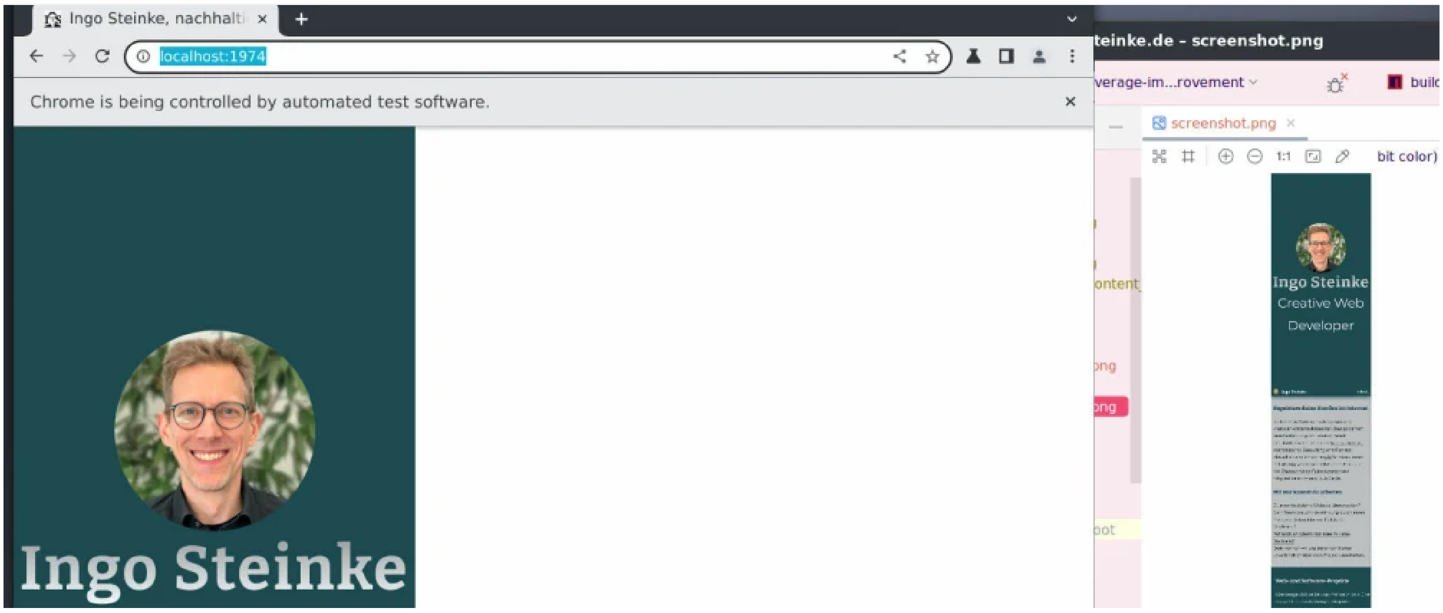


Edit Manage Stats

Ingo Steinke
Posted on Apr 3

2

Comparing Full Page Screenshots, Cross-Device

#webdev #testing #beginners #javascript

There are different tools for visual screenshot testing, like [BackstopJS](), and Cypress.io also has a [screenshot()]() function. In this series, I focus on writing tests with [CodeceptJS]().

Test-Driven Development vs. Regression Tests

Assuming pixel-perfect layout requirements, we could use screenshot testing for visual test-driven development: take a UI design screen (from Figma, Zeplin, Sketch, XD), put it in our web page prototype, and initialize our first screenshot as a base image in our testing tool.

Then we can either proceed and code the website ourselves, pass it to a junior coworker or a tool, like some bloggers suggested using chatGPT 4 or a custom web development automation tool.

Automated Pixel-Perfect Web Development?

In reality, I wouldn't work like that for various reasons. Even if we accept pixel-perfect web development, what about responsive design? Okay, maybe we have two or even three adaptive variants of our screen designs, so we can test a mobile, a table, and a desktop variation. In this post, we will see how to do that using CodeceptJS with its resemble helper.

But replicating a full page should not be the first step of our web project. Instead, we want to build a new site from composable, reusable small components. This could still be a use case for test-driven design-system validation from the beginning, but in practice, I would start introducing screenshot tests only when I have assembled some components to verify them on a demo page. Most projects did not even start with any concept like that.

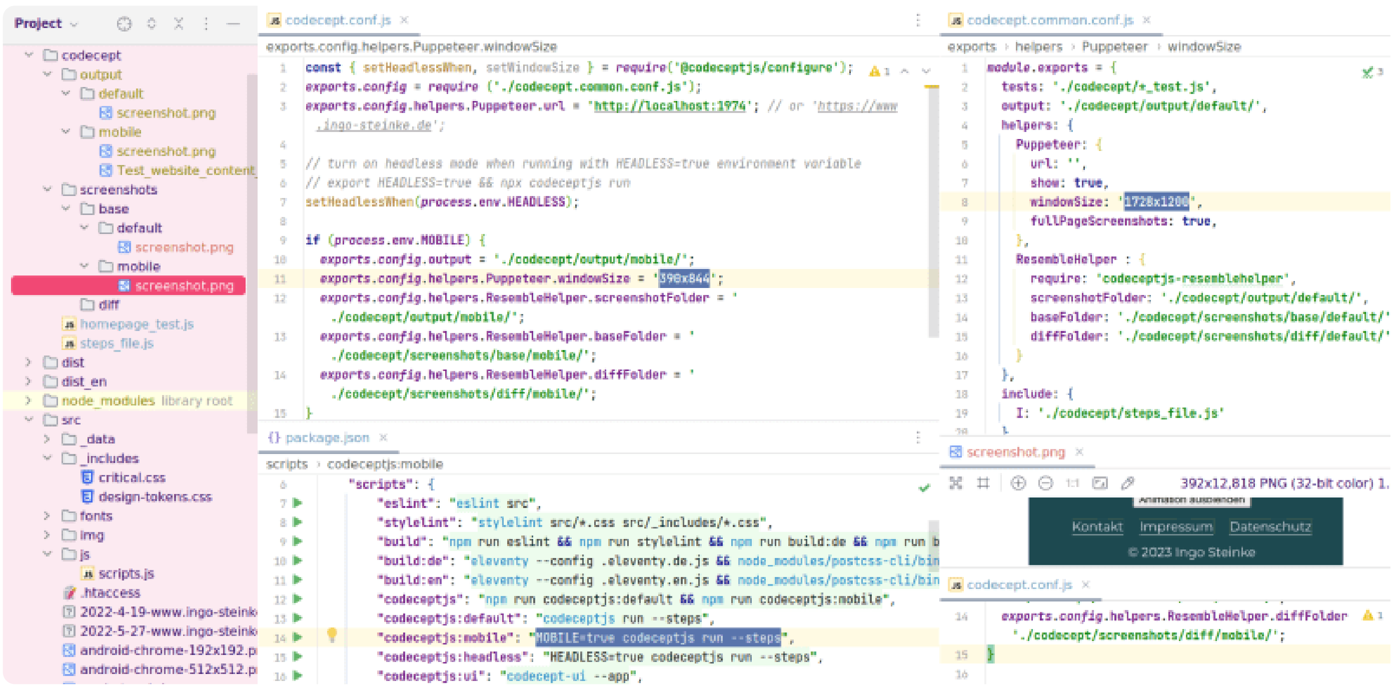
But we can still use automated screenshot comparison for regression testing to prevent breaking changes to the work that we already completed.

Full Page Screenshots and Alternative Viewport Sizes

Extending my screenshot tests, I discovered the useful configuration option to set `fullPageScreenshots: true`. When I wanted to set different window sizes, I was not the first one struggling to resize the browser window in CodeceptJS programmatically.

Screen Size Switch in CodeceptJS (Workaround)

I gave up in favor of the pragmatic approach to define different test setups in `package.json` and redefine the `config.helpers.Puppeteer.windowSize` based on an enviroment variable set when running the test.



codecept.conf.js — exports.config.helpers.Puppeteer.windowSize

```js
const { setHeadlessWhen, setWindowSize } = require('@codeceptjs/configure');
exports.config = require ('./codecept.common.conf.js');
exports.config.helpers.Puppeteer.url = 'http://localhost:1974'; // or 'https://www
  .ingo-steinke.de';

// turn on headless mode when running with HEADLESS=true environment variable
// export HEADLESS=true && npx codeceptjs run
setHeadlessWhen(process.env.HEADLESS);

if (process.env.MOBILE) {
  exports.config.output = './codecept/output/mobile/';
  exports.config.helpers.Puppeteer.windowSize = '390x844';
  exports.config.helpers.ResembleHelper.screenshotFolder = '
    ./codecept/output/mobile/';
  exports.config.helpers.ResembleHelper.baseFolder = '
    ./codecept/screenshots/base/mobile/';
  exports.config.helpers.ResembleHelper.diffFolder = '
    ./codecept/screenshots/diff/mobile/';
}
```

codecept.common.conf.js — exports › helpers › Puppeteer › windowSize

```js
module.exports = {
  tests: './codecept/*_test.js',
  output: './codecept/output/default/',
  helpers: {
    Puppeteer: {
      url: '',
      show: true,
      windowSize: '1728x1200',
      fullPageScreenshots: true,
    },
    ResembleHelper : {
      require: 'codeceptjs-resemblehelper',
      screenshotFolder: './codecept/output/default/',
      baseFolder: './codecept/screenshots/base/default/'
      diffFolder: './codecept/screenshots/diff/default/'
    }
  },
  include: {
    I: './codecept/steps_file.js'
  }
```

package.json — scripts › codeceptjs:mobile

```json
    "scripts": {
      "eslint": "eslint src",
      "stylelint": "stylelint src/*.css src/_includes/*.css",
      "build": "npm run eslint && npm run stylelint && npm run build:de && npm run b
      "build:de": "eleventy --config .eleventy.de.js && node_modules/postcss-cli/bin
      "build:en": "eleventy --config .eleventy.en.js && node_modules/postcss-cli/bin
      "codeceptjs": "npm run codeceptjs:default && npm run codeceptjs:mobile",
      "codeceptjs:default": "codeceptjs run --steps",
      "codeceptjs:mobile": "MOBILE=true codeceptjs run --steps",
      "codeceptjs:headless": "HEADLESS=true codeceptjs run --steps",
      "codeceptjs:ui": "codecept-ui --app",
```

codecept.conf.js

```js
  exports.config.helpers.ResembleHelper.diffFolder
    './codecept/screenshots/diff/mobile/';
}
```

I had already defined different setups for manual vs. continuous integration test runs, to test another URL, and to set headless mode only when running in a non-graphical shell like on GitHub or Netlify. Although I didn't actually set up any CI test step yet, I split the configuration to introduce variations based on environment variables.

Environment Variables to Control Test Variations

My common base codecept configuration is in `codecept.common.conf.js`, defining default test file name patterns and a default window size:

```
module.exports = {
  tests: './codecept/*_test.js',
  output: './codecept/output/default/',
  helpers: {
    Puppeteer: {
      url: '',
      show: true,
      windowSize: '1728x1200',
      fullPageScreenshots: true,
```

The configuration for a manual test on my local computer includes the common configuration an alters some settings based on environment variables.

Inspired by the `setHeadlessWhen` hook, I added a conditional switch to a mobile configuration using its own screen size and screenshot folders.

```
const { setHeadlessWhen, setWindowSize } = require('@codeceptjs/configure');
exports.config = require ('./codecept.common.conf.js');
exports.config.helpers.Puppeteer.url = 'http://localhost:1974'; // or 'https:/

// turn on headless mode when running with HEADLESS=true environment variable
// export HEADLESS=true && npx codeceptjs run
setHeadlessWhen(process.env.HEADLESS);

if (process.env.MOBILE) {
  exports.config.output = './codecept/output/mobile/';
  exports.config.helpers.Puppeteer.windowSize = '390x844';
  exports.config.helpers.ResembleHelper.screenshotFolder = './codecept/output/r
  exports.config.helpers.ResembleHelper.baseFolder = './codecept/screenshots/ba
  exports.config.helpers.ResembleHelper.diffFolder = './codecept/screenshots/d
}
```

Now I can add more test targets in `package.json`:

```
"codeceptjs:default": "codeceptjs run --steps",
"codeceptjs:mobile": "MOBILE=true codeceptjs run --steps",
"codeceptjs:headless": "HEADLESS=true codeceptjs run --steps",
```

and a default `codeceptjs` target that runs the default and mobile tests:

```
"codeceptjs": "npm run codeceptjs:default && npm run codeceptjs:mobile"
```

Now I can use the same scenario file (`homepage_test.js`) to test different viewport sizes, generating and comparing different full page screenshots.

We can repeat the process and save different screenshots in different scenarios or in different situations, like before and after opening a mobile burger menu.

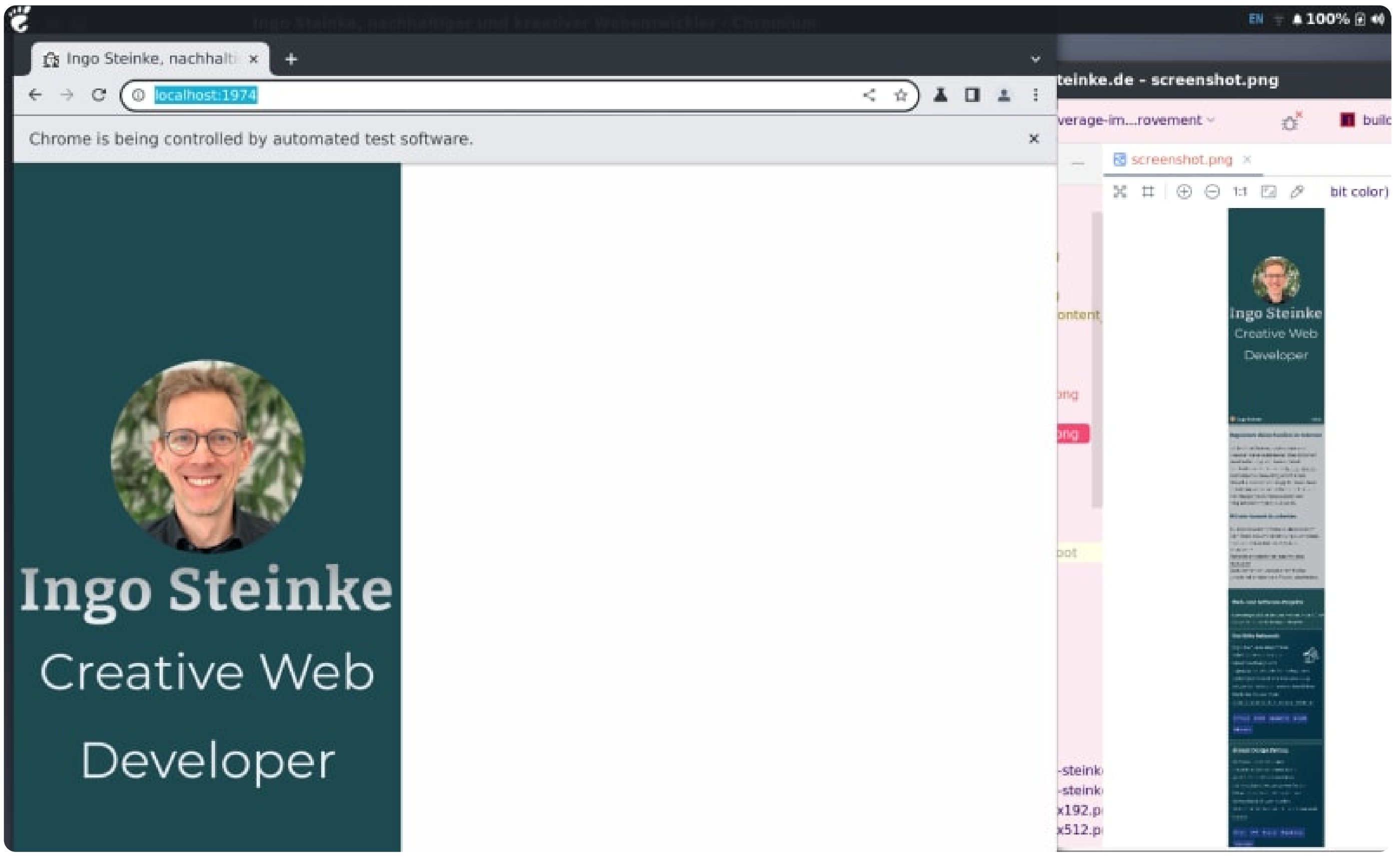
Prepare and Compare

Don't forget to toggle the configuration parameters that indicates whether to compare or to regenerate (or "prepare") the base screenshot images in our scenario file:

```
  I.saveScreenshot('screenshot.png');
  I.seeVisualDiff('screenshot.png', {
    tolerance: 2,
    prepareBaseImage: false // true: overwrite; false: compare
  });
```

Note that setting `prepareBaseImage: true` will save the latest screenshots as reference base images, even if other tests fail.

Running `npm run codeceptjs` , I can watch the test suite open actual browsers (as I did not choose the headless mode), and that the smaller viewport looks a bit out of place inside the larger browser window. But in the end, it runs my test scenario and saves the expected full-size screenshot.



Comparing the Output

The post title might be somehow misleading, if you thought that I wanted to compare the screenshots of one device with the screenshots of another one. That might be a challenge if we wanted to ensure a pixel-perfect layout looks exactly the same on Windows Edge and MacBook Safari. I don't know how that could be automated on a single machine, and I am not a fan of pixel-perfect cross-device web design anyway.

I don't want to compare a "mobile" width screenshot with a desktop screenshot either. That would be much more of a challenge in a conceptual way, and we should better use behavioral testing to make sure that the content is completely accessible on any device.

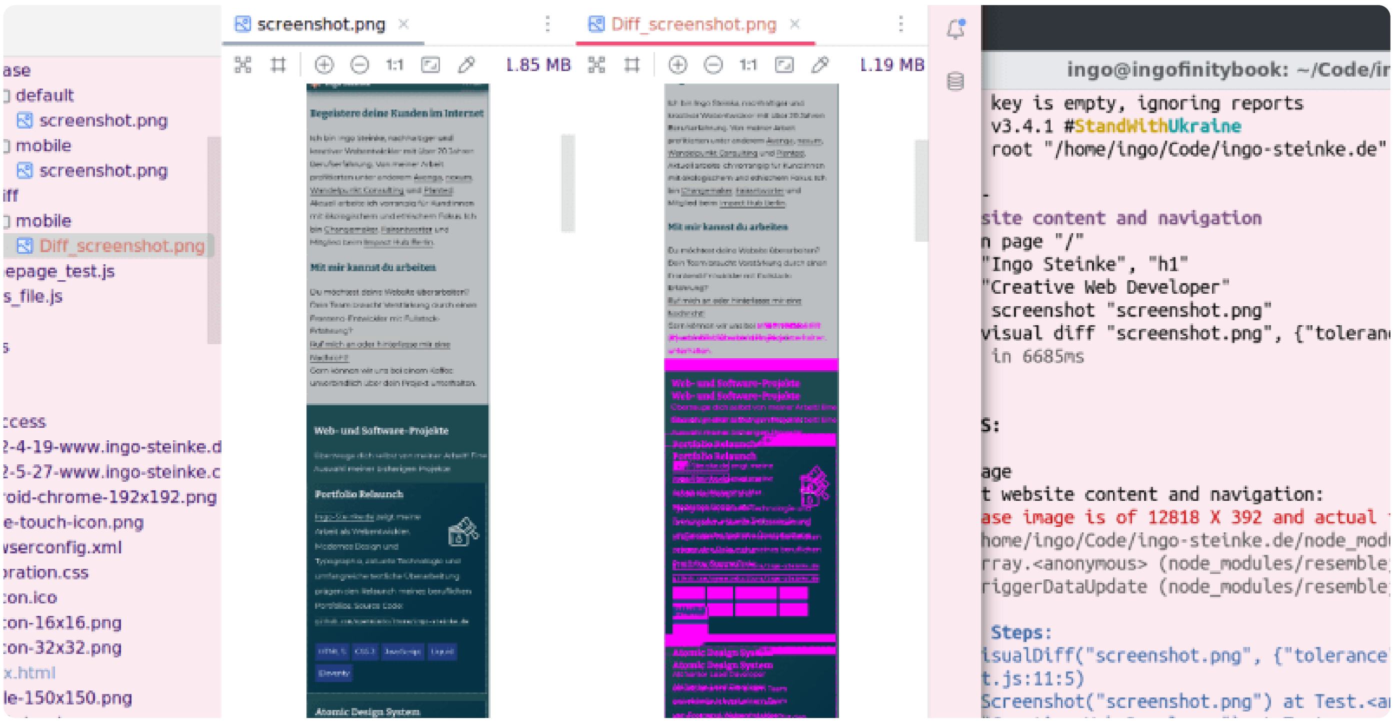
What I actually do is quite straightforward: take one screenshot of a correct output, according to manual testing, and use that as a reference for upcoming tests. So I can prove that my website still looks exactly the same in a given viewport when repeating the test in the future.

We might change nothing except for the browser version, we might update our project dependencies according to the semver specifications in our `package.json`, or we can refactor the source code and rename our CSS classes, which is what I plan to do to improve code quality and maintainability of my website. So watch out for future posts in this series...

But what does it look like if our screenshots have changed, and what if they haven't?

Evaluating Visual Test Results

If everything is fine, the tests are green and state "success" in the console. Otherwise, we see a notification that they "failed", and there will be a `diff` image showing the visual difference in form of a differential overlay.


screenshot.png
1:1
1.85 MB
Diff_screenshot.png
1.19 MB
ingo@ingofinitybook: ~/Code/ir
Begeistere deine Kunden im Internet
Mit mir kannst du arbeiten
Web- und Software-Projekte
Portfolio Relaunch
Atomic Design System

```
ase
] default
  screenshot.png
] mobile
  screenshot.png
iff
] mobile
  Diff_screenshot.png
epage_test.js
s_file.js

s

ccess
2-4-19-www.ingo-steinke.d
2-5-27-www.ingo-steinke.c
roid-chrome-192x192.png
e-touch-icon.png
wserconfig.xml
pration.css
con.ico
con-16x16.png
con-32x32.png
x.html
le-150x150.png
```

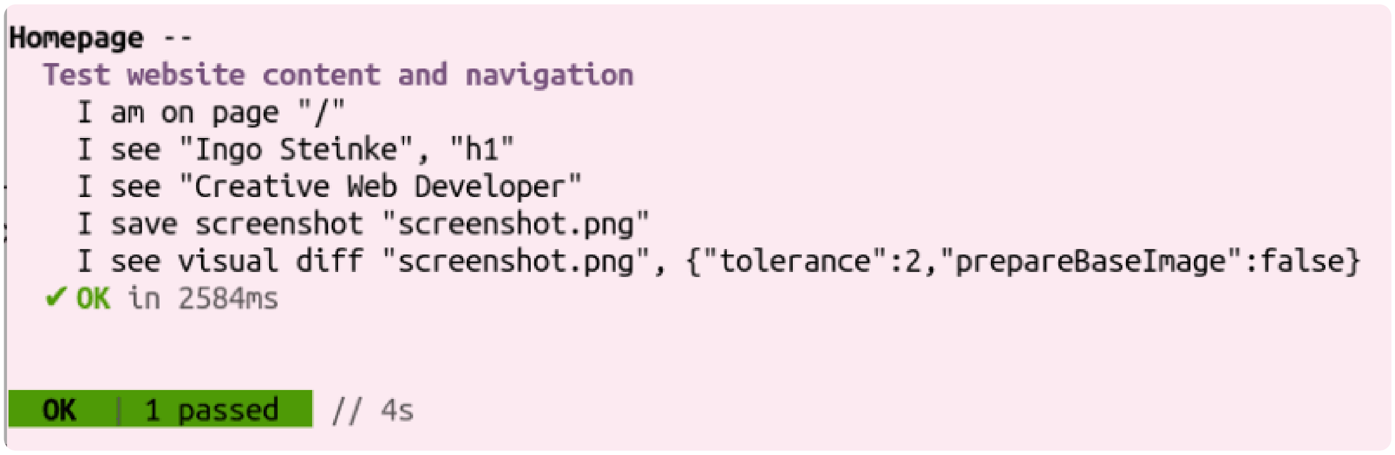
```
 key is empty, ignoring reports
 v3.4.1 #StandWithUkraine
 root "/home/ingo/Code/ingo-steinke.de"

-
site content and navigation
n page "/"
"Ingo Steinke", "h1"
"Creative Web Developer"
 screenshot "screenshot.png"
visual diff "screenshot.png", {"toleran
 in 6685ms


S:

age
t website content and navigation:
ase image is of 12818 X 392 and actual
home/ingo/Code/ingo-steinke.de/node_mod
rray.<anonymous> (node_modules/resemble
riggerDataUpdate (node_modules/resemble

 Steps:
isualDiff("screenshot.png", {"tolerance
t.js:11:5)
Screenshot("screenshot.png") at Test.<a
```

This is a typical example of a consecutive fault. While a quantitative would indicate an enormous divergence between the two screenshots, it should be obvious to the human beholder, that a small text change, increasing the paragraph's height by one line, shifts all content below by that line height. This might mask more subtle changes in the shifted content, so we have to take a very careful look.

There might be other causes shifting our content, either an intentional change of content or layout, or some sub-pixel rounding display issues, which might be the reason that there is a tolerance parameter that I set to 2 pixels when I first introduced visual regression tests in my project.

But we don't have to rely on the screenshots alone for our evaluation, and we might even revert the textual change temporarily to have a second look, before deciding if everything is fine and intended or if we need further investigation and bug fixes.

If everything is correct and the screenshot matches the reference image, we will see a success message stating that our tests have passed.



```
Homepage --
  Test website content and navigation
    I am on page "/"
    I see "Ingo Steinke", "h1"
    I see "Creative Web Developer"
    I save screenshot "screenshot.png"
    I see visual diff "screenshot.png", {"tolerance":2,"prepareBaseImage":false}
  ✔ OK in 2584ms


 OK  | 1 passed   // 4s
```

Ingo Steinke • May 9

Maybe I should add an other tutorial showing how to achieve the same result in Cypress, or ad some controversial claim in the headline? Or why do the most useful posts – at least from my own perspective – get the fewest likes an bookmarks?