


**PAUL VINCENT BEIGANG**

# **PRACTICAL EZE TESTING WITH CODECEPTJS**



**A "hands on" approach  
to modern test automation.**

# Practical End 2 End Testing with CodeceptJS

A “hands on” approach on modern test automation for the web.

Paul Vincent Beigang

This book is for sale at <http://leanpub.com/codeceptjs>

This version was published on 2020-11-17



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2020 Paul Vincent Beigang

# **Tweet This Book!**

Please help Paul Vincent Beigang by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#codeceptjs](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#codeceptjs](#)

# Contents

<b>1. Preparation for End 2 End Testing with CodeceptJS</b>	<b>1</b>
1.1 Application, Feature & Scenario under Test	1
1.2 Node.js Setup	1
1.3 Java Setup	2
1.4 Google Chrome	2
<b>2. Setup CodeceptJS with WebdriverIO</b>	<b>3</b>
<b>3. Create Your First CodeceptJS Test Scenario</b>	<b>5</b>
3.1 Scenario Description	5
3.2 Test Commands	5
<b>4. Run Your First CodeceptJS Test Locally</b>	<b>8</b>
4.1 Start Selenium Server	8
4.2 Run the Test	8
<b>5. Run a Scenario on BrowserStack with the Safari Browser</b>	<b>9</b>
5.1 Sign Up for a Free BrowserStack Account	9
5.2 Get BrowserStack Automate Credentials	9
5.4 Run Scenarios on BrowserStack	10
5.5 Install CodeceptJS BrowserStack Helper	10
5.7 Switch Configs Based on Local or BrowserStack Execution	12
<b>6. Don't Repeat Yourself - Page Object Refactoring</b>	<b>15</b>
6.1 Page Objects in CodeceptJS - Creation & Inclusion	15
6.2 Adding A Page Object Field	16
6.3 Page Object Usage	16
6.4 Page Object Methods	17
<b>7. How to Debug &amp; Fix a Failing E2E Test</b>	<b>18</b>
7.1 Debug Failing Safari Test Run - Ask the Right Questions	18
7.2 What does the error message really say?	18
7.3 What is the context of the error? Which exact steps led to the error?	18
7.4 Debug with the Video Recording on BrowserStack	19
7.5 Try to Reproduce the Failure Locally	19

## CONTENTS

7.6 Use CodeceptJS “pause()” to Start an Interactive Shell Session . . . . .	20
7.7.1 Verify the Failing Locator Locally . . . . .	21
7.7.2 Simulate “wait” with “pause()” . . . . .	21
7.8 Try to factor out timing issues . . . . .	22
7.9 Use Detailed WebDriverIO’s Log Level . . . . .	23
7.9 More Tips on Fixing Test Failures . . . . .	23
<b>8. Run a CodeceptJS Scenario in GitLab’s Continuous Integration (CI) Environment . . .</b>	<b>26</b>
8.1 GitLab CI & Git Repo Setup . . . . .	26
8.2 Setup CI Chrome Scenario Run . . . . .	27
8.3 GitLab CI Artifacts . . . . .	28
<b>9. Delicious Test Reports With Allure . . . . .</b>	<b>29</b>
<b>10. Parallel Execution . . . . .</b>	<b>31</b>
<b>11. Bonus Chapter: Quick Tips &amp; Shortcuts . . . . .</b>	<b>33</b>
Switch Between Local Test Run and Browserstack Test Run Locally . . . . .	33
Utilize BrowserStacks Raw Selenium Logs . . . . .	33
Use Visual Studio Code With Node.js Debugger to Dive Into The CodeceptJS Framework	34
Use PhpStorm With Node.js Debugger to Dive Into The CodeceptJS Framework . . . . .	34
The Best Way to Get Help . . . . .	35

# 1. Preparation for End 2 End Testing with CodeceptJS

## 1.1 Application, Feature & Scenario under Test

In order to write our first automated e2e test we need to know

1. The *url* of the application under test. In our case it is “http://the-internet.herokuapp.com”.
2. The *feature* covered by the test. In our case it is “Editor”.
3. The *scenario* under test. In our case it is “Can be opened and the text input and bold formatting is working”.

Workflow Tip:

My recommendation is to write down the feature and scenario before starting to write the CodeceptJS test. This helps with separating concerns - first we think about *what* we want to test, later when writing the automated e2e test we will think about *how* we will implement what the test should do.

If you are interested in a “feature & scenario blueprint” template which you can easily fill out, just send me an [email](mailto:pbeigang@gmail.com)<sup>1</sup> and I am happy to help out.

## 1.2 Node.js Setup

Node.js version 12 or higher is required to run CodeceptJS tests on your machine.

If you haven't already, install Node.js now.

If you are running Mac OS X, I suggest to use Homebrew to install Node.js.

To install Homebrew, run `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install"`

Then run `brew install node`.

As verification step check the installed version with `node -v`.

---

<sup>1</sup><mailto:pbeigang@gmail.com>

## 1.3 Java Setup

At minimum Java version 8 is required.

Assuming you are on Mac OS X, execute the following two commands

1. Install cask brew `tap caskroom/cask`
2. Install Java `brew cask install java`

As verification step check the installed version with `java -version`.

## 1.4 Google Chrome

Let's make sure that we have the Google Chrome browser installed, I verify the installation by running the following command in my mac terminal `/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --version`.

The expected output is something similar to `Google Chrome 84.0.4147.105`.

If you do not have Google Chrome installed, please install it now, we need it later.

Nice, this is everything we need to setup so that we can bootstrap our CodeceptJS project.

## 2. Setup CodeceptJS with WebdriverIO

We will start with CodeceptJS and its WebDriver helper. This will enable us to run the test in different browsers easily.

The first step is to create a working directory in your workspace with `mkdir my-auto-e2e-tests` and changing to it using `cd my-auto-e2e-tests`.

The next step is to create a `package.json` file. Which is needed to save our project dependencies. To do so, run `npm init -y`.

Following we install CodeceptJS with its WebDriver helper and the `selenium-standalone` package. To do that, run `npm install codeceptjs webdriverio selenium-standalone --save-dev`.

Now we need to run `npx selenium-standalone install` to download the selenium server and the default browser drivers.

Finally we can run `npx codeceptjs init` to setup our first CodeceptJS project. We use the following prompt answers:

- Just press return for the tests location.
- Confirm “WebDriver” as the CodeceptJS helper we want to use.
- The default “./output” path is also well, just press enter.
- We don’t use test localization, just press enter.
- As base url we use `http://the-internet.herokuapp.com`, type it in and press enter.
- “Chrome” as default browser is fine, just press enter.
- The feature being tested is “Editor” (yeah, we are well prepatated), type it in and press enter.
- The suggested filename of the test “Editor\_test.js” sounds good, press enter.

\*

Congratulations, you successfully did the necessary setup to run automated e2e tests with CodeceptJS on your local machine.

Just for the record, my current `package.json` looks like the following:



```
1  {
2    "name": "my-auto-e2e-tests",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "devDependencies": {
13     "codeceptjs": "^2.6.8",
14     "selenium-standalone": "^6.19.0",
15     "webdriverio": "^6.4.0"
16   }
17 }
```

## 3. Create Your First CodeceptJS Test Scenario

Now it is time to create our first CodeceptJS end to end test scenario, to do so we open the recently created file 'Editor\_test.js'.

### 3.1 Scenario Description

Start by replacing “test something” with “Can be opened and the text input and bold formatting is working.” as the scenario description.

### 3.2 Test Commands

#### 3.2.1 Add the First Test Command

Next we want to add the first test step to our first scenario, all available WebDriver helper step actions can be found under <https://codecept.io/helpers/WebDriver>.

The first step is to navigate to the index page of our application under test.

To do so we add `I.amOnPage("/")` as the first step in our scenario.

#### 3.2.2 CodeceptJS's “Smart Assertions”

Then we add `I.click("WYSIWYG Editor")` in the next line. This demonstrates an important concept of CodeceptJS. By clicking on a specific link text (here: “WYSIWYG Editor”) we add something which I call a “smart assertion”. The link must be available before CodeceptJS can click it, but there is not need to type out “wait for link to be available” explicitly. If the link would not be there, the scenario would automatically fail at this step.

#### 3.2.3 Add More Test Steps

Because the content of the WYSISWYG editor itself is injected with an iFrame to the page, we need to switch to that iFrame to be able to interact with the contents of the editor.

Almost every test step in end to end test automation for the web is based on locators. To find a proper locator I always open the Google Chrome developer tools and inspect the available elements.

In our use case the iFrame has a nice id attribute, which is `#mce_0_ifr`.

To switch the iFrame context in the scenario, we write `I.switchTo("#mce_0_ifr")` as the next scenario command.

Next is to fill the text area with some text, we inspect the page again with the Google Chrome dev tools and find `#tinymce` as possible locator.

Then we add the scenario command `I.fillField("#tinymce", "My text bold")`.

To select the last word of the text the test just filled in, we add `I.doubleClick("#tinymce")` as the following scenario.

To be able to click the “bold” button of the editor, we need to switch back from the iFrame context to the parent page, this is done by adding `I.switchTo()` to the next line. Calling `I.switchTo()` without an explicit locator as paramter will switch back to the iFrames parent context which is the surrounding page we want to interact with again.

The next steps is to find a locator for the bold icon. We use the dev tools again and find `#mceu_3`. To click the “bold” icon then, we add `I.click("#mceu_3")` as next scenario command.

Now we want to get rid of the previous made text selection, so we switch back into the iFrame context by adding `I.switchTo("#mce_0_ifr")` and clicking once on the textarea by adding the command `I.click("#tinymce")`.

Last but not least, the test should save a screenshot of the current state of the editor to the previously configured “output” folder of CodeceptJS. Therefor we add `I.saveScreenshot("editor_test.png")` as the final scenario command.

The complete `Editor_test.js` file should now look like

```
1 Feature("Editor");
2
3 Scenario(
4   "Can be opened and the text input and bold formatting is working",
5   I => {
6     I.amOnPage("/");
7     I.click("WYSIWYG Editor");
8     I.switchTo("#mce_0_ifr");
9     I.fillField("#tinymce", "My text bold");
10    I.doubleClick("#tinymce");
11    I.switchTo();
12    I.click("#mceu_3");
13    I.switchTo("#mce_0_ifr");
14    I.click("#tinymce");
15    I.saveScreenshot("Editor_test.png");
16  }
17 );
```

You made it - your first automated end to end test with CodeceptJS is done, congrats for your progress, I really appreciate you and our progress!

# 4. Run Your First CodeceptJS Test Locally

## 4.1 Start Selenium Server

Open a new terminal session and start the selenium server with the following command `npx selenium-standalone start`.

## 4.2 Run the Test

Open a new terminal session and run your first CodeceptJS test with the following command `npx codeceptjs -c configs/codecept.conf.js run Editor_test.js --steps`.

We use following commandline arguments:

1. The config path for our local default config
2. The filename to run only a single test and
3. `--steps` to print out the steps of the test

More run options can be found here <https://codecept.io/commands>.

That's it - Your CodeceptJS test is running, well done.

PS: The screenshot of the last test step is available under `./output/Editor_test.png`.

## 5. Run a Scenario on BrowserStack with the Safari Browser

The nice thing about the Selenium WebDriver concept is, that it is fairly easy to replace the local selenium server with a remote one.

### 5.1 Sign Up for a Free BrowserStack Account

In this course we will use BrowserStack as our provider. Please go ahead and sign up for a free trial at [https://www.browserstack.com/users/sign\\_up](https://www.browserstack.com/users/sign_up) to follow along. After you have signed up you can open the “Automate” dashboard at <https://automate.browserstack.com/dashboard>.

### 5.2 Get BrowserStack Automate Credentials

Go to <https://www.browserstack.com/accounts/settings>, look for the section “Automate” and copy your “Username” and “Access Key”, because we need both in the next step.

### 5.3 Reconfigure CodeceptJS to Run Your Test in the BrowserStack Cloud

To run our scenario against the Safari browser in the BrowserStack cloud we need to reconfigure CodeceptJS a bit. Open the `codecept.conf.js` file and add the following two properties to the `helpers.WebDriver` object:

1. `user: process.env.BROWSERSTACK_USERNAME`
2. `key: process.env.BROWSERSTACK_ACCESS_KEY`

Also change `browser: chrome` to `browser: safari` and add the following code block:

```
1  desiredCapabilities: {
2    "bstack:options": {
3      os: "OS X",
4      osVersion: "Mojave", // -> to run Safari 12 as Safari 13.1 has problems with the\
5      "click" method
6      resolution: "1920x1080",
7      seleniumVersion: "3.141.59",
8    },
9  },
```

## 5.4 Run Scenarios on BrowserStack

As we rely on `process.env.BROWSERSTACK_USERNAME` and `process.env.BROWSERSTACK_ACCESS_KEY` being present as environment variables, we need to make them available to our current environment before we can run the scenario on BrowserStack. To do so, run `export BROWSERSTACK_USERNAME=YOUR_USERNAME BROWSERSTACK_ACCESS_KEY=YOUR_ACCESS_KEY` and verify it with executing `echo $BROWSERSTACK_USERNAME $BROWSERSTACK_ACCESS_KEY`. You should see both values printed to your terminal.

Keep in mind, that you need to export both variables for every new terminal session you start.

Run `npx codeceptjs run --steps` and open the BrowserStack's "Automate Dashboard" <https://automate.browserstack.com>

We can find the running test in the left sidebar. If you watch the test run video which BrowserStack will provide you, you can see that the test stops to early but BrowserStack indicates the test as passed - whats wrong here? Just continue reading, we will fix this soon.

## 5.5 Install CodeceptJS BrowserStack Helper

To improve the scenario status on BrowserStack we install the CodeceptJS BrowserStack helper. To do so, run `npm install codeceptjs-bshelper --save-dev` and add the following code to the already existing `helpers` property in the `codecept.conf.js` file:

```
1  "BrowserstackHelper": {
2    "require": "codeceptjs-bshelper",
3    "user": process.env.BROWSERSTACK_USERNAME,
4    "key": process.env.BROWSERSTACK_ACCESS_KEY
5  },
6  "REST": {}
```

The empty `REST` property is needed to enable the REST helper which is used to communicate with the BrowserStack API.

Re-run your test with `npx codeceptjs run --steps`.

After the test run is finished the CodeceptJS BrowserStack helper outputs the direct link to the job run on BrowserStack, open it and you will see that the test run is marked as failed and the scenario name is used as test name.

PS: The BrowserStack link is public to make it easy to share with all your fellow colleagues. This comes in very handy when adding it to a bug report.

PPS: Add the property `project: "Practical E2E Testing"` and `buildName: "Chapter 5"` to the `helpers.WebDriver.desiredCapabilities` object to tag the BrowserStack job with a project and build name.

The full example config now looks like:

```

1  exports.config = {
2    tests: ".*_test.js",
3    output: "./output",
4    helpers: {
5      WebDriver: {
6        url: "http://the-internet.herokuapp.com",
7        browser: "safari",
8        windowSize: "1920x1080",
9        user: process.env.BROWSERSTACK_USERNAME,
10       key: process.env.BROWSERSTACK_ACCESS_KEY,
11       desiredCapabilities: {
12         "bstack:options": {
13           projectName: "Practical E2E Testing",
14           buildName: "Chapter 5",
15           os: "OS X",
16           osVersion: "Mojave", // -> to run Safari 12 as Safari 13.1 has problems wi\
17 th the "click" method
18           resolution: "1920x1080",
19           seleniumVersion: "3.141.59",
20         },
21       },
22     },
23     BrowserstackHelper: {
24       require: "codeceptjs-bshelper",
25       user: process.env.BROWSERSTACK_USERNAME,
26       key: process.env.BROWSERSTACK_ACCESS_KEY,
27     },
28     REST: {},
29   },
30   include: {

```



```

31     I: "./steps_file.js",
32     editorPage: "./pages/editor.js",
33   },
34   bootstrap: null,
35   mocha: {},
36   name: "my-auto-tests2",
37   plugins: {
38     retryFailedStep: {
39       enabled: false,
40     },
41     screenshotOnFail: {
42       enabled: true,
43     },
44   },
45 };

```

## 5.7 Switch Configs Based on Local or BrowserStack Execution

Usually we want to switch from local execution to execution in the cloud easily. To do so we will extract the BrowserStack specific config in a separate config file and re-use the common config parts. Then we can use the desired config for every run.

Lets create a new folder called `configs` and move `codecept.conf.js` to this folder. Because we now changed the default location of the config file we need to specify the config location on every execution like this `npx codeceptjs run -c configs/codecept.conf.js`.

We also need to reflect the new location for all relatives paths starting with `.`, which need to be changes to `..`, this applies to the tests, output, I and editorPage properties.

Then we are going to create a new empty file called `browserstack.conf.js` for the specific BrowserStack configuration parts. We cut the BrowserStack specific from the `codecept.conf.js` and add it to the `browserstack.conf.js` file, so both of them look like this:

```

1  // codecept.conf.js
2  const { setHeadlessWhen } = require("@codeceptjs/configure");
3
4  // turn on headless mode when running with HEADLESS=true environment variable
5  // export HEADLESS=true && npx codeceptjs run
6  setHeadlessWhen(process.env.HEADLESS);
7
8  exports.config = {
9    tests: "../*_test.js",

```

```

10  output: "../output",
11  helpers: {
12    WebDriver: {
13      url: "http://the-internet.herokuapp.com",
14      browser: "chrome",
15      windowSize: "1920x1080",
16      //logLevel: "trace",
17    },
18  },
19  include: {
20    I: "../steps_file.js",
21    editorPage: "../pages/editor.js",
22  },
23  bootstrap: null,
24  mocha: {},
25  name: "my-auto-tests2",
26  plugins: {
27    retryFailedStep: {
28      enabled: false,
29    },
30    screenshotOnFail: {
31      enabled: true,
32    },
33  },
34 };

```

```

1  // browserstack.conf.js
2  const merge = require("lodash.merge");
3  const { config: commonConfig } = require("../codecept.conf");
4
5  const specificConfig = {
6    helpers: {
7      WebDriver: {
8        browser: "safari",
9        user: process.env.BROWSERSTACK_USERNAME,
10       key: process.env.BROWSERSTACK_ACCESS_KEY,
11       desiredCapabilities: {
12         "bstack:options": {
13           projectName: "Practical E2E Testing",
14           buildName: "Chapter 5",
15           os: "OS X",
16           osVersion: "Mojave", // -> to run Safari 12 as Safari 13.1 has problems wi\

```

```
17 th the "click" method
18     resolution: "1920x1080",
19     seleniumVersion: "3.141.59",
20   },
21 },
22 },
23 BrowserstackHelper: {
24   require: "codeceptjs-bshelper",
25   user: process.env.BROWSERSTACK_USERNAME,
26   key: process.env.BROWSERSTACK_ACCESS_KEY,
27 },
28 REST: {},
29 },
30 };
31
32 exports.config = merge(commonConfig, specificConfig);
```

To make this work also run `npm install lodash.merge --save-dev`.

Afterwards we can run our scenario locally in Chrome with executing `npx codeceptjs run -c configs/codecept.conf.js` and in the BrowserStack cloud against Safari with `npx codeceptjs run -c configs/browserstack.conf.js`

## 6. Don't Repeat Yourself - Page Object Refactoring

In our first scenario we used the locators directly in the scenario. For starting out I recommend to exactly do that. At the point where we begin to repeat ourselves, we start to think about extracting the repetition in a single occurrence. In our first scenario we repeated the locators “#mce\_0\_ifr” and “#tinymce”.

### 6.1 Page Objects in CodeceptJS - Creation & Inclusion

For new test automation engineers the term “page objects” comes in as bullshit bingo term. Don't believe the hype, the concept is simple and solid, no big deal here.

We create our first page object by running “npx codeceptjs gpo”.

As name of the page object we type in “editor”. CodeceptJS suggests “./pages/editor.js” as filepath which is fine, just hit enter.

CodeceptJS then prints the following output which contains two useful pieces

```
1 Update your config file (include section):
2
3   include: {
4     editorPage: './pages/editor.js',
5   },
6
7 Use editorPage as parameter in test scenarios to access this object
```

So we open our “codecept.conf.js” file and we add “editorPage: './pages/editor.js’” to the “include property” which now looks like

```
1 include: {
2   I: './steps_file.js',
3   editorPage: './pages/editor.js'
4 }
```

The other important piece is “Use editorPage as parameter in test scenarios to access this object”. We will get back to this information in section 6.3.

## 6.2 Adding A Page Object Field

As we want to extract the repeated locators “#mce\_0\_ifr” and “#tinymce” to our freshly created page object, we open the file “pages/editor.js” and adding both locators as properties of the object, so that it looks like the following

```

1  const { I } = inject();
2
3  module.exports = {
4    iFrame: `#mce_0_ifr`,
5    textarea: `#tinymce`
6  }
7
8  That way, the locators will be available as "editorPage.iFrame" and "editorPage.text\
9  area" within our test scenario.
```

## 6.3 Page Object Usage

We open the file “Editor\_test.js” and add “editorPage” as paramter for the scenario method, so the codeline looks like

```

1  Scenario('Can be opened and the text input and bold formatting is working.', (I, edi\
2  torPage) => { ... }
```

With this we can access the editorPage object within the scenario, which we will in a second.

Then we replace the usages of “#mce\_0\_ifr” with “editorPage.iFrame” and the usages of “#tinymce” with “editorPage.textarea”.

The full scenario refactored with a page object now looks like this

```

1  Feature('Editor');
2
3  Scenario('Can be opened and the text input and bold formatting is working.', (I, edi\
4  torPage) => {
5    I.amOnPage("/");
6    I.click("WYSIWYG Editor");
7    I.switchTo(editorPage.iFrame);
8    I.fillField(editorPage.textarea, "My text bold");
9    I.doubleClick(editorPage.textarea);
10   I.switchTo();
11   I.click("#mceu_3");
```

```
12     I.switchTo(editorPage.iFrame);
13     I.click(editorPage.textarea);
14     I.saveScreenshot("editor_test.png");
15 });
```

## 6.4 Page Object Methods

In the same manner as for objects you can define and reuse page object methods. This is commonly used for a repetitive set of actions.

Just a theoretical example as we don't need it for the current use case: A page object method would be added to the file "pages/editor.js" and defined like the following:

```
module.exports = {
  iFrame: '#mce_0_ifr',
  textarea: '#tinymce',
  myMethod: function() {
    I.amOnPage("/xyz");
    I.click("#xyz");
    I.waitForText("xyz");
  }
}
```

After injecting the page object as described above the method can be called like this `editorPage.myMethod()`.

# 7. How to Debug & Fix a Failing E2E Test

Fine, fine, fine!

We now have a running automated e2e test which we can literally run on a huge amount of different browsers and OS combinations, which is great - Congratulations on your constant learning progress.

Buuuuut... somehow the test, which just was passing in Chrome, does fail in Safari.

I can already hear you asking: “Why is that?”

That is a very good question, prepare yourself to ask this question quite often when writing cross-browser end to end tests.

If the error does not appear for you, there is another common example. A so called “flaky” test, which means: Sometimes the test passes, and sometimes the test does not pass.

How can this be?

Well, with e2e tests we test all systems in an integrated way, which means there are a lot of things happening and moving in a short timespan. One common reason for flaky tests are inconsistent timings, as the webserver under test not always responds in the same time, this might be the reason for your test to be flaky.

## 7.1 Debug Failing Safari Test Run - Ask the Right Questions

I recommend to find answers to the following questions step by step, at some point, we will find a way to fix the flaky test.

## 7.2 What does the error message really say?

In our case it says `Element "#mce_0_ifr" was not found by text|CSS|XPath.`

## 7.3 What is the context of the error? Which exact steps led to the error?

As we are writing automated end to end tests, we already have the steps but we still need to understand the context, in a “bigger” test scenario the context is not always trivial.

In our case the test steps are:

```
1 Scenario Steps:
2
3 - I.switchTo("#mce_0_ifr") at Test.I (editor_test.js:8:7)
4 - I.click("WYSIWYG Editor") at Test.I (editor_test.js:7:7)
5 - I.amOnPage("/") at Test.I (editor_test.js:6:7)
```

So when CodeceptJS tries to switch to the iFrame with the ID “mce\_0\_ifr”, the scenario fails.

I would conclude the iFrame is just not there when CodeceptJS tries to interact with it. How can we confirm this hypothesis?

## 7.4 Debug with the Video Recording on BrowserStack

Thanks to the `codeceptjs-bshelper` we already have the BrowserStack test run link available, just open it, it looks like <https://automate.browserstack.com/builds/1e498e0b710fd8ec8d99432fdb37fce7292716cc/s?token=xxxx>.

There you go, just click on the “Play” button and see the recorded failing test running.

While watching the video it seems that the WYSIWYG editor is loading as expected, but somehow CodeceptJS can’t switch to the iFrame.

Maybe CodeceptJS tries to switch to the iFrame “too early”, before the WYSIWYG editor is loaded entirely and available to interact with it.

## 7.5 Try to Reproduce the Failure Locally

If you can run the targeted browser on your local machine, now it’s a good time to do. This brings us enhanced debug possibilities.

I am working on a Mac so I can run the test against Safari locally.

Replace the values for the following to properties `browser`: `safari` and `windowSize`: `"maximize"` in the configuration. The `helpers.WebDriver` object should now look like:

```
1 WebDriver: {
2   url: "http://the-internet.herokuapp.com",
3   browser: "safari",
4   windowSize: "maximize"
5 }
```



To run the test we need to start a the Selenium server locally first, we use `npx selenium-standalone start`.

In a new terminal window we run `npx codeceptjs -c configs/codecept.conf.js run --steps` to run the test.

Yeb, we get the same error `Element "#mce_0_ifr" was not found by text|CSS|XPath`.

## 7.6 Use CodeceptJS “`pause()`” to Start an Interactive Shell Session

Now I suggest to add the `pause()` call before the failing step in the test, it should look like following now:

```
1 Feature("Editor");
2
3 Scenario(
4   "Can be opened and the text input and bold formatting is working.",
5   I => {
6     I.amOnPage("/");
7     I.click("WYSIWYG Editor");
8     pause();
9     I.switchTo("#mce_0_ifr");
10    I.fillField("#tinymce", "My text bold");
11    I.doubleClick("#tinymce");
12    I.switchTo();
13    I.click("#mceu_3");
14    I.switchTo("#mce_0_ifr");
15    I.click("#tinymce");
16    I.saveScreenshot("editor_test.png");
17    ...
```

Run the test again with `npx codeceptjs run editor_test.js --steps`, the terminal shows something like this:

```
1 Editor --
2   Can be opened and the text input and bold formatting is working.
3   Test: I
4     I am on page "/"
5     I click "WYSIWYG Editor"
6   Interactive shell started
7   Use JavaScript syntax to try steps in action
8   - Press ENTER to run the next step
9   - Press TAB twice to see all available commands
10  - Type exit + Enter to exit the interactive shell
11  I.
```

## 7.7.1 Verify the Failing Locator Locally

Maybe the locator is wrong all together, let us just make sure the locator is working in Safari.

CodeceptJS stops the test execution where we called `pause()` and presents an interactive shell to us. From here we can inspect the application under test live in the browser.

Just right click in the Safari window and confirm to cancel the automation session. For now we confirm to cancel the automation session as we want to open the developer tools.

Right click again and select “Inspect Element”.

Press “cmd + f” and search for “#mce\_0\_ifr”.

On the left of the search input we can see one result, which means we successfully verified the locator in Safari, great.

## 7.7.2 Simulate “wait” with “pause()”

We can also “simulate” a wait step. Just press enter to continue the execution. From now on CodeceptJS executes all test steps one by one. You can see now, that the `I.switchTo("#mce_0_ifr")` step passed since no error is logged:

```

1 Editor --
2   Can be opened and the text input and bold formatting is working.
3   Test: I
4     I am on page "/"
5     I click "WYSIWYG Editor"
6   Interactive shell started
7   Use JavaScript syntax to try steps in action
8   - Press ENTER to run the next step
9   - Press TAB twice to see all available commands
10  - Type exit + Enter to exit the interactive shell
11  I.
12    I switch to "#mce_0_ifr"
13  I.

```


## 7.8 Try to factor out timing issues

In the previous step we learned that adding the `pause()` command helped make the scenario pass, so it is still valid to follow a hypothesis where we say: “Well, Safari seems to be a bit slower when processing the WYSIWYG editor than Chrome, maybe this is the reason why the test is passing in Chrome but failing in Safari.”

Let’s replace the `pause()` call with the following step `I.waitForElement("#mce_0_ifr")`, which will basically verify the existence of “#mce\_0\_ifr” before interacting with it. Save and re-run the test with `npx codeceptjs run -c configs/codecept.conf.js --steps`.

Well, it’s working now! The scenario passed with the following output:

```

1 Editor --
2   Can be opened and the text input and bold formatting is working.
3   Test: I
4     I am on page "/"
5     I click "WYSIWYG Editor"
6     I wait for element "#mce_0_ifr"
7     I switch to "#mce_0_ifr"
8     I fill field "#tinymce", "My text bold"
9     I double click "#tinymce"
10    I switch to
11    I click "#mceu_3"
12    I switch to "#mce_0_ifr"
13    I click "#tinymce"
14    I switch to
15    I save screenshot "editor_test.png"
16   OK in 2400ms

```

```

17
18 Test has Passed
19
20 OK | 1 passed // 3s

```

Lets see the scenario passing against Safari in the BrowserStack cloud as well, I think you already remember the command `npx codeceptjs run -c configs/browserstack.conf.js --steps`.

And voila - it is also passing in Safari on BrowserStack.

## 7.9 Use Detailed WebdriverIO's Log Level

WebdriverIO's log level can be a very helpful source of information. To enable it add the `logLevel: trace` property to the config so it looks like this:

```

1  helpers: {
2    WebDriver: {
3      url: "http://the-internet.herokuapp.com",
4      browser: "safari",
5      windowSize: "maximize",
6      logLevel: "trace",

```

Now run the scenario again and you will see detailed output in the terminal.

For more `logLevel` options see <https://webdriver.io/docs/options.html#loglevel>.

## 7.9 More Tipps on Fixing Test Failures

Following there are some more very, very valuable steps when we did not succeed until here with fixing a failing scenario.

### Update to latest CodeceptJS version

The first step is to check which CodeceptJS version we are currently using, `devDependencies` in our `package.json` is telling us that we are running `"codeceptjs": "2.6.8"`, that could be another version in your case.

Then go to <https://www.npmjs.com/package/codeceptjs> and check for the latest version of CodeceptJS available. We can see that 2.6.10 is already available, lets change the `package.json` to include `"codeceptjs": "2.6.10"` and run `npm install`.

Check <https://codecept.io/changelog> for breaking changes and keep in mind that you can easily roll-back the version at any time, I think you already know how.

You can verify the current CodeceptJS version by running `npx codeceptjs --version`, it just logs its version number right to the terminal:

```
1 npx codecept.js
2 CodeceptJS v2.1.0
```

## Update to latest WebDriverIO version

The first step is to check which WebDriverIO version we are currently using, `devDependencies` in our `package.json` is telling us that we are running `"webdriverio": "6.4.2"`, that could be another version in your case.

Then go to <https://www.npmjs.com/package/webdriverio> and check for the latest version of CodeceptJS available. We can see that 6.5.2 is already available, let's change the `package.json` to include `"webdriverio": "6.5.2"` and run `npm install` again.

Check <https://github.com/webdriverio/webdriverio/blob/master/CHANGELOG.md> for breaking changes.

## Update to latest selenium-standalone & selenium version

You should do the same process for <https://www.npmjs.com/package/selenium-standalone> when running tests locally.

Keep also in mind that selenium is changing over time, the changelog can be found here <https://raw.githubusercontent.com/SeleniumHQ/selenium-standalone/master/CHANGELOG.md>. selenium-standalone's readme explains how to use a specific selenium version.

When running tests on BrowserStack update the according property in `helpers.WebDriver.desiredCapabilities.s` in `codecept.conf.js`. You can find the latest versions available on BrowserStack using BrowserStack's capabilities generator <https://www.browserstack.com/automate/capabilities>.

## Update to latest Browser Driver & Browser Version

Consider also to check the specific browser version your test is running against, locally (just open the browser and check the version) or on BrowserStack (see capabilities generator above).

There is one more thing, besides the browser version we also have the browser driver which can be outdated as well, locally selenium-standalone is taking care of that, on BrowserStack you can configure the browser driver version in the capabilities as well.

## Get a Better Understanding of your AUT

Sometimes you will end up in a situation where you feel like you have tried everything in the world to fix a failing scenario, but you can't get it to work - until now. That's a good time to broaden your perspective and look at the application under test from a different angle.

Ask yourself: What is exactly happening in the context of the test failure?

In my past experience such situations were often related to JavaScript event handling. For example scenario tried to click a button, no, the scenario even clicked the button but the application doesn't reacted properly to the click. Therefore the following test assertion was failing. But how can this be the case?

You need to understand that JavaScript also needs time to process the code, for example to attach an event listener to a button. Generally speaking: If the app under test is a bit slow (heavy JS) and the test execution is fast (Chrome), it may be the case that for the previous example the test clicks on the button before the event listener is attached, therefore the app won't react properly and the scenario fails.

## Screen Resolution

Another hard to find, but common ground for failing tests is the screen resolution of the machine which runs a scenario. At first sight it may seem obvious, but later when moving to CI and running tests in headless mode, it is easy to forget about the resolution in different environments.

Basically we have the following three places to configure the screen resolution:

1. For a local scenario runs, configure `helpers.WebDriver.windowSize` in the CodeceptJS config.
2. In CI set the `SCREEN_WIDTH` and `SCREEN_HEIGHT` environment variables to configure the selenium-standalone docker container.
3. For BrowserStack set the desired resolution in the CodeceptJS config via `helpers.WebDriver.desiredCapabilities`.

## Ask for Help

There are tremendous ways to connect with the friendly CodeceptJS community and get help from talented people from all over the world. Please join us!

### Project Chat Channels

- CodeceptJS Slack Chat: [https://join.slack.com/t/codeceptjs/shared\\_invite/enQtMzA5OTM4NDM2MzA4LTNi](https://join.slack.com/t/codeceptjs/shared_invite/enQtMzA5OTM4NDM2MzA4LTNi)
- Community Forum: <https://codecept.discourse.group/>
- StackOverflow CodeceptJS tagged Questions <https://stackoverflow.com/questions/tagged/codeceptjs>
- For bug reports use CodeceptJS GitHub issues <https://github.com/Codeception/CodeceptJS/issues>
- If you need support for webdriverio, join the webdriverio Gitter chat: <https://gitter.im/webdriverio/webdriverio>
- If you need Selenium support, join the Selenium IRC: See <https://www.seleniumhq.org/support/>

### Involve BrowserStack's Support

If you get stuck with the BrowserStack config, do not hesitate to contact their support, they response very fast and are eager to help.

The best way is to use the "Contact support" button from a session run, because it will automatically add the session id to the support request which makes it a lot easier for the support to follow along and help you out.

## 8. Run a CodeceptJS Scenario in GitLab's Continuous Integration (CI) Environment

OK, lets take some time to recap what we have done so far: We setup CodeceptJS and all it's dependencies, planned & created our first automated end to end test, run it locally, run it in the cloud in a different browser then Google Chrome, debugged a cross-browser test automation issue and fixed that. Also we got a great understanding about all moving parts of the automation landscape and where to look when things go wrong.

One quick todo for you: Please change back the property `helpers.WebDriver.browser` to "chrome" in the `codecept.conf.js` file.

So what is the next piece missing on our test automation success journey?

In my opinion, it is Continuous Integration.

The automated e2e tests should be prepared to run on every code change, automatically.

Let us not loose any time and learn on a real example right away how to do that, are you ready to follow along?

### 8.1 GitLab CI & Git Repo Setup

Register for a free GitLab account at [https://gitlab.com/users/sign\\_in](https://gitlab.com/users/sign_in) and log in.

Then create a new project on GitLab. After that, go to <https://gitlab.com/profile/keys> and add your public ssh key to be able to push to the freshly created Git repository on gitlab.com.

In your codeceptjs project folder create a new file called `.gitignore` with the following contents:

```
1 # .gitignore
2
3 node_modules
4 output
```

Then execute the following commands to setup a local Git repository, link it to the remote repository and push your local code into the remote repository.

```
1 git init
2 git remote add origin git@gitlab.com:YOUR_GITLAB_USERNAME/YOUR_PROJECT_SLUG.git
3 git add .
4 git commit -m "Initial commit"
5 git push -u origin master
```

That's it for the setup, let us head over to the next step.

## 8.2 Setup CI Chrome Scenario Run

GitLab CI is configured through a `.gitlab-ci.yml` file which just need to be added to your repository and pushed into the remote, GitLab will pick it up automatically and run the configured CI job.

Create a new file in the root of the repository called `.gitlab-ci.yml` with the following contents:

```
1 image: node:12.16.1
2
3 stages:
4   - chrome-ci
5
6 variables:
7   SCREEN_WIDTH: 1920
8   SCREEN_HEIGHT: 1080
9
10 services:
11   - name: selenium/standalone-chrome:3.141.59-20200826
12
13 e2e-tests-chrome:
14   stage: chrome-ci
15   before_script:
16     - npm install
17   script:
18     - npx codeceptjs run -c configs/ci.conf.js --steps
19   after_script:
20     - cp -r output/ ci_artifacts/
21   artifacts:
22     name: "$CI_JOB_STAGE-$CI_COMMIT_REF_NAME"
23     paths:
24       - ci_artifacts/
25   when: always
```

Also we need to adapt CodeceptJS's configuration a bit. To do so, we create a new file called `ci.conf.js` in the `configs` folder.

Add the following contents to `ci.conf.js`:



```
1  const merge = require("lodash.merge");
2  const { config: commonConfig } = require("./codecept.conf");
3
4  const specificConfig = {
5    helpers: {
6      WebDriver: {
7        host: process.env.SELЕНИUM_STANDALONE_CHROME_PORT_4444_TCP_ADDR,
8        protocol: "http",
9        port: 4444,
10      },
11    },
12  };
13
14  exports.config = merge(commonConfig, specificConfig);
```

Basically we just changed how the Selenium server is accessed in CI.

We are good to go, CI integration is almost there, how great is that?

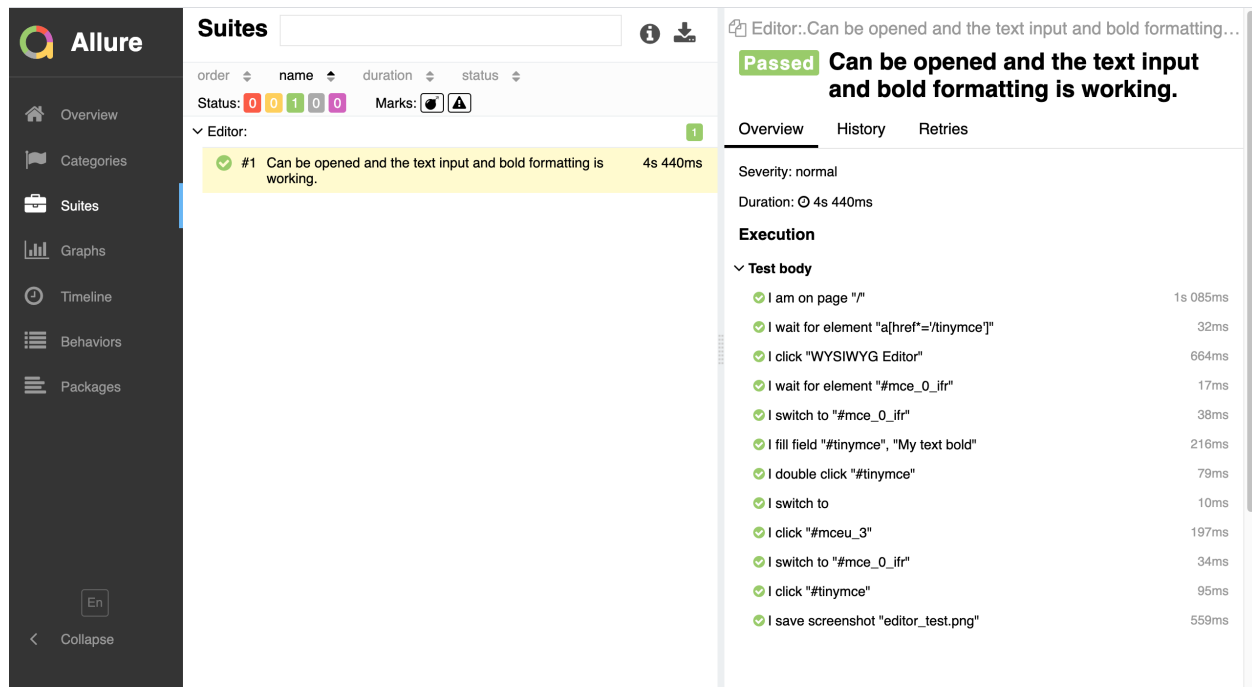
You just need to run `git add .gitlab-ci.yml codecept.conf.js && git commit -m "Add chrome ci test job." && git push` and head over to [https://gitlab.com/YOUR\\_GITLAB\\_USERNAME/YOUR\\_PROJECT\\_SLUG/-/jobs](https://gitlab.com/YOUR_GITLAB_USERNAME/YOUR_PROJECT_SLUG/-/jobs) to watch your CodeceptJS test running automatically in GitLab CI, for free.

## 8.3 GitLab CI Artifacts

A nice bonus for you: The GitLab CI configuration already serves so called “artifacts” for you, as you can see in the configuration we first copy the “output” folder to a different location as GitLab ignores all git ignored paths and won’t serve them as artifacts and then using that temporary location as base for the artifacts to be served via GitLab’s UI.

As an example head over to <https://gitlab.com/paulvincent/codeceptjs-e2e-testing/-/jobs/764697577/artifacts/browse> artifacts/ and you can open the screenshot generated by the last step `test I . saveScreenshot("editor_test.png");`, executed in GitLab’s CI system.

## 9. Delicious Test Reports With Allure



The screenshot displays the Allure web interface. On the left is a dark sidebar with navigation links: Overview, Categories, Suites, Graphs, Timeline, Behaviors, and Packages. The main area is titled 'Suites' and shows a table of test results. A single test is listed with a green checkmark, the name '#1 Can be opened and the text input and bold formatting is working.', and a duration of '4s 440ms'. To the right of the table, a detailed view of the test is shown, including a 'Passed' status, a description, and a list of execution steps with their durations.

order	name	duration	status
1	#1 Can be opened and the text input and bold formatting is working.	4s 440ms	Passed

**Execution**

- I am on page "/" (1s 085ms)
- I wait for element "a[href='/tiny\_mce']" (32ms)
- I click "WYSIWYG Editor" (664ms)
- I wait for element "#mce\_0\_ifr" (17ms)
- I switch to "#mce\_0\_ifr" (38ms)
- I fill field "#tiny\_mce", "My text bold" (216ms)
- I double click "#tiny\_mce" (79ms)
- I switch to (10ms)
- I click "#mceu\_3" (197ms)
- I switch to "#mce\_0\_ifr" (34ms)
- I click "#tiny\_mce" (95ms)
- I save screenshot "editor\_test.png" (559ms)

Allure example report

The best test suite can be almost useless without a proper reporting system, so let's get started with the last stop of our practical CodeceptJS journey.

Allure is a pretty neat reporting standard which is supported by CodeceptJS and my first choice when it comes to useful reports.

We can install Allure by running `npm install allure-commandline --save-dev`.

Then in the `codecept.conf.js` file add "allure" under the "plugins" object:

```
1 "plugins": {  
2   "allure": {}  
3 }
```

Then re-run your test with `npx codeceptjs run -c configs/codecept.conf.js --plugins allure`.

When finished, open a new terminal tab, change into the project directory and serve the allure reports with `npx allure serve output`.

That's the way to get sexy CodeceptJS test result reports :) - Enjoy!

Another nice thing about Allure is, that you get step timings in the report, which can be pretty useful for analyzing the scenario performance.

# 10. Parallel Execution

Parallel scenario execution is a big deal when you aim for making your test suite really valuable. What you want is fast test feedback, and that's exactly what parallel test execution is made for.

Assume that we have ten scenarios, each running two minutes. If you run them serial, it would take 20 minutes until the test suite result is available.

With parallel execution, it is possible to cut down the feedback time a lot. If you have enough resources and can run ten scenarios in parallel, the execution time could go down to something in between three to five minutes. The logical conclusion here would be to assume that the overall execution time with ten parallel threads should be two minutes, I was expecting the same, but reality taught me differently. Just add a few minutes to the logical result and it will be close to what will happen.

The most up to date and future proof concept to run test scenarios in parallel with CodeceptJS is the `run-workers` command, please find the according [documentation here](#)<sup>2</sup>.

To make an actual `run-workers` execution we duplicate the existing scenario in `Editor_test.js`. Just open the file, copy the whole scenario block and paste it below the existing scenario. Then prepend a "2" in front of the second scenario description so that we can distinguish both scenarios, the whole `Editor_test.js` file should now look like:

```
1 Feature("Editor");
2
3 Scenario(
4   "Can be opened and the text input and bold formatting is working.",
5   (I, editorPage) => {
6     I.amOnPage("/");
7     I.waitForElement("a[href*='/tinymce']");
8     I.click("WYSIWYG Editor");
9     I.waitForElement(editorPage.iFrame);
10    I.switchTo(editorPage.iFrame);
11    I.fillField(editorPage.textarea, "My text bold");
12    I.doubleClick(editorPage.textarea);
13    I.switchTo();
14    I.click("#mceu_3");
15    I.switchTo(editorPage.iFrame);
16    I.click(editorPage.textarea);
17    I.saveScreenshot("editor_test.png");
18  }
```

---

<sup>2</sup><https://codecept.io/parallel/#parallel-execution-by-workers>

```
19 );
20
21 Scenario(
22   "2 Can be opened and the text input and bold formatting is working.",
23   (I, editorPage) => {
24     I.amOnPage("/");
25     I.waitForElement("a[href*='/tinymce']");
26     I.click("WYSIWYG Editor");
27     I.waitForElement(editorPage.iFrame);
28     I.switchTo(editorPage.iFrame);
29     I.fillField(editorPage.textarea, "My text bold");
30     I.doubleClick(editorPage.textarea);
31     I.switchTo();
32     I.click("#mceu_3");
33     I.switchTo(editorPage.iFrame);
34     I.click(editorPage.textarea);
35     I.saveScreenshot("editor_test.png");
36   }
37 );
```

Great, now you can run both scenarios in parallel with the following command `npx codeceptjs run-workers 2 -c configs/codecept.conf.js`.

Note the “2” after `run-workers`, it actually determines the parallel thread count. Be aware that a lot more threads will consume a good amount of hardware resources.

# 11. Bonus Chapter: Quick Tipps & Shortcuts

## Switch Between Local Test Run and Browserstack Test Run Locally

For switching between local testing and cloud testing a nice small bash alias will help you to export the required BrowserStack environment variables and also disable them again.

Open a new command line window and type `nano ~/.bashrc`.

Add a new line with this contents, make sure to replace `MY_USERNAME` and `MY_KEY` with the appropriate values which you will find in your BrowserStack profile:

```
1 alias enable-bs="export BROWSERSTACK_USERNAME=MY_USERNAME && export BROWSERSTACK_ACC\
2 ESS_KEY=MY_KEY"
3 alias disable-bs="export BROWSERSTACK_USERNAME= && BROWSERSTACK_ACCESS_KEY="
```

Save the file.

Source the config with running `source ~/.bashrc`.

Then you can run `enable-bs`. To verify the alias run `echo $BROWSERSTACK_USERNAME`. It will print out your username if everything is working, also double check the key with `echo $BROWSERSTACK_KEY`.

Then run your scenario on BrowserStack with `npx codeceptjs run -c configs/browserstack.conf.js`.

Make sure to *disable BrowserStack* afterwards if you want to execute the scenario locally again, therefor you can use our new bash alias `disable-bs`.

## Utilize BrowserStacks Raw Selenium Logs

If you open a specific scenario run on the BrowserStack dashboard there is the “Selenium Logs” tabs available, which can be a very valuable source of information. Select it and then click on “View Raw Selenium Logs”. This makes the whole Selenium debug available to you.

If you are digging down the road of debugging a specific thing make sure to have a look into this logs as well.

## Use Visual Studio Code With Node.js Debugger to Dive Into The CodeceptJS Framework

If you need to debug a more specific thing it is highly recommended to use a debugger of your favorite IDE.

Open Visual Studio Code and go to the “run view” on the left side.

Then we will create a launch configuration file to store the debugging configuration.

Click on “create a launch.json file”.

Select “Node.js” and change the config to the following contents:

```
1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "type": "node",
6       "request": "launch",
7       "name": "CodeceptJS",
8       "skipFiles": ["<node_internals>/**"],
9       "args": ["run", "-c", "configs/codecept.conf.js", "--grep", "Editor"],
10      "program": "${workspaceFolder}/node_modules/.bin/codeceptjs"
11    }
12  ]
13 }
```

Open the “Editor\_test.js” file and click left to line number 6, to add a breakpoint to that execution point.

Switch back to the “run view” on the left sidebar and click the green triangle right before “CodeceptJS”.

This will start the editor scenario in debug mode and will pause execution right where we placed the breakpoint.

From here you can start analyzing values, the execution flow and a lot more - use it wisely!

For more debug configuration info for Visual Studio Code please check <https://code.visualstudio.com/docs/editor/de>

## Use PhpStorm With Node.js Debugger to Dive Into The CodeceptJS Framework

In this specific example I will walk you through setting up the Node.js debugger in PhpStorm.

Start PhpStorm and click “Open”. Open your e2e test repository.

Then click “Run -> Debug -> Edit Configurations”.

Top left click on the “+” icon and select “Node.js”. “Name” it “CodeceptJS”.

For “JavaScript file” fill in `node_modules/codeceptjs/bin/codecept.js`.

For “Application paramters” fill in `run -c configs/codecept.conf.js`.

Click “Apply”, then click “close”.

Open the file `Editor_test.js` and click on the empty space right next to line number “6”. This will add a “red breakpoint” symbol next to number and it means, that the debugger will stop the program execution at this point.

Now click “Run -> Debug CodeceptJS”.

## The Best Way to Get Help

So what is your goal when asking for help? Your goal is to find a solution fast.

Even if it feels a bit counter intuitive and may cost you some willpower to overcome a initial resistance, my best advice I can give is the following:

If you are really craving for help and solution, make sure that you make your question / bug / ask reproducible. Meaning that anybody you is willed to help out can start with the same context as you have locally on your side.

Specifically the best thing is to publish a public repository which is automaticall executed in CI.

And this is not very difficult, because you already learned anything you need to know to do this.

The easiest way is to just fork <https://gitlab.com/paulvincent/codeceptjs-e2e-testing/> and commit and push your changes to the repo, thex will be executed automatically already.

With making your case reproducible for the “answering side”, you will increase the probability to get a high quality answer a lot - trust me.

Please also make sure to add all the needed informationen to your request: exact failure message, OS, Java version, Node.js version, CodeceptJS version, enabled helpers, enabled helpers version (if applicable), how you run selenium (if applicable) and your CodeceptJS configuration file.

If you follow my advice to publish a public repo, all the information is already there ;)