In all other test cases, where the login state is just a precondition to the tests, the desired login state should ideally be created programmatically, for example by using stubbed API calls. This is both faster and safer, and it does not needlessly repeat the user interface login functionality multiple times.

## PageObject example

If you need to use Page Objects to centralise locators for elements, then it is recommended to build a JavaScript Class for each page and to get the locator property values through “get... ()” methods for each locator.

1. Build the Page Object Class and export it:

```
class HomePage
{
  getEditBox()
  {
    return cy.get('#myeditbox')
  }
}
export default HomePage
```

2. To use it in your tests, first import the Page Object Class:

```
import HomePage from '../folderPath/HomePage'
```

3. Second, build an Object to use the Page Object Class:

```
const homePage = new HomePage()
homePage.getEditBox().type('Hello')
```

## Installing Node and Cypress

### Install Node and the Node Package Manager (npm)

Before you can install and use Cypress, you need to install Node.

Please follow an installation guide for your operating system and version.

You can check that Node has been successfully installed with this command:

```
node -v
```

When you installed Node, you also automatically installed the “npm” Command Line Interface (CLI), which is the package manager for Node. You can check the installed version with:

```
npm -v
```

### Installing Cypress in Microsoft Windows

1. Change in the project folder where Cypress should get installed as part of the project:

```
cd /your_project_path
```

2. Initialise the project folder (if the folder is empty):

```
npm init -y
```

This will create a default “package.json” file that will be used to define the project.

3. Install Cypress with:

```
npm install --save-dev cypress
```

### Upgrade Cypress to a newer version

The following example command upgrades Cypress to version 12.11.0:

```
npm install --save-dev cypress@12.11.0
```



## Running Cypress interactively

Cypress has 2 main execution commands:

- "cypress open" opens the GUI interactive mode (Test Runner application).
- "cypress run" executes in headless mode (suitable for Continuous Integration).

1. Add this to the "scripts" settings in the "package.json" file:

```
"scripts": {  
  "cypress:open": "cypress open"  
},
```

2. You can then start the Cypress Test Runner application with:

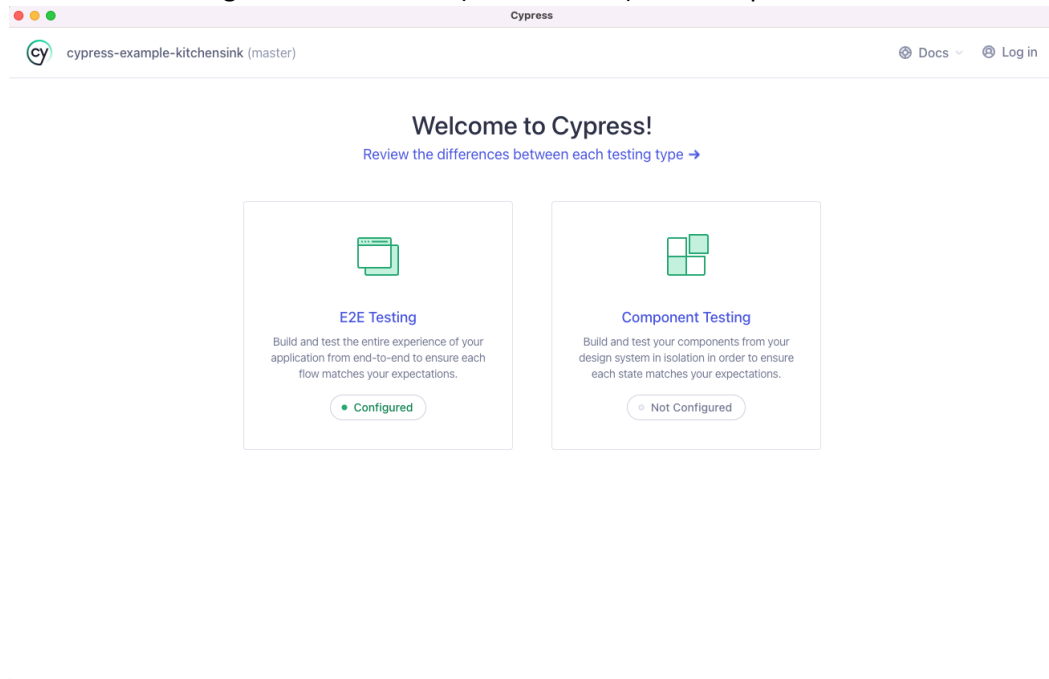
```
npm run cypress:open
```

The following options are available for "cypress open":

Option	Description
--browser, -b	Specify a different browser to run tests in
--config, -c	Specify configuration
--detached, -d	Open Cypress in detached mode
--config-file, -c	Specify a configuration file that will be used to run the tests. The values declared in this file override the ones in the "cypress.config.js" file.
--env, -e	Specify environment variables
--global	Run in global mode
--help, -h	Output usage information
--port, -p	Override default port
--project, -P	Path to a specific project

After running "cypress open" for the first time, Cypress will ask you, if you would like to create E2E (End-to-End) Tests, or Component Tests (visual Unit Tests).

Select "E2E Testing" for conventional (Selenium like) business process tests:



3. Cypress informs you about configuration files that it added to your projects:

## Configuration Files

We added the following files to your project:

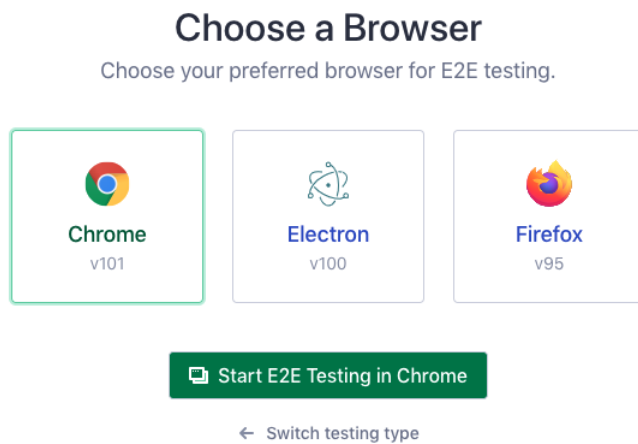
```
✓ cypress.config.js
  The Cypress config file for E2E testing.

1  const { defineConfig } = require("cypress");
2
3  module.exports = defineConfig({
4    e2e: {
5      setupNodeEvents(on, config) {
6        // implement node event listeners here
7      },
8    },
9  });
  Copy

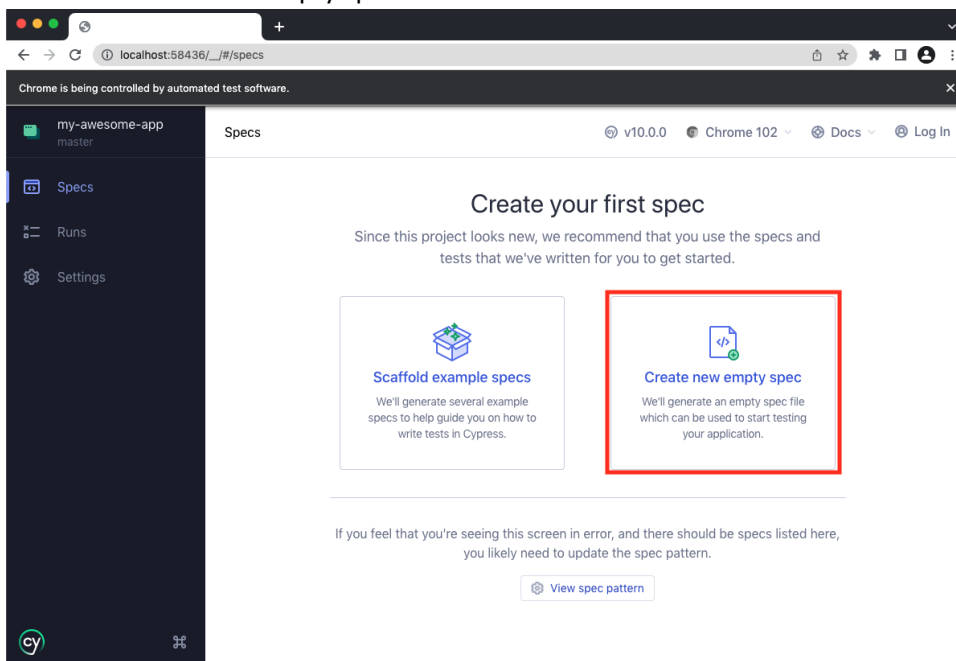
✓ cypress/support/e2e.js
  The support file that is bundled and loaded before each E2E spec.

1  // *****
2  // This example support/e2e.js is processed and
3  // loaded automatically before your test files.
4  //
5  // This is a great place to put global configuration and
6  // behavior that modifies Cypress
```

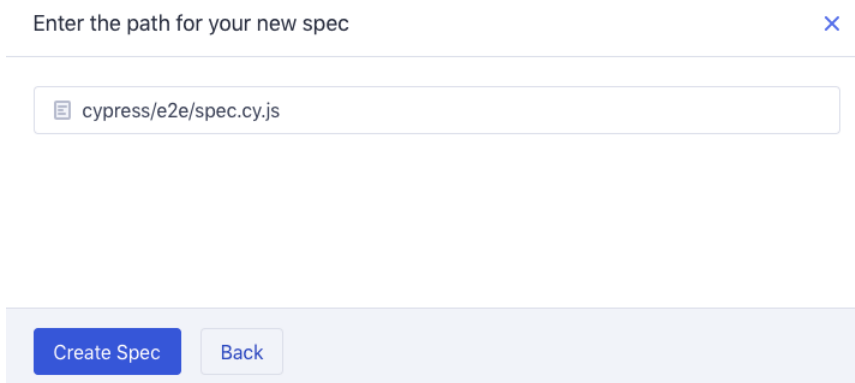
4. Choose one of your locally installed browsers for your E2E Tests:



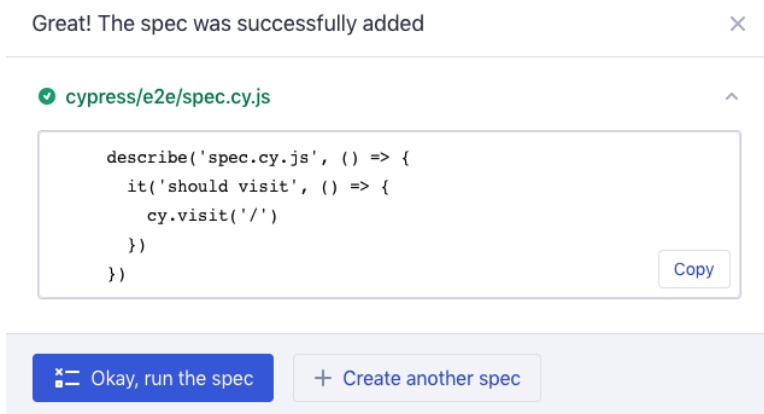
5. Click on "Create new empty spec":



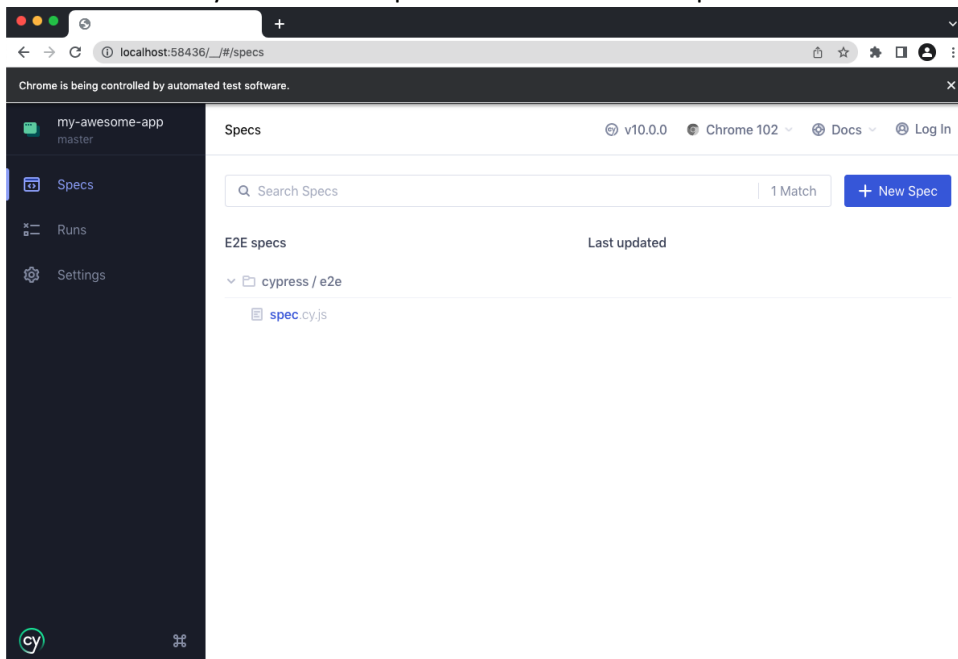
6. Accept or change the default spec file name and click on the "Create Spec" button:



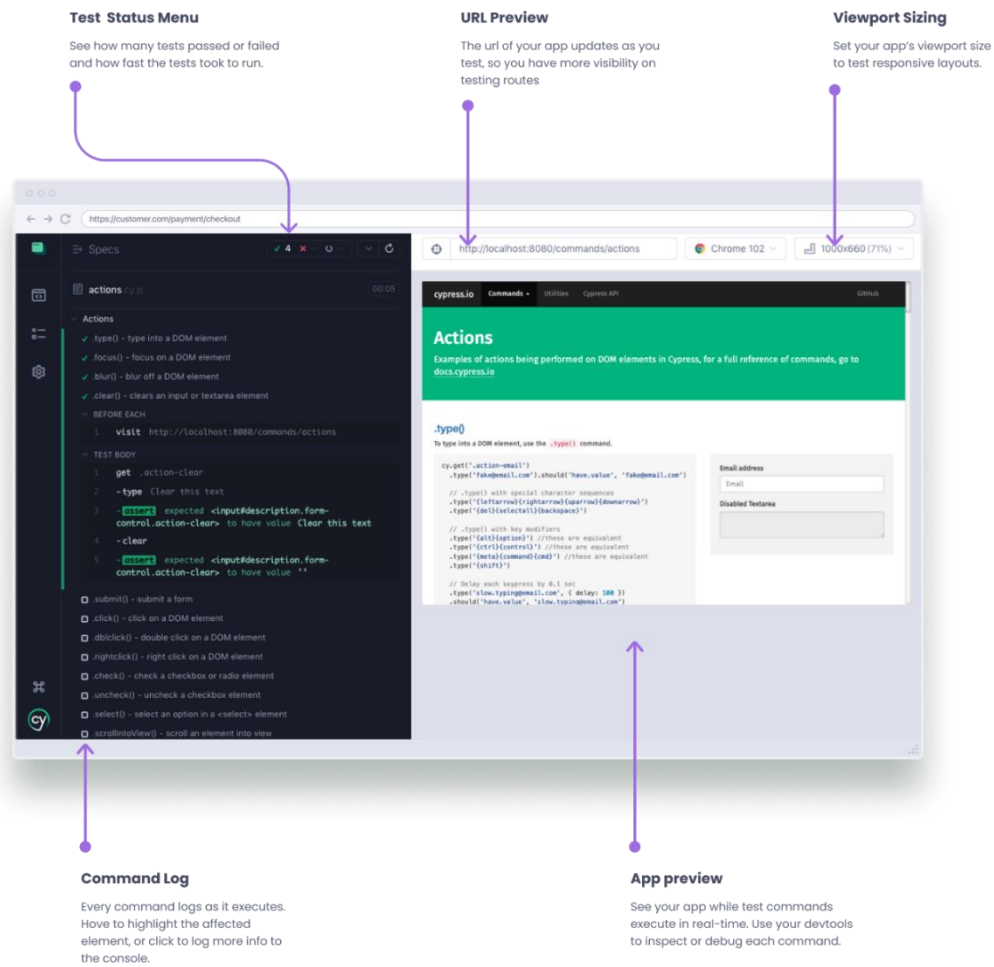
7. Close the newly-generated spec file dialog by clicking on the Window “X”:



8. You will now see your new test spec file in the list of E2E specs:



9. Cypress runs tests in a unique Test Runner application with many interactive features that allow you to see commands as they execute while also viewing the application under test.



## Cypress folder structure

cypress/downloads	Path to folder where files downloaded during a test are saved
cypress/e2e	(E2E) Test cases
cypress/fixtures	Test data in form of key-value pairs for the tests
cypress/support	Reusable methods or customized commands, which can be utilised by test cases directly without object creation. "command.js" can contain re-usable code. "e2e.js" runs before each test file. This can be leveraged for handling pre-requisites for tests.
node_modules	Project dependencies from the Nodes Package Manager (npm)

## Cypress configuration files

cypress.config.js	The values of the current configurations can be modified here, which overrules the default configurations
package.json	Dependencies and scripts for the projects

## Running Cypress from the command line or terminal

1. Add this to the "scripts" section in the "package.json" file:

```
"scripts": {  
  "cypress:run": "cypress run"  
},
```

2. You can now run cypress with:

```
npm run cypress:run
```

If you just want to run a single spec file, then the command is for example:

```
npm run cypress:run -- --spec "cypress/e2e/example.cy.js"
```

If you want to run all spec file in a folder, then the command is for example:

```
npm run cypress:run -- --spec "cypress/e2e/subfolder/**/"
```

Option	Description
--browser, -b	Specify a different browser to run tests in
--ci-build-id	Specify a unique identifier for a run to enable grouping or parallelisation
--config, -c	Specify configuration
--config-file, -c	Specify a configuration file that will be used to run the tests. The values declared in this file override the ones in the "cypress.config.js" file.
--env, -e	Specify environment variables
--group	Group recorded tests together under a single run
--headed	Display the Electron browser instead of running headlessly
--headless	Run tests without launching a browser
--help, -h	Output usage information
--key, -k	Specify your secret record key
--no-exit	Keep Cypress Test Runner open after tests in a spec file run
--parallel	Run recorded specs in parallel across multiple machines
--port, -p	Override default port
--project, -P	Path to a specific project
--record	Whether to record the test run
--reporter, -r	Specify a Mocha reporter
--reporter-options, -o	Specify Mocha reporter options
--spec, -s	Specify the spec files to run

## Some example execution options

### Using browsers

Cypress currently supports Google Chrome in the varieties Chrome, Chrome Beta, Chrome Canary, Chromium, Edge, Edge Beta, Edge Canary, Edge Dev, and Electron. Mozilla Firefox is currently supported in the varieties Firefox, Firefox Developer Edition, and Firefox Nightly. WebKit has experimental support only and therefore requires opt-in. Cypress does recognise installed browser versions automatically.

The Electron browser is a version of Chromium that comes with Electron. It comes baked into Cypress and does not need to be installed separately. By default, when running `cypress run` from the command line or terminal, it will launch Electron headlessly. Because Electron is the default browser, it is typically run in Continuous Integration. If you are seeing failures in Continuous Integration, you might want to debug the tests by running them locally with the `--headed` option.

A specific browser can be launched with:

```
cypress run -- --browser chrome
```

A browser can also be launched by specifying a path to the binary:

```
cypress run -- --browser /usr/bin/chromium
```

Note: Cypress generates its own isolated profile apart from your normal browser profile.

### Environment variables

Cypress environment variables are global variables that can be used across all tests.

Cypress environment variables are dynamic name-value pairs that influence the way Cypress executes tests. These environment variables are useful when there is a need to run the tests in multiple environments, or when the defined values are prone to quickly changing.

In Cypress, you can define single or multiple environment variables either as strings or JSON objects.

In Cypress, there are many different options to use environment variables.

#### Option 1: Using environment variables from `cypress.env.json`

For this option, you need to create a new file called `cypress.env.json`. Here is some example content:

```
{  
  "host": "veronica.dev.local",  
  "api_server": "http://localhost:8888/api/v1/"  
}
```

It can then be used like this:

```
Cypress.env() // {host: 'veronica.dev.local', api_server: 'http://localhost:8888/api/v1'}
Cypress.env('host') // 'veronica.dev.local'
Cypress.env('api_server') // 'http://localhost:8888/api/v1/'
```

Option 2: Using environment variables from “package.json”

You can also set environment variables in “package.json”.

This example defines an environmental variable called “TransferProtocol”:

```
"scripts": {
  "cypress:run": "cypress run --env TransferProtocol='http'",
  "cypress:run:v2": "cypress run --env TransferProtocol='https'"
},
```

It can then be used in tests like this:

```
cy.visit(`${Cypress.env("TransferProtocol")}://www.example.com`)
```

Scripts can then be executed with either

```
npm run cypress:run
```

or

```
npm run cypress:run:v2
```

## Configuration Options

Cypress can set and override configurations using commands running on the terminal.

This example overrides the viewport settings:

```
"scripts": {
  "cypress:tablet-view": "cypress run --config viewportHeight=763,viewportWidth=700"
},
```

It can be run with:

```
npm run cypress:tablet-view
```



## Configure Cypress

Configuration options are set in the file `“cypress.config.js”`.

This example `“baseUrl”` automatically prefixes `“cy.visit()”` and `“cy.request()”` commands:

```
{
  baseUrl: "https://example.cypress.io"
}
```

With this setting, it is then possible to call the home page just with `“cy.visit('/')”`, instead of `“cy.visit('https://example.cypress.io')”`.

Cypress has many more configuration options that you can use to customise its behaviour. These include where your tests live, default timeout periods, environment variables, which reporter(s) to use, and many more.

The default timeout periods are:

Option	Default	Description
<code>defaultCommandTimeout</code>	4000	Time, in milliseconds, to wait until most DOM based commands are considered timed out
<code>execTimeout</code>	60000	Time, in milliseconds, to wait for a system command to finish executing during a <code>“cy.exec()”</code> command
<code>taskTimeout</code>	60000	Time, in milliseconds, to wait for a task to finish executing during a <code>“cy.task()”</code> command
<code>pageLoadTimeout</code>	60000	Time, in milliseconds, to wait for page transition events or <code>“cy.visit()”</code> , <code>“cy.go()”</code> , <code>“cy.reload()”</code> commands to fire their page load events. Network requests are limited by the underlying operating system, and may still time out if this value is increased.
<code>requestTimeout</code>	5000	Time, in milliseconds, to wait for an XHR request to go out in a <code>“cy.wait()”</code> command
<code>responseTimeout</code>	30000	Time, in milliseconds, to wait until a response in a <code>“cy.request()”</code> , <code>“cy.wait()”</code> , <code>“cy.fixture()”</code> , <code>“cy.getCookie()”</code> , <code>“cy.getCookies()”</code> , <code>“cy.setCookie()”</code> , <code>“cy.clearCookie()”</code> , <code>“cy.clearCookies()”</code> , and <code>“cy.screenshot()”</code> commands

## Retries

A powerful configuration, particularly for unstable tests, is retries. The retries option retries the test, if it fails. At times tests fail because of network or environmental issues. In these circumstances, retrying tests is very important, as there might not be any problem with the actual tests themselves. The retries option is specified in the `“cypress.config.js”` file. The following example retries a failed tests 3 more times when executed in run mode, and 2 more times when executed in open mode:

```
retries: {
  runMode: 3,
  openMode: 2
}
```

## Writing Cypress tests

Cypress End-to-End test spec files should use the “\*.cy.js” file extension and be stored in the “cypress/e2e” folder of the project. This ensures that the test spec files will automatically be picked up by the Test Runner application.

This example test uses the example application named “Kitchen Sink” from the “cypress.io” web site. It does the following:

1. Calls the “Kitchen Sink” start page <https://example.cypress.io> .
2. Clicks on an element that contains the text “type”.
3. Verifies (=asserts) that the resulting URL includes “/commands/actions”.
4. Types “fake@email.com” into the element “.action-email” and verifies (=asserts) that the text is correctly entered into the element.

```
describe('My First Test', () => {
  it('Gets, types and asserts', () => {
    cy.visit('https://example.cypress.io')
    cy.contains('type').click()
    // Should be on a new URL which includes '/commands/actions'
    cy.url().should('include', '/commands/actions')
    // Get an input, type into it and verify that the value has been updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
  })
})
```

### “.find()”, “.children()”, and “.eq(n)” methods

In Cypress, elements are queried using the JQuery syntax. This also allows JQuery DOM traversals.

The following example demonstrates parent-child relationships. It looks for “'#main-content'”, then inside of this element for “.article”, and then for the first element of “'img[src^="/static"]'”. The “.children()” method differs from “.find()” in that “.children()” only travels a single level down the DOM tree while “.find()” can traverse down multiple levels to select descendant elements (grandchildren etc.) as well.

If instead of the first element of an array in the following example, the  $n$ -th element should be selected, then the “.eq( $n$ )” method can be used, for example “.eq(3)” to select the fourth element (as arrays are zero-based).

```
// Each method is equivalent to its jQuery counterpart.
cy.get('#main-content')
  .find('.article')
  .children('img[src^="/static"]')
  .first()
```

## “.each()” method

The “.each()” method allows iterating through an array like structure (arrays or objects with a length property). However, it is unsafe to chain further commands that rely on the subject after “.each()”, which is why wrapping is required before clicking on it, as in this example:

```
cy.get('ul>li').each(($el, index, $list) => {
  // $el is a wrapped jQuery element
  if ($el.someMethod() === 'something') {
    // wrap this element so we can
    // use cypress commands on it
    cy.wrap($el).click()
  } else {
    // do something else
  }
})
```

## Get the value of (HTML) properties

Cypress cannot get the value of (HTML) properties directly, but the jQuery “prop()” function can be used for this purpose.

The following extracts and logs the value of the HTML “href” property of “'#myElement'”

```
cy.get('#myElement').then(function(el)
{
  const myUrl=$el.prop('href')
  cy.log(myUrl)
})
```

## Individual timeout settings

Most commands allow individual timeout settings (the Cypress default timeout is 4 seconds):

```
// Give this element 10 seconds to appear
cy.get('.my-slow-selector', { timeout: 10000 })
```

## Debugging commands

<code>cy.pause()</code>	Stop “cy” commands from running and allow interaction with the application under test. You can then “resume” running all commands or choose to step through the “next” commands from the Command Log. Unlike the “.debug” command, “cy.pause()” does not have to be chained to other commands and can be used independently.
<code>.debug()</code>	Set a debugger and log what the previous command yields. You need to have your Developer Tools (<F12>) open for “.debug()” to hit the breakpoint. Example use for logging out the current subject for debugging: <pre>cy.get('.ls-btn').click({ force: true }).debug()</pre>

### Running specific tests or spec files only

By default, Cypress executes all tests and specs files. If only a single test (or spec) should be executed, then this can be achieved by adding “.only” to the test (or spec), for example:

```
it.only('Gets, types and asserts', () => {  
  ...  
})
```

This is particularly useful during test development when a new test case gets added to the existing test cases.

### Skipping blocks (specs) or single tests

Tests that are in a whole block (spec) can be excluded from execution by skipping the whole test block (spec) with “describe.skip(...)”.

Single tests can be excluded from execution by skipping the test with “it.skip(...)”.

## Interacting with Elements

These action commands can interact with elements:

### `.click()`

```
cy.get('button').click() // Click on button (in the centre of the button)
cy.get('button').click('topLeft') // Click on button (in the top left corner of the button)
cy.focused().click() // Click on el with focus
cy.contains('Welcome').click() // Click on first el containing 'Welcome'
```

An element will be clicked in the centre by default, .but an element can also be clicked “topLeft”, “top”, “topRight”, “left”, “right”, “bottomLeft”, “bottom”, and “bottomRight”.

### `.dblclick()`

```
cy.get('button').dblclick() // Double click on button
cy.focused().dblclick() // Double click on el with focus
cy.contains('Welcome').dblclick() // Double click on first el containing 'Welcome'
```

### `.rightclick()`

```
cy.get('button').rightclick() // Right-click on button
cy.focused().rightclick() // Right-click on el with focus
cy.contains('Welcome').rightclick() // Right-click on first el containing 'Welcome'
```

### `.type()`

```
cy.get('input').type('Hello, World') // Type 'Hello, World' into the 'input'
cy.get('input').type('Hello, World',{force: true}) // Type 'Hello, World' into disabled 'input'
```

### `.clear()`

```
cy.get('[type="text"]').clear() // Clear text input
cy.get('textarea').type('Hi!').clear() // Clear textarea
cy.focused().clear() // Clear focused input/textarea
```

### `.check()`

```
cy.get('[type="checkbox"]').check() // Check checkbox element
cy.get('[type="radio"]').first().check() // Check first radio element
cy.get('[type="radio"]').should('be.checked') // Verification
```

### `.uncheck()`

```
cy.get('[type="checkbox"]').uncheck() // Unchecks checkbox element
cy.get('[type="radio"]').should('not.be.checked') // Verification
```

### `.select()` for static drop-downs

```
cy.get('select').select('user-1') // Select the 'user-1' option
cy.get('select').select(['user-1', 'user-2']) // Select both 'user-1' and 'user-2'
```

## .selectFile()

```
cy.get('input[type=file]').selectFile('path/to/file.json') // Attach the file from disk
```

## .scrollIntoView ()

```
cy.get('button#checkout').scrollIntoView().should('be.visible') // Scroll an element into view
```

## .scrollTo ()

```
cy.scrollTo('bottom') // Scroll to the bottom of the window
```

Valid positions are “topLeft”, “top”, “topRight”, “left”, “center”, “right”, “bottomLeft”, “bottom”, and “bottomRight”.

## .trigger()

```
cy.get('a').trigger('mousedown') // Trigger mousedown event on link
```

The trigger command in Cypress helps trigger an event on the DOM element. X, Y positions can also be supplied to the trigger command.

This command is useful for all mouse events, including drag and drop operations. Mouse events include “mousedown”, “mouseup”, “mouseover”, and “mousemove”.

## Overriding negative checks

Cypress does a lot of internal checks and adjustments before action commands get executed. Sometimes, a negative result from these checks will prevent the execution. In these cases, it is possible to override the negative results by using “`{force: true}`”, for example:

```
cy.get('button').click({force: true})
```

Using “`{force: true}`” is often required for actions on inaccessible elements.

## Browser navigation

Going back to the previous page is possible by passing “`-1`” or “`'back'`” to the “`go`” method, for example:

```
cy.go(-1)  
cy.go('back')
```

Similarly, going forward to the next page is accomplished by passing “`1`” or “`'forward'`” to the “`go`” method:

```
cy.go(1)  
cy.go('forward')
```

Cypress also provides a method called “`reload`” for refreshing or reloading the page:

```
cy.reload()
```

# Assertions

## Implicit and Explicit Assertions

An Implicit Assertion is an assertion that applies to the object provided by the parent chained command. This category of assertions generally includes commands such as “.should()” and “.and()”. As these commands don’t stand independently and always depend on the previously chained parent command, they automatically inherit and act on the object yielded by the previous command.

When there is a need to pass an explicit object for the assertion, it falls under the category of Explicit Assertion. This category of assertions contains commands such as “expect()” and “assert()”.

## Negative Assertions

By adding “.should('not.exist')” to any DOM command, Cypress will reverse its default assertion and automatically wait until the element does not exist.

## Common Assertions

### Length

```
// retry until finding 3 matching <li.selected>  
cy.get('li.selected').should('have.length', 3)
```

### Class

```
// retry until this input does not have class disabled  
cy.get('form').find('input').should('not.have.class', 'disabled')
```

### Value

```
// retry until this textarea has the correct value  
cy.get('textarea').should('have.value', 'foo bar baz')
```

### Text Content

```
// partial text match for non-input HTML element  
cy.get('#text-example').should('contain', 'welcome to')
```

```
// retry until this span does not contain 'click me'  
cy.get('a').parent('span.help').should('not.contain', 'click me')
```

### Visibility

```
// retry until this button is visible  
cy.get('button').should('be.visible')
```



## Existence

```
// retry until loading spinner no longer exists  
cy.get('#loading').should('not.exist')
```

## State

```
// retry until our radio is checked  
cy.get(':radio').should('be.checked')
```

## CSS

```
// checks if the web element has a certain CSS property  
cy.get('#txt-flid').should('have.css', 'display', 'block')
```

## Attribute

```
// checks if the web element has an attribute "minlength" with a value of "2"  
cy.get('#txt-flid').should('have.attr', 'minlength', '2')
```

## Multiple assertions

Multiple assertions can be chained by using multiple “.should()”. Alternatively, a second (or more) assertion can also be written as “.and()”, which is exactly the same as “.should()”, but makes tests better human readable.

## Hooks

Cypress provides hooks (borrowed from the underlying Mocha framework).

Hooks are helpful to set conditions that you want to run before a set of tests or before each test.

They're also helpful to clean up conditions after a set of tests or after each test.

```
before(() => {  
  // runs once before all tests in the block  
})  
  
after(() => {  
  // runs once after all tests in the block  
})  
  
beforeEach(() => {  
  // runs before each test in the block  
})  
  
afterEach(() => {  
  // runs after each test in the block  
})
```

## Closures and conditional statements

Cypress handles Promises automatically when chaining Cypress commands and assertions. This means that it chains commands with an automatically generated internal logic that is similar to using “.then()”, but without the need to handle the Promise manually with a function.

If you chain Cypress commands and something else, for example a jQuery function like “text()” or a JavaScript command, then you need to handle the Promise of the previous Cypress command manually by using “.then()”.

To access what each Cypress command yields use “.then()”:

```
cy.get('button').then(($btn) => {  
  // $btn is the object that the previous command yielded  
})
```

This is a full example of using “.then”:

```
cy.get('button').then(($btn) => {  
  // store the button's text  
  const txt = $btn.text()  
  // submit a form  
  cy.get('form').submit()  
  // compare the two buttons' text  
  // and make sure they are different  
  cy.get('button').should(($btn2) => {  
    expect($btn2.text()).not.to.eq(txt)  
  })  
})
```

Closures can also be used for conditional testing, for example:

```
// this only works if there's 100% guarantee  
// body has fully rendered without any pending changes  
// to its state  
cy.get('body').then(($body) => {  
  // synchronously ask for the body's text  
  // and do something based on whether it includes  
  // another string  
  if ($body.text().includes('some string')) {  
    // yup found it  
    cy.get(...).should(...)  
  } else {  
    // nope not here  
    cy.get(...).should(...)  
  }  
})
```

## Aliases

Aliases are a way to prevent the usage of “.then ()” functions in tests.

Aliases are often used to share objects between hooks and tests. Another great use of aliasing for sharing contexts is with Cypress fixtures.

Aliases allow sharing and reusing of objects. To alias something you’d like to share, you use the “.as ()” command.

Cypress aliases can be accessed in two ways:

1. The “this” keyword before the alias is used for (direct) synchronous access.
2. Using the “@” character before the alias is used for asynchronous access (execute and continue without waiting for the response).

```
beforeEach(() => {
  // alias the $btn.text() as 'text'
  cy.get('button').invoke('text').as('text')
})

it('has access to text', () => {
  this.text // is now available
})
```

Aliases have other special characteristics when being used with DOM elements. After you alias DOM elements, you can then later access them for reuse.

```
// alias all of the tr's found in the table as 'rows'
cy.get('table').find('tr').as('rows')
```

Internally, Cypress has made a reference to the “<tr>” collection returned as the alias “rows”. To reference these same “rows” later, you can use the “cy.get ()” command.

```
// Cypress returns the reference to the <tr>'s
// which allows us to continue to chain commands
// finding the 1st row.
cy.get('@rows').first().click()
```

Because of using the “@” character in “cy.get ()”, instead of querying the DOM for elements, “cy.get ()” looks for an existing alias called “rows” and returns the reference (if it finds it).

## Table example

The following example uses this example Table:

Country	AirportName	AirportCode
USA	Los Angeles	LAX
Germany	Frankfurt	FRA
Australia	Sydney	SYD

It loops through the values of the second column (AirportName) and checks that if the text contains “Germany”, then the next (third) column (AirportCode) should have text containing “FRA”.

```
cy.get('tr td:nth-child(2)').each(($el, index, $list) => {
  const airportNameText=$el.text()
  if(airportNameText.includes('Germany'))
  {
    cy.get('tr td:nth-child(2) ').eq(index).next().then(function(airportCode)
    {
      const airportCodeText=airportCode.text()
      expect(airportCodeText).to.equal('FRA')
    })
  }
})
```

## Stubbing, Spying, and controlling date and time (Clocks)

Most web applications make background calls to (API) services. Cypress makes it easy to control the responses of these background service calls by replacing or simulating the responses of these service calls, even if the API's are not built or available yet (mocking).

Cypress makes it easy to stub a server response and control the body, status, headers, or even delay of the server response.

### Use “`cy.request()`” instead of “`cy.visit()`” for API requests

The “`cy.request()`” command is responsible for making HTTP requests to (API) endpoints. This command can be used to execute API requests and receive responses without the need to create or import an external library to make and handle API requests and responses.

Because “`cy.request()`” does not execute inside of the browser, it is not subject to the cross-domain restrictions of the browser. It can therefore also call other superdomains.

#### Example GET method

```
cy.request({method: 'GET',
  url: 'https://jsonplaceholder.cypress.io/comments'}).should((response) => {
  // the server sometimes gets an extra comment posted from another machine
  // which gets returned as 1 extra object
  expect(response.body)
    .to.have.property('length')
    .and.be.oneOf([500, 501])
  expect(response.status).to.eq(200)
  expect(response).to.have.property('headers')
  expect(response).to.have.property('duration')
})
```

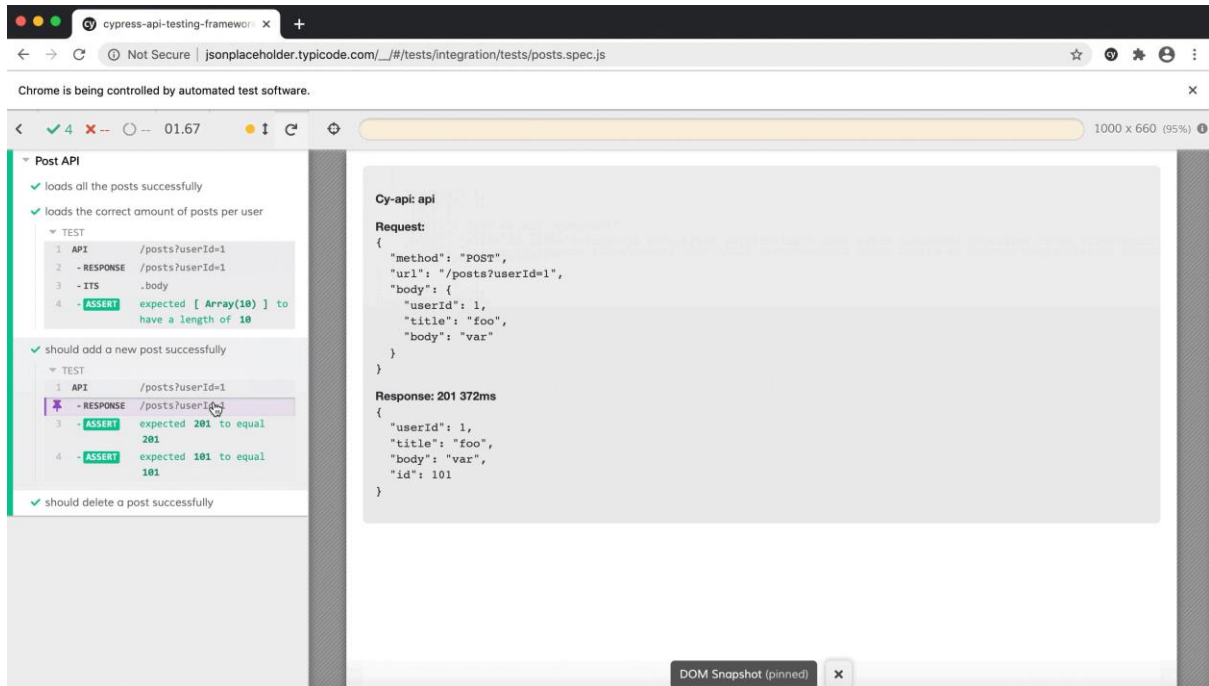
#### Example POST method

```
cy.request({method: 'POST',
  url: 'https://jsonplaceholder.typicode.com/posts',
  body: {title: 'foo', body: 'bar', userId: 1}}).should((response) => {
  expect(response.status).to.eq(201)
})
```

## Replace “cy.request()” with “cy.api()”

With the “@bahmutov/cy-api” Cypress plugin, you can replace the “cy.request()” command with the “cy.api()” command. Both commands work exactly the same way. They both use the same parameters.

When using the Cypress Test Runner, “cy.api()” shows results that are more suitable for API’s, including the request, the response, and the response time:



To install the “@bahmutov/cy-api” Cypress plugin:

```
npm install --save-dev @bahmutov/cy-api
```

Add the following line to your Cypress support file (usually “cypress/support/index.js”):

```
import '@bahmutov/cy-api'
```

## Intercept

“`cy.intercept()`” offers flexibility and granular control over handling of the network layer. It has out-of-the-box support for intercepting fetch calls, page loads, and resource loads in addition to the pre-existing support for XMLHttpRequests (XHR).

“`cy.intercept()`” can handle “GET”, “POST”, “PUT”, “PATCH”, and “DELETE” methods.

“`cy.intercept()`” must be called before it can be used in tests, so that the routes are recorded before they are called. It is therefore often part of a “`before()`” hook.

The following is an example that shows the “`cy.intercept()`” command listening for an XHR response that it expects Cypress to make on initialization of the application. It waits for the route response to have been called before it completes execution.

The “`cy.wait()`” command enables waiting for the (stubbed or genuine) response before proceeding with the next Cypress command. This can make tests more robust. Another benefit of using “`cy.wait()`” on intercepted requests is that it allows you to access the actual XHR object. This is useful when you want to make assertions about this object. You can check the URL, Method, Status Code, Request Body, Request Headers, Response Body, and Response Headers of the XHR object.

Due to the asynchronous nature of JavaScript, it is strongly recommended to use “.`as()`” to catch the Promise of “`cy.intercept()`”, then to trigger the interception, for example by using “`cy.visit()`” as in the following example, and then to use “`cy.wait()`” to wait for the interception to complete before continuing:

```
describe('Routing a request', () => {
  it('can wait for a app initialization', () => {
    cy.intercept('POST', '**/j/**').as('initializeTodoApp')
    cy.visit('https://todomvc.com/examples/react/#/')
    cy.wait('@initializeTodoApp') // wait for intercept response
  })
})
```

“`cy.wait()`” returns two properties, “request” and “response”. These properties can be used to verify the correct execution, for example:

```
cy.intercept('GET', '**/comments/*').as('getComment')
cy.get('form').submit()
cy.wait('@getComment').its('response.statusCode').should('be.oneOf', [200, 304])
```

“`cy.intercept()`” provides the ability to override XHR responses returned by the requests made by Cypress tests during execution. Overriding the XHR responses is called “stubbing”.



## Stubbing with and without using Fixtures

A fixture is a fixed set of data usually located in a file (often of type “\*.json”) that is used in your tests. The purpose of a test fixture is to ensure that there is a well-known fixed environment in which tests are run, so that results are repeatable. Fixtures are accessed within tests by calling the “cy.fixture()” command.

The following example uses a fixture file called “activities.json” (that is stored in the default Cypress fixture folder “/cypress/fixtures”) to automatically stub responses for “GET” requests that include the substring “/myapi” (for example to “https://example.com/myapi?\_limit=3”):

```
cy.intercept('GET', '/myapi', {fixture:'activities.json'})
```

It is also possible to set the stubbed response directly without using a Fixture file:

```
cy.intercept('GET', '/myapi', {statusCode:200,body:'All OK!'})
```

The official Cypress documentation has many more examples and explains the many possible options for both the requests and the responses.

## Spying

Spying is also very easy with Cypress. It is basically the same as stubbing, but without the third argument for the payload data.

A spy gives you the ability to “spy” on a function, by letting you capture and then assert that the function was called with the right arguments, or that the function was called a certain number of times, or what the return value was, or what context the function was called with.

Spies are only used for verification of working elements or methods in Cypress. Spies do not modify the behaviour of functions, they leave them perfectly intact.

Spying example:

```
const obj = {
  sum(a, b) {
    return a + b
  }
}
const spyRequest = cy.spy(obj, 'sum')
obj.sum(1, 2) // trigger the spy
expect(spyRequest).to.be.called
expect(spyRequest.returnValues[0]).to.eq(3)
```

## Clocks

There are situations when it is useful to control your application's date and time in order to override its behaviour or avoid slow tests.

With `cy.clock()` you can control:

- "Date"
- "setTimeout"
- "setInterval"

## Data-driven tests

The easiest way for data-driven tests in Cypress is by looping through an array using “.forEach()”, as in this example:

```
const names = ['foo', 'bar', 'baz']
names.forEach(name => {
  it('works for ${name}', () => {
    cy.visit('/')
    cy.contains(name)
  })
})
```

This example runs and reports 3 test executions (one execution for each array member).

It is of course also possible to run and report just one test execution, but with 3 different assertions inside the test by declaring and using the array inside the test.

Using “.wrap()” for iterations is often helpful, as in this example:

```
const items = [
  {text: 'Buy milk', expectedLength: 1},
  {text: 'Buy eggs', expectedLength: 2},
  {text: 'Buy bread', expectedLength: 3}
]

cy.wrap(items)
  .each(todo => {
    cy.get('.my-input')
      .type(todo.text)
      .type('{enter}')
    cy.get('.my-list li')
      .should('have.length', todo.expectedLength)
  })
```

## Reading data from a fixture file for data-driven tests

### Reading data from a JSON file

Example fixture file “/cypress/fixtures/testdata.json”:

```
{
  "data": [
    {
      "city": "Tokyo",
      "country": "Japan"
    },
    {
      "city": "Shanghai",
      "country": "China"
    }
  ]
}
```

*If fixture data is required in a single test case only*

Example test that iterates through the data from a JSON fixture file within a single test case:

```
it('Read and log data from a JSON fixture file', () => {
  cy.fixture('testdata.json').then(alldata => {
    alldata.data.forEach(data => {
      cy.log(data.city)
      cy.log(data.country)
    })
  })
})
```

*If fixture data is required in multiple test cases in the same block*

If the fixture data is required in multiple test cases, then the data should be read in a “before ()” hook. Note that this requires for the data to be addressed with “.this”.

```
before(function() {
  // runs once before all tests in the block
  cy.fixture('testdata.json').then(function(data)
  {
    this.data = data
  })
})
```

In the test cases, the data needs to be accessed using “.this”

```
cy.log(this.data.city)
```

## Reading data from a Comma-Separated Values (CSV) file

Test data is often held and maintained in CSV files, which are easy to use for business users through Microsoft Excel.

CSV files can easily be converted to JSON files through Node parser packages, such as Papa Parse: <https://www.papaparse.com/>. Please check the excellent Papa Parse documentation for parameters and much more.

To work with Papa Parse, you have to install it with:

```
npm install --save-dev papaparse
```

For parsing from CSV to JSON, you have to add:

```
import {parse} from "papaparse"
```

Example CSV file `"/cypress/fixtures/csv_example.csv"`:

```
city,country  
"Tokyo","Japan"  
"Shanghai","China"
```

## Comma Separated Values (CSV) fixture example

Example test that converts a CSV fixture file to a JSON file fixture file and iterates through the JSON data of the newly created JSON fixture file:

```
/// <reference types="cypress" />

import {parse} from "papaparse"

describe('Convert CSV to JSON (fixture) with Papa Parse', function () {

  let allData // "allData" is an object that contains an array called "allData.data"

  before(() => {
    // convert CSV fixture to JSON fixture
    cy.readFile('./cypress/fixtures/csv_example.csv').then(str => {
      cy.writeFile('./cypress/fixtures/csv_example.json', parse(str, {header:true}))
    })
    // read JSON fixture data
    cy.fixture('csv_example.json').as('dataJson').then(dataJson => {
      allData = dataJson
    })
  })

  it('Convert, read, and log CSV data', function () {
    // use JSON fixture data
    allData.data.forEach(data => {
      cy.log(data.city)
      cy.log(data.country)
    })
  })
})
```

## “Cy.visit()” configuration options

Option	Default	Description
<code>url</code>	<code>null</code>	The URL to visit. Behaves the same as the <code>url</code> argument.
<code>Method</code>	<code>GET</code>	The HTTP method to use in the visit. Can be <code>GET</code> or <code>POST</code> .
<code>Body</code>	<code>null</code>	An optional body to send along with a <code>POST</code> request. If it is a string, it will be passed along unmodified. If it is an object, it will be URL encoded to a string and sent with a <code>Content-Type: application/x-www-urlencod</code> header.
<code>Headers</code>	<code>{}</code>	An object that maps HTTP header names to values to be sent along with the request. <i>Note:</i> <code>headers</code> will only be sent for the initial <code>cy.visit()</code> request, not for any subsequent requests.
<code>Qs</code>	<code>null</code>	Query parameters to append to the <code>url</code> of the request
<code>Log</code>	<code>true</code>	Displays the command in the <a href="#">Command log</a>
<code>Auth</code>	<code>null</code>	Adds Basic Authorization headers
<code>failOnStatusCode</code>	<code>True</code>	Whether to fail on response codes other than <code>2xx</code> and <code>3xx</code>
<code>onBeforeLoad</code>	<code>function</code>	Called before your page has loaded all of its resources.
<code>onLoad</code>	<code>function</code>	Called once your page has fired its load event.
<code>retryOnStatusCodeFailure</code>	<code>False</code>	Whether Cypress should automatically retry status code errors under the hood. Cypress will retry a request up to 4 times if this is set to true.
<code>retryOnNetworkFailure</code>	<code>True</code>	Whether Cypress should automatically retry transient network errors under the hood. Cypress will retry a request up to 4 times if this is set to true.
<code>timeout</code>	<code>pageLoadTimeout</code>	Time to wait for <code>cy.visit()</code> to resolve before <a href="#">timing out</a>

## Multi-domain testing with “cy.origin()” and “cy.session()”

The “cy.origin()” and “cy.session()” commands allow for easy switching between origins to seamlessly test syndicated authentication, cross-site CMS workflows, and much more.

### Using “cy.origin()” and “cy.session()”

In normal use, a single Cypress test may only run commands in a single origin. This is a limitation determined by standard web security features of the browser. The “cy.origin()” command allows tests to bypass this limitation inside a “cy.origin()” command.

The “cy.session()” command caches and restores cookies, localStorage and sessionStorage after a successful login. The steps that the login code takes inside the “cy.origin()” command to create the session will only be performed once when it's called the first time in any given spec file. Subsequent calls will restore the session from cache.

The following example wraps “cy.origin()” with “cy.session()”:

```
Cypress.Commands.add('login', (username, password) => {
  const args = { username, password }
  // Username & password can be used as the cache key too
  cy.session(args, () => {
    cy.origin('my-auth.com', { args }, ({ username, password }) => {
      cy.visit('/login')
      cy.contains('Username').find('input').type(username)
      cy.contains('Password').find('input').type(password)
      cy.get('button').contains('Login').click()
    })
    cy.url().should('contain', '/home')
  },
  {
    validate() {
      cy.request('/api/user').its('status').should('eq', 200)
    },
  })
})
```



## Browser tab handling (Workaround)

Cypress does not have a specific command to work with browser tabs. However, there is a workaround method in jQuery through which Cypress can handle browser tabs.

Cypress can use the jQuery method “removeAttr” to remove the attribute that asks for a new tab. It deletes the attribute that is passed as one of the parameters to the “invoke” method. Once the HTML “target=\_blank” is removed, then the target opens in the parent window. Later on after performing the operations on it, you can shift back to the parent URL with the “cy.go('back')” command.

```
/// <reference types="cypress" />

describe('Browser tab handling example', function () {
  // test case
  it('New tab opened up through a link with a HTML target="_blank" attribute', function () {
    // url launch
    cy.visit("https://the-internet.herokuapp.com/windows")
    // delete target attribute created by the link
    cy.get('.example > a').invoke('removeAttr', 'target').click()
    // verify tab title
    cy.title().should('eq', 'New Window')
    // shift back to parent window
    cy.go('back')
  })
})
```

## Frames

To work with (i)Frames, you have to install a Cypress plugin:

```
npm install --save-dev cypress-iframe
```

For the frame implementation in Cypress, you have to add this statement:

```
import 'cypress-iframe'
```

The method `frameLoaded()` is used to move the focus from the main page to the frame. Once the focus is shifted, you can interact with the elements inside the frame. This is done with the `iframe()` method.

If you would like to use IntelliSense in Microsoft Visual Studio Code, then you should also add `<reference types="cypress-iframe" />` to the program code:

```
/// <reference types="cypress" />
/// <reference types="cypress-iframe" />

import 'cypress-iframe'

describe('Frame handling example', function () {
  // test case
  it('Switch to iFrame', function () {
    // launch URL
    cy.visit("https://jqueryui.com/draggable/")
    // frame loading
    cy.frameLoaded('.demo-frame')
    // shifting focus
    cy.iframe().find("#draggable").then(function(t){
      const frmtxt = t.text()
      // assertion to verify text
      expect(frmtxt).to.contains('Drag me around')
      cy.log(frmtxt)
    })
  })
})
```

## XPath support

Cypress supports CSS selectors by default, but XPath support can be added.

For XPath support in Cypress, you have to install a plugin:

```
npm install --save-dev cypress-xpath
```

Once the installation is done, you have to add the statement

```
require('cypress-xpath')
```

to the “cypress/support/index.js” file.

This will then enable “cy.xpath()”, for example:

```
cy.xpath('//ul[@class="todo-list"]/li').should('have.length',3)
```

## Tags

All test blocks (“describe”) and all tests (“it”) should be tagged.

This allows for selective execution of test blocks and/or tests with specific tags.

In addition to filtering for tags, the “cypress-grep” plugin also allows for filtering for a part (substring) of the name.

### Installing the “cypress-grep” plugin

You need to install the “cypress-grep” plugin with:

```
npm install --save-dev cypress-grep
```

Add (change) this to the “cypress.config.js” file:

```
{
  e2e: {
    setupNodeEvents(on, config) {
      require('cypress-grep/src/plugin')(config)
      return config
    }
  }
}
```

You also have to add this

```
const registerCypressGrep = require('cypress-grep')
registerCypressGrep()
```

to the “cypress/support/e2e.js” file.

### Example use of tags

Tags can be used on test blocks (“describe”), and/or on tests (“it”), as in this example:

```
describe('block with a tag', { tags: '@regression' }, () => {
  it('example test one', { tags: ['@firstTag', '@secondTag'] }, () => {
    expect(true).to.be.true
  })
  it('example test two', { tags: '@firstTag' }, () => {
    expect(true).to.be.true
  })
})
```

## Filtering with cypress-grep

The filtering is done from command line (CLI) by environment variables.

When using the “grep” and “grepTags” filters, all of the specs are executed and then the filters get applied. This can be very wasteful, if only a few specs contain the grep in the test titles. Thus for a positive filter, you can pre-filter specs using the “grepFilterSpecs=true” parameter.

### Cypress-grep use examples:

```
# run only the tests with "auth user" in the title
$ npx cypress run --env grep="auth user"

# run tests with "hello" or "auth user" in their titles by separating them with a ";" character
$ npx cypress run --env grep="hello;auth user"

# run tests tagged @fast
$ npx cypress run --env grepTags=@fast

# run only the tests tagged "@smoke" that have "login" in their titles
$ npx cypress run --env grep=login,grepTags=@smoke

# only run the specs that have any tests with "user" in their titles
$ npx cypress run --env grep=user,grepFilterSpecs=true

# only run the specs that have any tests tagged "@smoke"
$ npx cypress run --env grepTags=@smoke,grepFilterSpecs=true

# run only tests that do not have any tags and are not inside blocks that have any tags
$ npx cypress run --env grepUntagged=true
```

## Behaviour-Driven Development (BDD)

The following recipe uses Cucumber and Gherkin in Cypress with an example Behaviour-Driven Development (BDD) test case.

The example test case searches for “New York” on Google Search. It then checks, if the search returns a page with the correct HTML title tag.

With the setup described in this recipe, all “Feature” files must be stored in the “cypress/integration/features” folder, while “Step Definition” files must to be stored in a sub-folder with the name of the feature, for example a “cypress/integration/features/*myfeature*” folder for the steps of a Feature called “myfeature”.

Please note that the file structure must be “cypress/integration/...” and not “cypress/e2e/...”

1. Install the cypress-cucumber-preprocessor plugin:

```
npm install --save-dev cypress-cucumber-preprocessor
```

2. Modify your “cypress.config.js” file to include the following:

```
const { defineConfig } = require("cypress");
const cucumber = require('cypress-cucumber-preprocessor').default

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      on('file:preprocessor', cucumber())
    },
    specPattern: 'cypress/integration/features/*.feature'
  }
});
```

3. Add this to “package.json”:

```
"cypress-cucumber-preprocessor": {
  "nonGlobalStepDefinitions": true
},
```

4. Create a new folder “cypress/integration/features”.

5. In the just created folder “cypress/integration/features”, generate a new file called “GoogleSearch.feature” with this content:

```
Feature: Google Search
  This Feature covers Google Search tests

  Scenario: Do a successful Google Search
    Given I am on the Google Search page
    When I type New York in the Google Search field
    And I click the Google Search button
    Then I should get a page with the correct title
```

6. The Step Definitions must be in a sub-folder with the same name as the Feature. Therefore, create a new folder “cypress/integration/features/GoogleSearch”.
7. In the just created folder “cypress/integration/features/GoogleSearch”, generate a new file called “GoogleSearch.steps.js” with this content:

```
import { Given, When, Then } from 'cypress-cucumber-preprocessor/steps'

Given('I am on the Google Search page', () => {
  cy.visit('https://www.google.com/')
})

When('I type New York in the Google Search field', () => {
  cy.get('#APjFqb').type('New York')
})

And('I click the Google Search button', () => {
  cy.get('div[class="lJ9FBc"] input[value="Google Search"]').click()
})

Then('I should get a page with the correct title', () => {
  cy.title().should('eq', 'New York - Google Search')
})
```

8. You can now run the test, for example with:

```
npx cypress run
```

## Parallel test executions

The “cypress-split” plugin lets you automatically split the entire list of Cypress tests (specs) to run in parallel on any Continuous Integration server (or even when running Cypress locally using Docker containers).

Install the plugin by following the plugin README documentation and use one of the Continuous Integration examples provided to run the tests:

<https://github.com/bahmutov/cypress-split>

### GitHub Actions parallel execution example

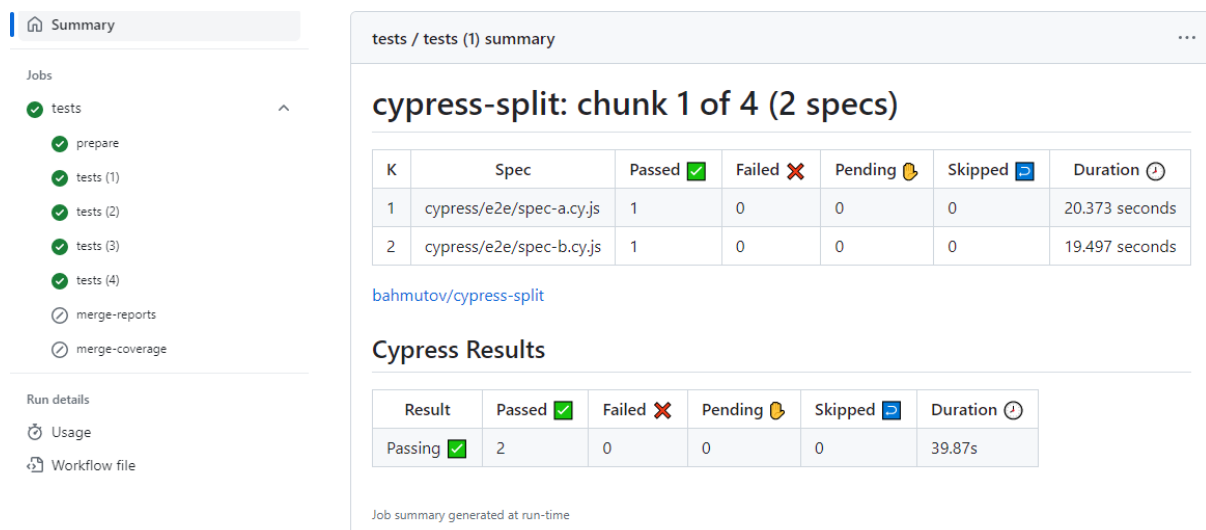
If you use GitHub Actions for Continuous integration, then you can follow this example:

<https://github.com/bahmutov/cypress-split-example>

In GitHub Actions, the reusable workflow is the simplest way to run Cypress tests:

```
name: reusable
on: [push]
jobs:
  tests:
    # use the reusable workflow to check out the code, install
dependencies
    # and run the Cypress tests
    # https://github.com/bahmutov/cypress-workflows
    uses: bahmutov/cypress-workflows/.github/workflows/split.yml@v1
    with:
      n: 4
```

The “cypress-split” plugin even outputs its information to the GitHub Actions report:



The screenshot shows a GitHub Actions report for a job named 'tests'. The report is titled 'tests / tests (1) summary' and displays the following information:

- Jobs:** A list of jobs including 'tests', 'prepare', 'tests (1)', 'tests (2)', 'tests (3)', 'tests (4)', 'merge-reports', and 'merge-coverage'. The 'tests' job is currently selected and marked as successful.
- Run details:** Includes 'Usage' and 'Workflow file'.
- Summary:** 'cypress-split: chunk 1 of 4 (2 specs)'. Below this is a table showing the results for two specs:

K	Spec	Passed	Failed	Pending	Skipped	Duration
1	cypress/e2e/spec-a.cy.js	1	0	0	0	20.373 seconds
2	cypress/e2e/spec-b.cy.js	1	0	0	0	19.497 seconds

Below the table, the report shows the 'Cypress Results' section with a summary table:

Result	Passed	Failed	Pending	Skipped	Duration
Passing	2	0	0	0	39.87s

At the bottom, it notes 'Job summary generated at run-time'.

You can Git clone an example that includes a full demonstration OpenCart shopping process from:

<https://github.com/BrunoBosshard/cypress-split-example>



## Jenkins integration

Cypress can run on many different Continuous Integration servers, such as Jenkins.

Cypress offers a demo web application called “Kitchen Sink” that is available online at <https://example.cypress.io/> . The program code for this demo web application is available at <https://github.com/cypress-io/cypress-example-kitchensink> .

There are two demo “Jenkinsfile” for the “Kitchen Sink” application that show how to run Cypress from Jenkins.

A basic “Jenkinsfile” example is available at <https://github.com/cypress-io/cypress-example-kitchensink/blob/master/basic/Jenkinsfile> .

A more advanced “Jenkinsfile” with full parallel configuration (simultaneous execution of tests on multiple agents/nodes) is available at <https://github.com/cypress-io/cypress-example-kitchensink/blob/master/Jenkinsfile> .

## Docker

Cypress offers several different pre-configured Docker images with a range of Node versions and with and without preinstalled browsers:

<https://github.com/cypress-io/cypress-docker-images>

The Docker images already have the X virtual framebuffer “Xvfb” pre-installed.

If you would like to run Cypress in a Docker container viewable with a Remote Desktop (VNC) client, then this is great starting point for building your own solution:

<https://spin.atomicobject.com/2021/10/14/cypress-running-docker-container/>

If you just want to use an already built Docker image viewable with a Remote Desktop (VNC) client, then check this out:

<https://github.com/piopi/cypress-desktop>

## Reporters

Because Cypress is built on top of the Mocha framework, any reporter built for Mocha can be used with Cypress.

There are also Cypress reporters available for the Junit format, for example:

<https://npm.io/package/cypress-junit-reporter> .

It is also possible to use multiple Cypress reporters at the same time. Often, users are using the default “spec” reporter to write to the terminal, but then also generate an actual “junit” report file.

### Setup the “cypress-mochawesome-reporter”

Please visit <https://npm.io/package/cypress-mochawesome-reporter> for details. The following steps are distilled from these instructions.

1. Install the "cypress-mochawesome-reporter" from command line or terminal:

```
npm i --save-dev cypress-mochawesome-reporter
```

2. Add this to the "cypress.config.js" configuration file:

```
{  
  "reporter": "cypress-mochawesome-reporter"  
}
```

3. Add this to the “cypress/support/index.js” file:

```
import 'cypress-mochawesome-reporter/register'
```

4. Add this to the “package.json” scripts:

```
"scripts": {  
  "cypress:report": "generate-mochawesome-report"  
}
```

### Generate the HTML report using the “cypress-mochawesome-reporter”

1. Delete any old report data as described in the next section (“Delete old test results data before creating a new report”).
2. Run the Cypress tests, for example from command line or terminal with:

```
“npm run cypress:run”
```

3. Generate the HTML report from command line or terminal with:

```
npm run cypress:report
```

4. You can now find the generated HTML report as “index.html” in the “cypress/reports/html” folder.

### Delete old test results data before creating a new report

Old test results data will not automatically get deleted and new test results data will just be added to the existing test results data. It is therefore usually required to delete old test results data before any new round of a test execution and reporting.

There are multiple solutions to automate this. One way is to use an operating system command (such as “rm …”) or batch script to delete all files in the “cypress/results/json” folder. Another solution is to use a Node module such as “rimraf”, for example:

1. Install “rimraf” with:

```
npm install --save-dev rimraf
```

2. Add this to add to the “package.json” scripts:

```
"scripts": {  
  "cypress:deleteResults": "rimraf .\\**\\cypress\\results\\json"  
},
```

3. You can now run this script from command line or terminal with:

```
npm run cypress:deleteResults
```

## Intelligent Code Completion in Microsoft Visual Studio Code

There are two ways to get intelligent code completion (“IntelliSense”) in Microsoft Visual Studio.

### Option 1: Add “triple slash directives” to every program code page

You can add the following triple slash directive on the top of every page where you want to use intelligent code completion:

```
///
```

### Option 2: Add a configuration file

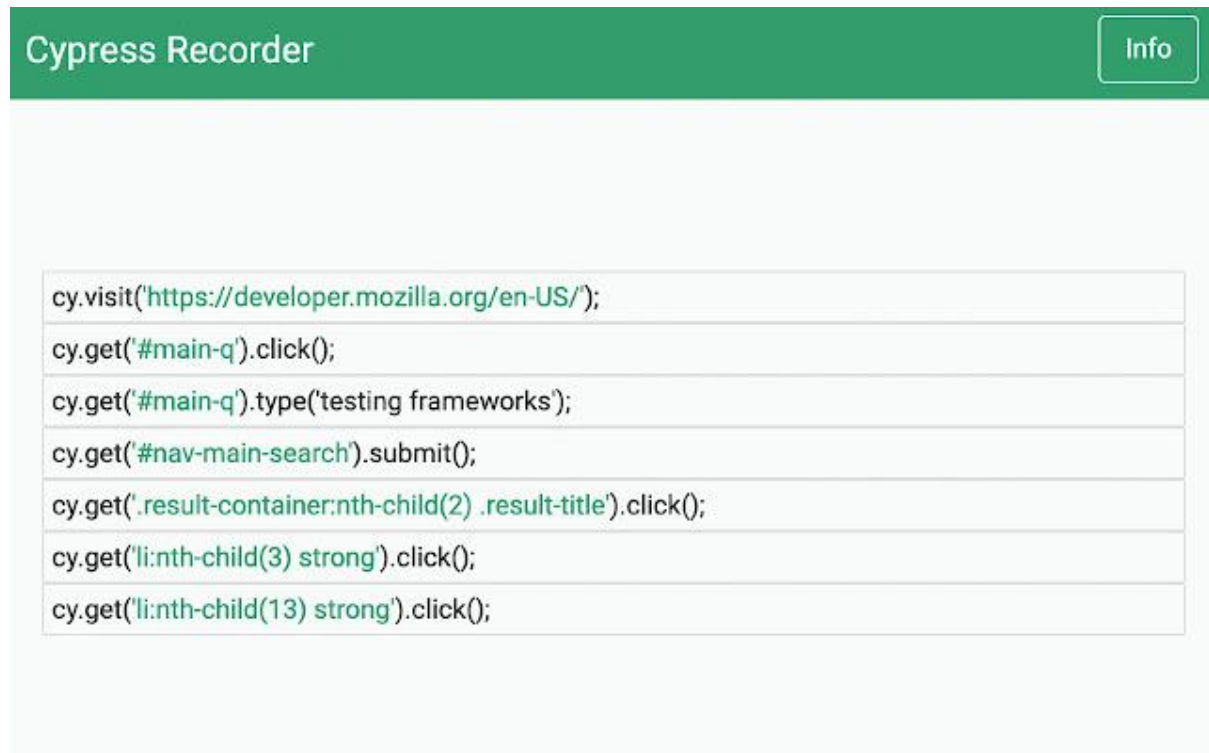
If you automatically want intelligent code completion on all program code pages, then you can add a “`jsconfig.json`” file to the root of your project (where the other configuration files reside).

```
{
  "compilerOptions": {
    "types": ["cypress"],
    "resolveJsonModule": true
  }
}
```

## Cypress Recorder

The Cypress Recorder is a Google Chrome extension that records user interaction within a web application and generates Cypress scripts to allow the developer to replicate that particular session.

<https://chrome.google.com/webstore/detail/cypress-recorder/glcapdcacdfkokcmicllhcjigeodacab>



The screenshot shows the Cypress Recorder interface. At the top, there is a green header with the text "Cypress Recorder" and an "Info" button. Below the header, there is a list of generated Cypress commands, each in a separate row with a light green background and a border. The commands are:

```
cy.visit('https://developer.mozilla.org/en-US/');  
cy.get('#main-q').click();  
cy.get('#main-q').type('testing frameworks');  
cy.get('#nav-main-search').submit();  
cy.get('.result-container:nth-child(2) .result-title').click();  
cy.get('li:nth-child(3) strong').click();  
cy.get('li:nth-child(13) strong').click();
```

Features of the Cypress Recorder:

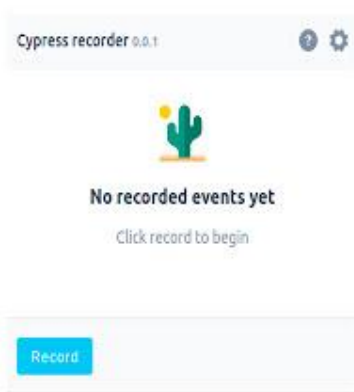
- Record clicks, typing, submits, and navigation in the browser.
- See the scripts render live as they are generated.
- Delete accidental actions.
- Reorder actions as necessary.
- Pause and resume recording within a single session.
- Record navigation within a domain.
- Copy the generated code to your clipboard.

## Cypress Scenario Recorder

The Cypress Scenario Recorder is another Google Chrome extension that records browser interactions and generates Cypress scripts. It is based on the Puppeteer recorder, which is no longer maintained.

<https://chrome.google.com/webstore/detail/cypress-scenario-recorder/fmpgoobcionmfneadjapdabmjfkmfkb>

### 1. Click record



### 2. Track recorded events



### 3. Generate a Cypress script

