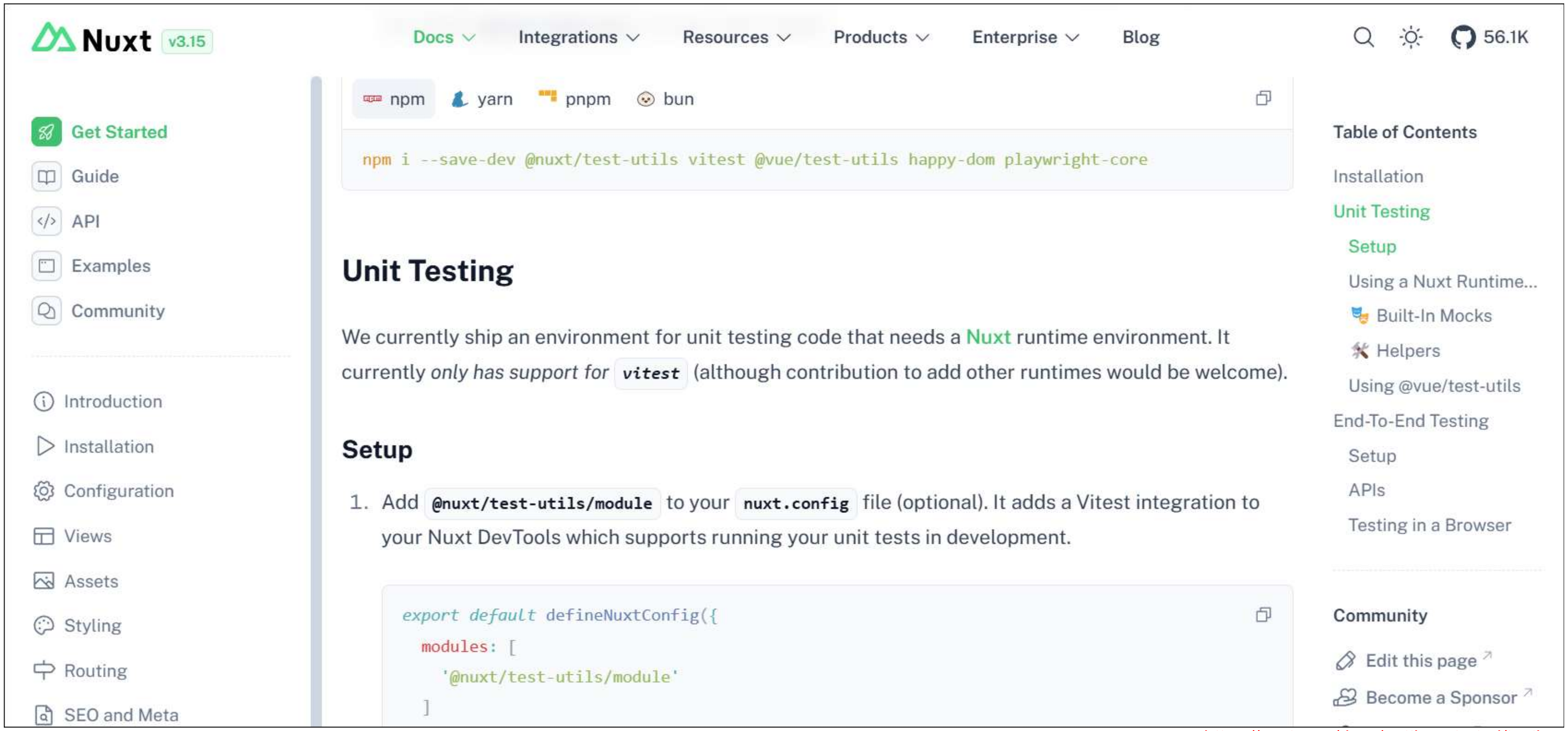
Nuxt Fundamentals

@nuxt/test-utils

Peter Kassenaar –
info@kassenaar.com

Using Nuxt-specific test utils



```
npm i --save-dev @nuxt/test-utils vitest @vue/test-utils happy-dom playwright-core
```

Unit Testing

We currently ship an environment for unit testing code that needs a Nuxt runtime environment. It currently *only has support for* `vitest` (although contribution to add other runtimes would be welcome).

Setup

1. Add `@nuxt/test-utils/module` to your `nuxt.config` file (optional). It adds a Vitest integration to your Nuxt DevTools which supports running your unit tests in development.

```
export default defineNuxtConfig({
  modules: [
    '@nuxt/test-utils/module'
  ]
```

https://nuxt.com/docs/getting-started/testing

Nuxt specific testing looks A LOT like 'just' Vue testing with Vitest.

So, master that first!

There are some extras, though – Borders are often times unclear (IMO)!

Extra dependencies

```
npm i --save-dev @nuxt/test-utils vitest @vue/test-utils
happy-dom playwright-core
```

- `@nuxt/test-utils` for nuxt runtime environment
- `vitest` as the test runner
- `@vue/test-utils` for Vue testing
- `happy-dom` as a DOM library for Node
- `playwright` as solution for e2e-tests

```json
{
  "...",
  "devDependencies": {
    "@nuxt/test-utils": "^3.15.4",
    "@vue/test-utils": "^2.4.6",
    "happy-dom": "^16.8.1",
    "playwright-core": "^1.50.1",
    "vitest": "^3.0.5"
  }
}
```

package.json

Update files

1. Update `nuxt.config.ts` to add modules

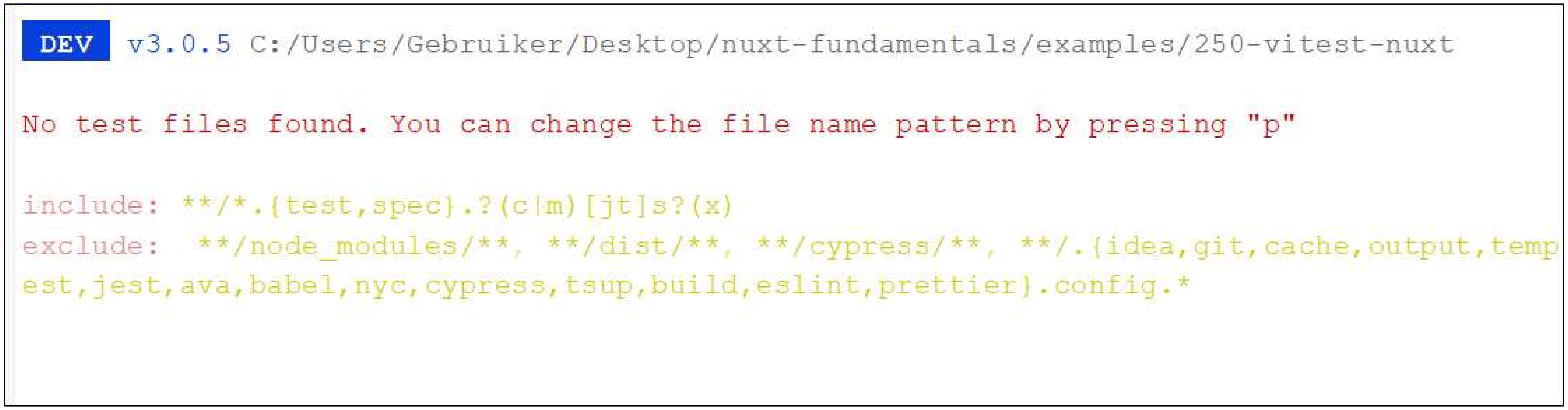
```
// nuxt.config.ts
// https://nuxt.com/docs/api/configuration/nuxt-config
export default defineNuxtConfig({
  compatibilityDate: '2024-11-01',
  devtools: { enabled: true },
  modules:[
      '@nuxt/test-utils/module'
  ]
})
```

2. Update `vitest.config.ts` to use nuxt as testing environment

```
// vitest.config.ts
import {defineVitestConfig} from '@nuxt/test-utils/config'

export default defineVitestConfig({
    test: {
        environment: 'nuxt',
    }
    // other custom configuration required...
})
```

See if it all works

- Add script to package.json `“test”: “vitest”`
- `Npm run test`
- Tests should fail, b/c `No test files found`.
  - Which, of course, is correct
  - But at least, everything works!
- We can start writing tests now



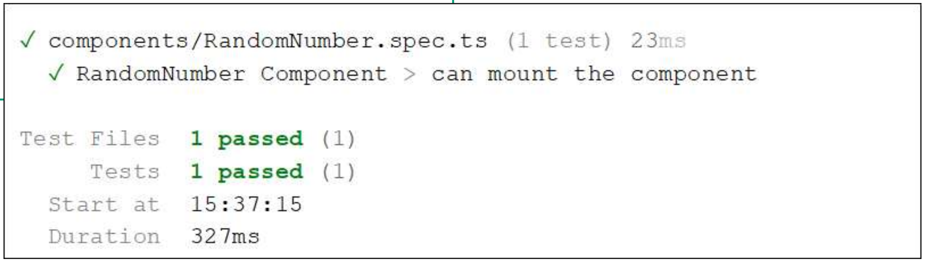
```
 DEV  v3.0.5 C:/Users/Gebruiker/Desktop/nuxt-fundamentals/examples/250-vitest-nuxt

No test files found. You can change the file name pattern by pressing "p"

include: **/*.{test,spec}.?(c|m)[jt]s?(x)
exclude:  **/node_modules/**, **/dist/**, **/cypress/**, **/.{idea,git,cache,output,temp
est,jest,ava,babel,nyc,cypress,tsup,build,eslint,prettier}.config.*
```

1. Test if the component can mount

- Using `mountSuspended()`
  - This is a Nuxt-alternative to `mount()`
  - *"Mount a Vue component to a DOM element, but do NOT render de component immediately"*
- Suspend the rendering until explicitly resumed
  - For instance: wait until fetching async data is complete
- Not really necessary in our case, but as a precaution
  - `mountSuspended()` gives you more control over the rendering process

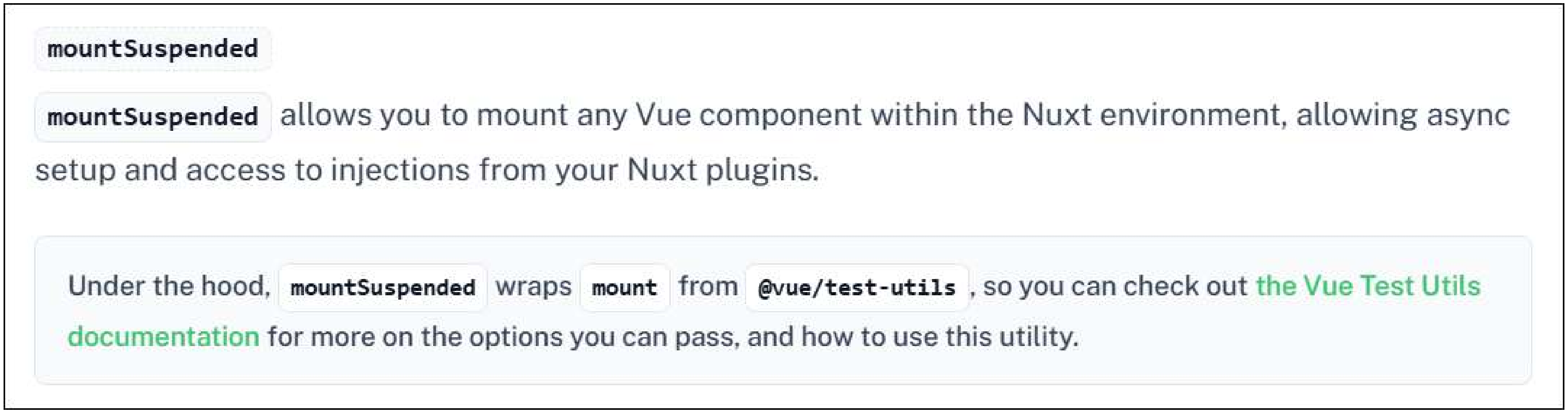
```
it('can mount the component', async () => {
    const component = await mountSuspended(RandomNumber)
    expect(component).toBeTruthy();
})
```

randomNumber.spec.ts



```
✓ components/RandomNumber.spec.ts (1 test) 23ms
  ✓ RandomNumber Component > can mount the component

Test Files  1 passed (1)
     Tests  1 passed (1)
  Start at  15:37:15
  Duration  327ms
```

https://nuxt.com/docs/getting-started/testing


`mountSuspended`

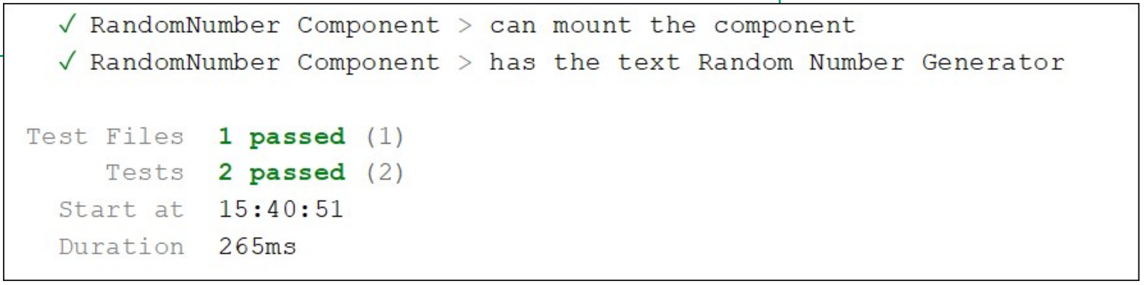
`mountSuspended` allows you to mount any Vue component within the Nuxt environment, allowing async setup and access to injections from your Nuxt plugins.

Under the hood, `mountSuspended` wraps `mount` from `@vue/test-utils`, so you can check out the Vue Test Utils documentation for more on the options you can pass, and how to use this utility.

2. Test if component is correctly rendered

- The function `mountSuspended()` *will* render the component, but it waits for async operations to complete, before doing so.
- So, we can test if `text` or `html` is available in the DOM
- Note: no `vm.wrapper` necessary

```
// 2. Test if the HTML is correctly rendered
it('has the text Random Number Generator', async () => {
    const component = await mountSuspended(RandomNumber)
    expect(component.html()).toContain('Random number generator');
})
```



```
  ✓ RandomNumber Component > can mount the component
  ✓ RandomNumber Component > has the text Random Number Generator

Test Files  1 passed (1)
     Tests  2 passed (2)
  Start at  15:40:51
  Duration  265ms
```

`randomNumber.spec.ts`

3. Test if the state is correctly used

- Test if the calculated number / state is correctly used
- We mock the `useState` function, b/c the calculated number is different every time
- Use the `mockNuxtImport()` macro for that
  - Can be used only once (1x!) in a file

```
// Helper function to mock useState
mockNuxtImport('useState', () => {
    return () => 20000;
})
```

https://nuxt.com/docs/getting-started/testing#mocknuxtimport

```
// 3. Test if the state is correctly used.
it('returns the state', async () => {
    const component = await mountSuspended(RandomNumber)
    console.log(component.text());
    expect(component.text()).toContain('20000')
})
```

4. Using a spy

- We want to make sure that clicking the `Refresh` button actually calls the `refresh()` function
- In traditional situations you use a *spy* for that
- Spies are available on the `vi` object, imported from `vitest`:

```
// NOTE: will NOT work in Vue 3 <script setup> blocks!!
it('should call the refresh function when the refresh button is clicked', async () => {
  const wrapper = await mountSuspended(RandomNumber)

  // Create a spy on the 'refresh' function
  const refreshSpy = vi.spyOn(wrapper.vm, 'refresh’)

  // Simulate click on the refresh button
  await wrapper.find('button').trigger('click’)

  // Verify that the refresh function was called
  expect(refreshSpy).toHaveBeenCalled()
})
```

Use .spyOn() in Options API

Spies in <script setup> blocks

- In `<script setup>` blocks, internals are NOT exposed on the wrapper/vm.
  - *"Because of the Vue 3 Composition API `refresh()` is inaccessible as a method on `component.vm`"*
  - Test is failing.
- We therefore need to test on *outcome*, instead of directly spying on a method
  - *"Instead of directly testing if `refresh()` is called, the test verifies the outcome of clicking the button, which results in changing the state and updating the displayed random number.*
  - *This approach aligns well with the philosophy of Vue 3's Composition API and how `<script setup>` encapsulates methods and data."*

Therefore, Composition API test like:

```
// 4. Test if the refresh() function is called when button is clicked
it('should call the refresh function when button is clicked', async () => {
    // Mount the component using `mountSuspended`
    const component = await mountSuspended(RandomNumber)

    // Access the text element displaying the random number
    const numberBefore = component.find('h3').text()

    // Simulate click on the refresh button
    await component.find('button').trigger('click')

    // Access the text element displaying the random number again
    const numberAfter = component.find('h3').text()

    // Verify that the number displayed is updated
    expect(numberBefore).not.toBe(numberAfter)
})
```

Expect result to be different

When are traditional spies still used?

- Spies in Vue 3 are mostly used *outside* components
- For instance on libraries or utility functions

```
// do something like:
import { someFunction } from "@/utils";

vi.spyOn(someFunction);
someFunction();
expect(someFunction).toHaveBeenCalled();
```

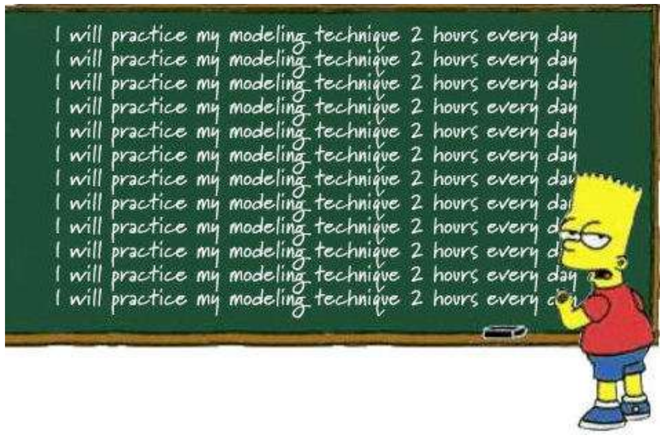
Mocking `Math.random`

- We can also mock `Math.random` and simulate clicks.
  - This will ensure a new random number is generated and shown in the UI
  - This is a combination of techniques
- use `vi.spyOn` to spy on `Math.random` and replace it with a mock using `.mockImplementation()`.
  - This gives us access to mock-specific methods such as `mockReturnValueOnce`.
- After the test, call `randomSpy.mockRestore()` to restore the original `Math.random` implementation
  - To ensure: *no interference* with other tests.

```
it('should generate a new random number on refresh', async () => {

    // Spy on Math.random() and mock it's implementation
    const randomSpy = vi.spyOn(Math, 'random')
        .mockImplementation(() => 0);

    // Mock the first random number
    randomSpy.mockReturnValueOnce(0.5); // First random number
    const component = await mountSuspended(RandomNumber);

    // Simulate click on the refresh button
    await component.find('button').trigger('click')

    // Initial rendering of the random number
    const initialNum = parseInt(component.find('h3').text());
    expect(initialNum).toBe(50000); // 0.5 * 100000

    // Mock Math.random() for subsequent numbers
    randomSpy.mockReturnValueOnce(0.8); // Second random number
    await component.find('button').trigger('click'); // Simulate click event
    const numAfterFirstClick = parseInt(component.find('h3').text());
    expect(numAfterFirstClick).toBe(80000); // 0.8 * 100000


    …
    // Clean up by restoring the original implementation
    randomSpy.mockRestore();
})
```

Workshop #2

- Add Nuxt3-specific nuxt test-utils and see presentation on examples of tests
- Add tests to a component in your own application



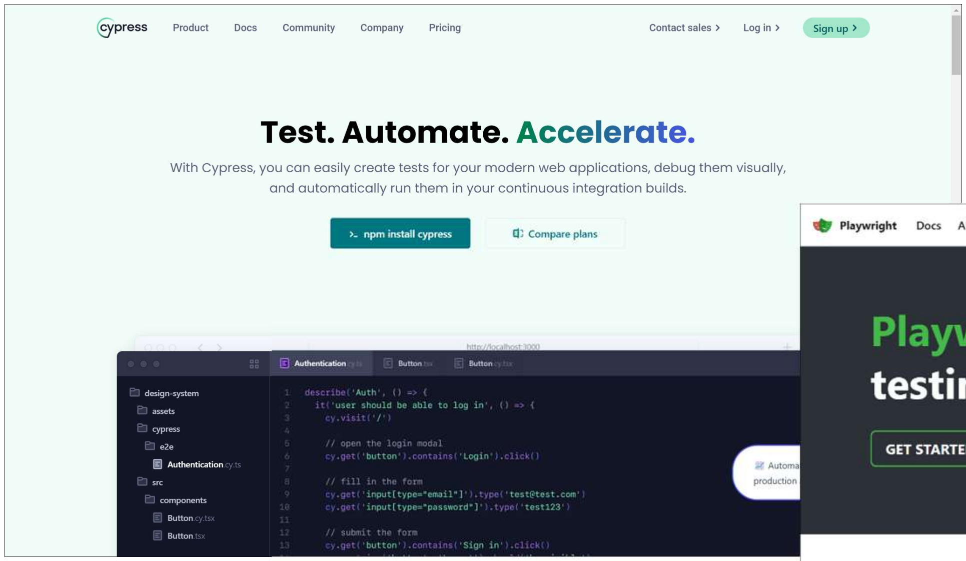
End-2-end testing

Using Playwright for your e2e-tests

End-to-end (e2e)-tests

- Spin up an actual server and (headless) browser
- Load the complete application in the test
- Navigate to pages, interact with application via testing code
- Also called *Scenario Testing*
- Testers write scenario's like:
  - *"Open the browser, navigate to the login page, Click the sign-up button, fill in username and password, click Submit, then expect the application to register you as a new user."*
  - In other words: you test if your combination of pages and components work as intended.
- You DON'T have to test component functionality – this is already done in unit tests!

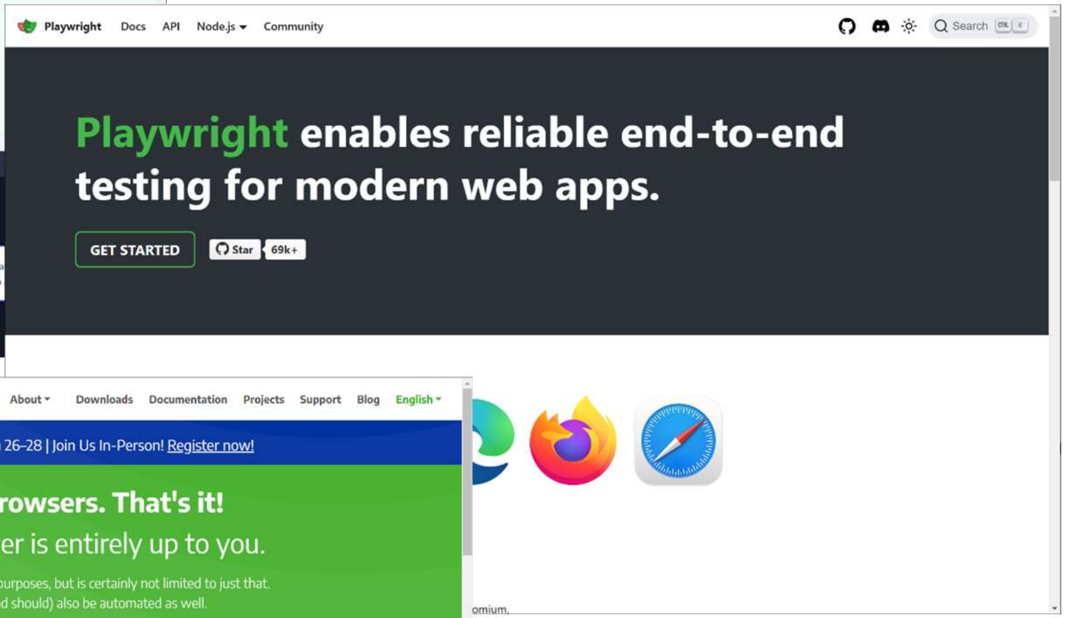
E2e-tooling



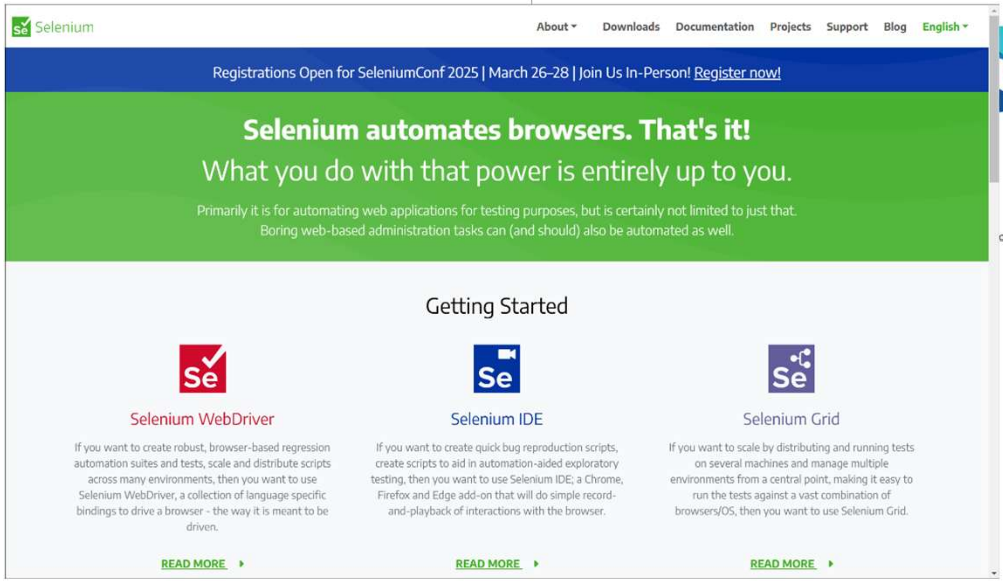
Test. Automate. Accelerate.

With Cypress, you can easily create tests for your modern web applications, debug them visually, and automatically run them in your continuous integration builds.

```
describe('Auth', () => {
  it('user should be able to log in', () => {
    cy.visit('/')

    // open the login modal
    cy.get('button').contains('Login').click()

    // fill in the form
    cy.get('input[type="email"]').type('test@test.com')
    cy.get('input[type="password"]').type('test123')

    // submit the form
    cy.get('button').contains('Sign in').click()
```



Playwright enables reliable end-to-end testing for modern web apps.



Registrations Open for SeleniumConf 2025 | March 26-28 | Join Us In-Person! Register now!

Selenium automates browsers. That's it!

What you do with that power is entirely up to you.

Primarily it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should) also be automated as well.

Getting Started

Selenium WebDriver

If you want to create robust, browser-based regression automation suites and tests, scale and distribute scripts across many environments, then you want to use Selenium WebDriver, a collection of language specific bindings to drive a browser - the way it is meant to be driven.

Selenium IDE

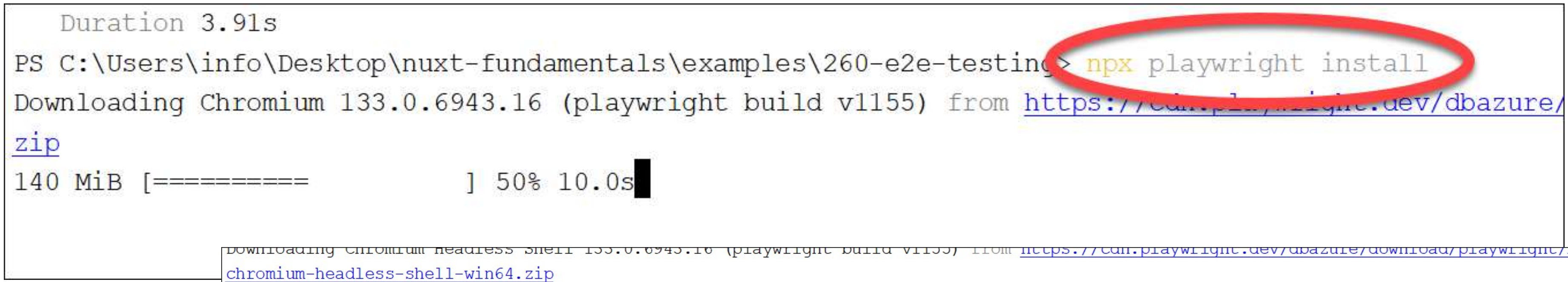
If you want to create quick bug reproduction scripts, create scripts to aid in automation-aided exploratory testing, then you want to use Selenium IDE; a Chrome, Firefox and Edge add-on that will do simple record-and-playback of interactions with the browser.

Selenium Grid

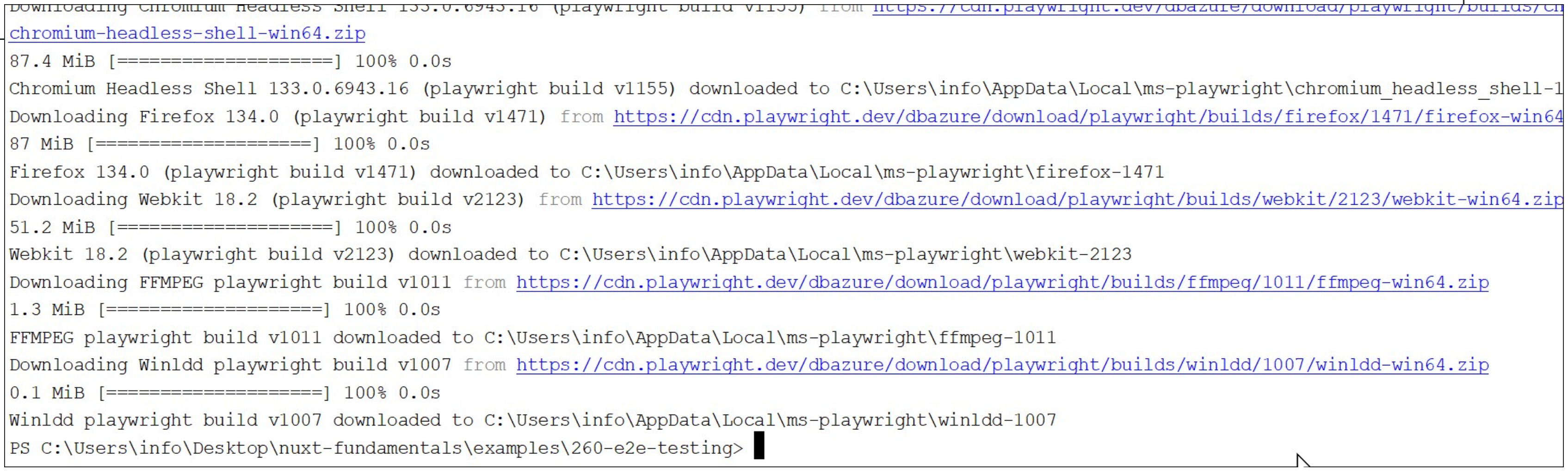
If you want to scale by distributing and running tests on several machines and manage multiple environments from a central point, making it easy to run the tests against a vast combination of browsers/OS, then you want to use Selenium Grid.

Install playwright globally

(=headless chromium)



```
  Duration 3.91s
PS C:\Users\info\Desktop\nuxt-fundamentals\examples\260-e2e-testing> npx playwright install
Downloading Chromium 133.0.6943.16 (playwright build v1155) from https://cdn.playwright.dev/dbazure/
zip
140 MiB [==========          ] 50% 10.0s
```



```
Downloading Chromium Headless Shell 133.0.6943.16 (playwright build v1155) from https://cdn.playwright.dev/dbazure/download/playwright/builds/ch
chromium-headless-shell-win64.zip
87.4 MiB [====================] 100% 0.0s
Chromium Headless Shell 133.0.6943.16 (playwright build v1155) downloaded to C:\Users\info\AppData\Local\ms-playwright\chromium_headless_shell-1
Downloading Firefox 134.0 (playwright build v1471) from https://cdn.playwright.dev/dbazure/download/playwright/builds/firefox/1471/firefox-win64
87 MiB [====================] 100% 0.0s
Firefox 134.0 (playwright build v1471) downloaded to C:\Users\info\AppData\Local\ms-playwright\firefox-1471
Downloading Webkit 18.2 (playwright build v2123) from https://cdn.playwright.dev/dbazure/download/playwright/builds/webkit/2123/webkit-win64.zip
51.2 MiB [====================] 100% 0.0s
Webkit 18.2 (playwright build v2123) downloaded to C:\Users\info\AppData\Local\ms-playwright\webkit-2123
Downloading FFMPEG playwright build v1011 from https://cdn.playwright.dev/dbazure/download/playwright/builds/ffmpeg/1011/ffmpeg-win64.zip
1.3 MiB [====================] 100% 0.0s
FFMPEG playwright build v1011 downloaded to C:\Users\info\AppData\Local\ms-playwright\ffmpeg-1011
Downloading Winldd playwright build v1007 from https://cdn.playwright.dev/dbazure/download/playwright/builds/winldd/1007/winldd-win64.zip
0.1 MiB [====================] 100% 0.0s
Winldd playwright build v1007 downloaded to C:\Users\info\AppData\Local\ms-playwright\winldd-1007
PS C:\Users\info\Desktop\nuxt-fundamentals\examples\260-e2e-testing>
```

Headless browsers installed

E2e testing? Import the right package!

- Import the e2e-packages to use end-to-end testing!
  - `import {setup, $fetch, createPage, url} from "@nuxt/test-utils/e2e‟`

```
import {describe, it, expect} from "vitest";
// NOTE: import from 'e2e' module!
import {setup, $fetch, createPage, url} from "@nuxt/test-utils/e2e";

describe('Complete App, e2e', async () => {
    await setup() // await the setting up of the complete nuxt application

    // Option 1: using Vitest
    it('…', async () => {
        …
    });

    // Option 2: with playwright
    it('…', async () => {
      …
    })
})
```

1. Using Vitest

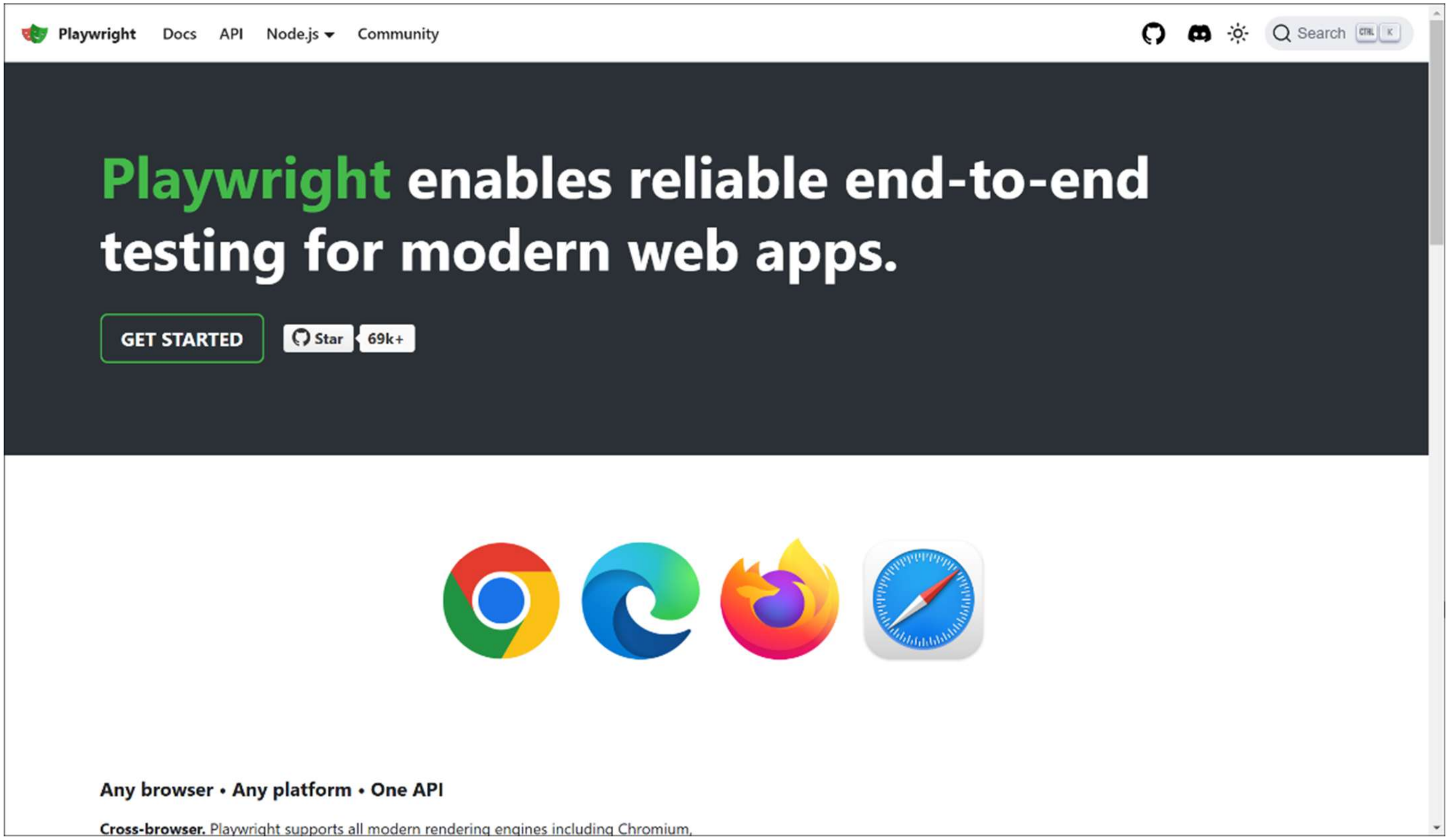
- You CAN simply use Vitest for e2e-testing,
  - but this way you can’t really interact with the generated DOM.
- You can however, for instance, test if the page is rendered correctly

```
// Option 1: using Vitest
it('1. contains text as a string (Vitest)', async () => {
    const html = await $fetch('/');// fetch the page, from e2e test-utils(!).
    expect(html).contains('Random number generator')
});
```

2. Using Playwright

- Playwright creates a DOM that you can query and interact with.
- More advantages (though Cypress and Selenium offer this also, mostly)
    - Parallel execution (Tests run in multiple browsers at the same time)
    - Headless mode by default: Faster than UI-based testing.
    - Auto-waiting: No need for `waitForSelector()`, as Playwright waits for elements to be ready.
    - Cross-Browser & Cross-Platform - Supports Chromium, Firefox, WebKit (Safari) and Edge.Allows mobile emulation (iOS & Android).
    - Can run tests on Windows, macOS, and Linux without extra setup.
    - SSR & SPA-Friendly
    - Network & API Testing - Intercept network requests to mock/stub API responses.
    - …and more…

https://playwright.dev/



Playwright enables reliable end-to-end testing for modern web apps.

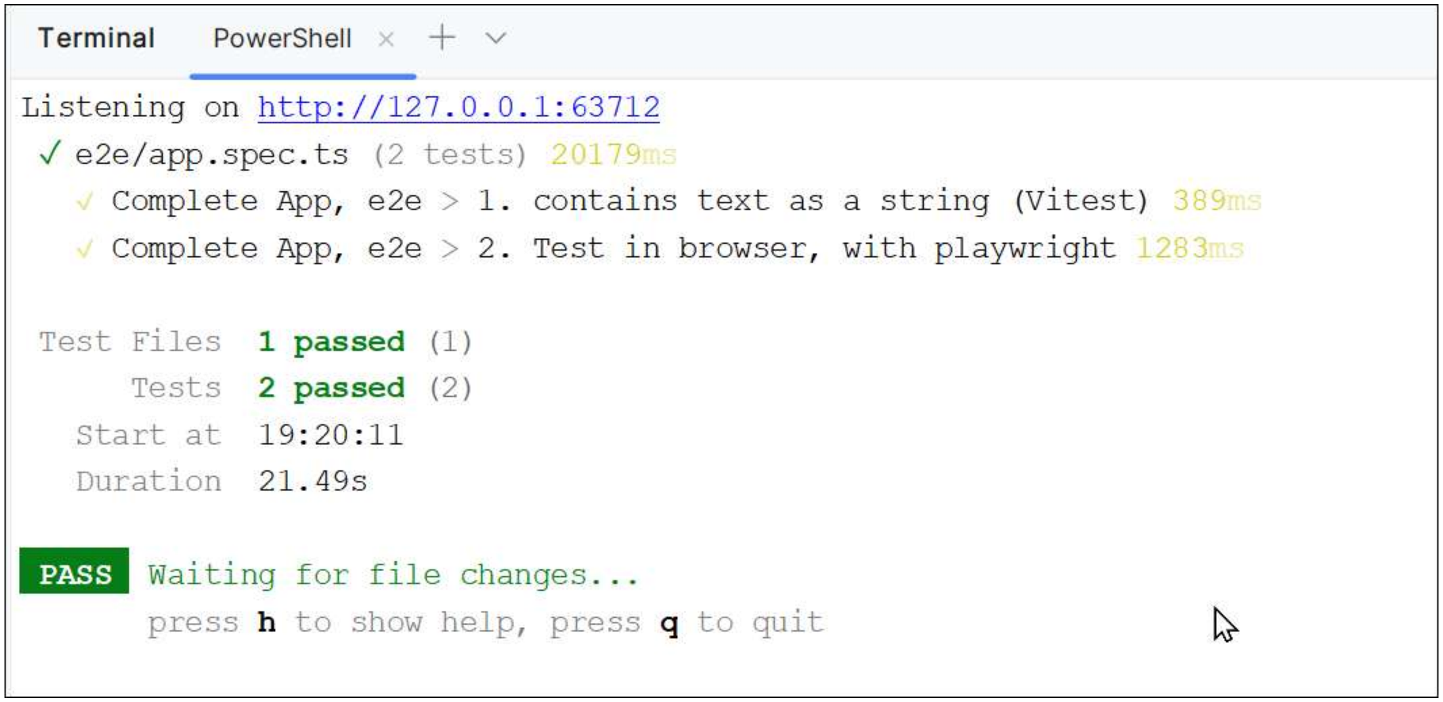
Any browser • Any platform • One API

Cross-browser. Playwright supports all modern rendering engines including Chromium,

Sample playwright test

```
// Option 2: with playwright
it('2. Test in browser, with playwright', async () => {
    // 1. create the page (imported from test-utils/e2e)
    const page = await createPage();
    // 2. go to the root, wait until page is fully hydrated
    await page.goto(url('/'), {
        waitUntil: 'hydration'
    });
    // 3. get the generated number from the page. It lives inside an <h3>,
    // therefore we use that selector.
    const text = await page.textContent('h3');
    // 4. casting
    const number = Number(text);
    // 5. expectation
    expect(number).toBeGreaterThan(0);

    // 6. Let's interact with the page.
    // We now expect a new number, which is different from the previous number
    await page.click('button');
    const newText = await page.textContent('h3');
    const newNumber = Number(newText);
    expect(number).not.toBe(newNumber);
})
```

Sample output



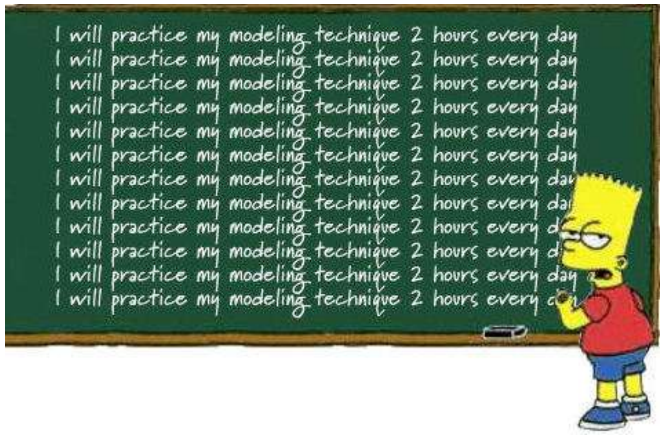
```
Listening on http://127.0.0.1:63712
 ✓ e2e/app.spec.ts (2 tests) 20179ms
   √ Complete App, e2e > 1. contains text as a string (Vitest) 389ms
   √ Complete App, e2e > 2. Test in browser, with playwright 1283ms

 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  19:20:11
   Duration  21.49s

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Note: e2e-testing takes *considerably longer*, because the web server and browser have to start in-memory.

A fast computer definitely helps!

Workshop e2e

- Create a page in your app that you can navigate to, for instance `./pages/about.vue`
- Make sure the page is composed of multiple components.
- Install the correct playwright dependencies
- Create an e2e-folder and write an `./e2e/about.spec.ts` that tests:
  - If the page can be navigated to
  - If the page has the correct header
  - If the page contains the stuff that you included
- Update the test to navigate from the homepage to the `/about` page and see if everthing still works
  - Tip: create a link, or `MainNavigation` on homepage
- Read the playwright-docs to use it directly
  - https://playwright.dev/docs/intro



Checkpoint

- You know what generic types of tests are available
- You are able to identify testing files in a project
- You can create both unit tests and e2e-tests
- You can test the basic behavior of a component
- You know which dependencies to install in your project to enable testing
- You what to include in the imports of the page depending on your tests.
- When using AI, always include the *framework*, *version number* and *API* used in your prompt!