UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 24

WEB TESTING

Luigi Libero Lucio Starace, PhD

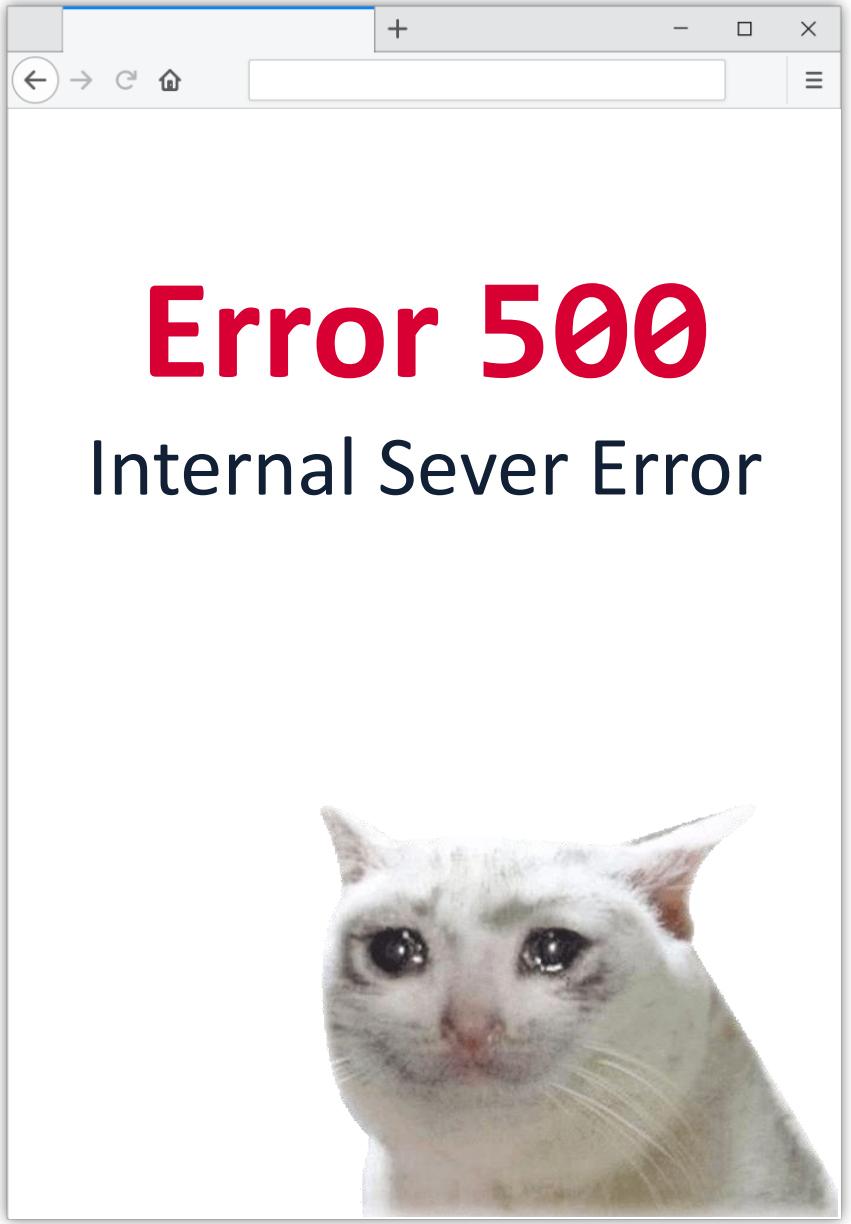
luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

HIGH QUALITY WEB APPS

- We have learned to design and implement modern, secure web apps
- One piece missing before we can actually deliver **high quality** apps
- Guess what?
  - **Testing**!
- Testing is fundamental to catch as many **bugs** as possible before they reach production (where they can cost quite a lot of **$$$**!)



SOFTWARE VERIFICATION (REDUX)

Practices aimed at ensuring that a software satisfies a specification

- **Dynamic Verification** (involves program execution)
  - **Software Testing**
- **Static Verification** (does not involve program execution)
  - Code Review
  - Static Analysis (e.g.: Taint Analysis)
  - Automated Verification (model checking)
- The goal is to catch defects or bugs before the product is released

SOFTWARE TESTING (REDUX)

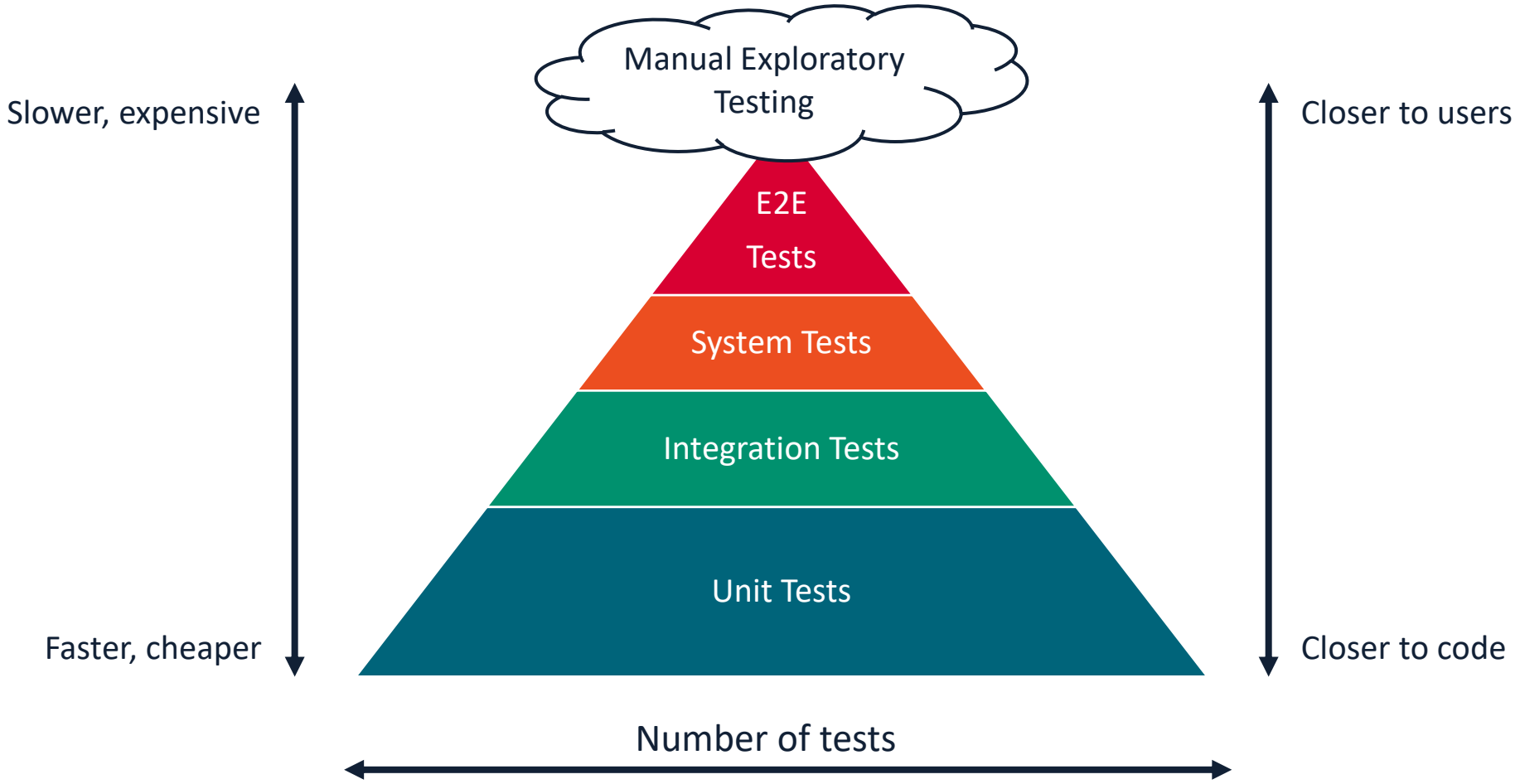
- A software test is a sequence of actions designed to evaluate a particular aspect or functionality of a software
- The **Software Under Test** is often called **SUT**
- A Test generally consists in:
  - Ensuring the required pre-conditions are fulfilled (**Arrange** phase)
  - Executing one or more actions (**Act** phase)
  - Checking that the SUT behaved as intended (**Assert** phase)

TESTING LEVELS (REDUX)

Testing can be performed at different levels:

- **Unit testing**
  - Tests a single *unit* (e.g. a class, or a method)
- **Integration testing**
  - Checking that different units work together as intended
- **System testing**
  - Targets the system as a whole, against its specification
- **End-to-End (E2E) testing**
  - Targets the SUT from the viewpoint of its intended end users

THE TESTING PYRAMID

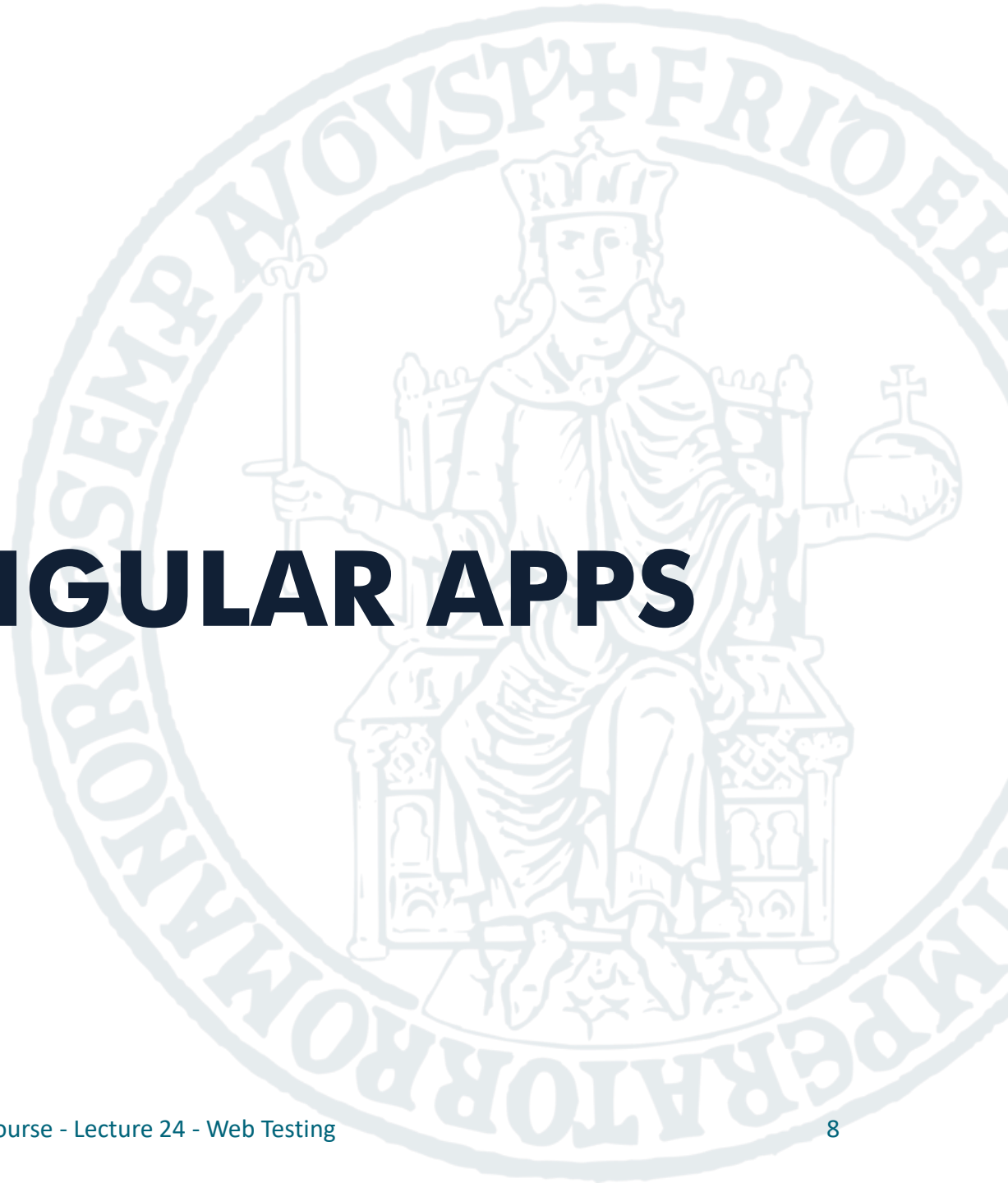


WEB APPLICATION TESTING

Web Apps, like any software, should be properly tested to ensure high quality. Some peculiar challenges arise:

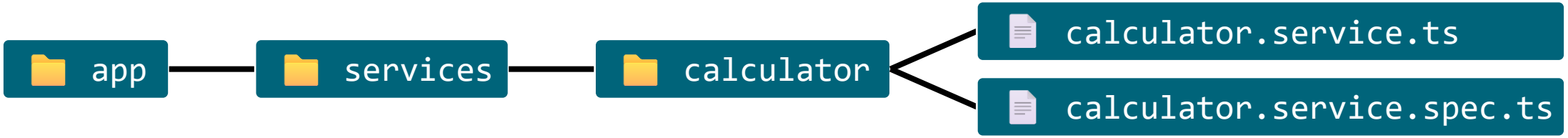
- When doing **Unit Testing**, we need to deal with:
  - Async code
  - Dependencies on external services
- When doing **End-to-End Testing**, we need to deal with:
  - **Fragility**
  - **Flakyness**

UNIT TESTING ANGULAR APPS



UNIT TESTING ANGULAR APPS

- Angular comes with all we need to test our apps using Jasmine
- By default, test code is written in **spec** (specification) **files**
  - Stub spec files are also generated by default by the Angular CLI
  - Typically placed in the same directory as the component/service they test
  - Widely adopted naming convention is to add «**.spec**» to the name of the tested component/service



JASMINE SPEC FILES

```
import { TestBed } from '@angular/core/testing';
import { CalculatorService } from './calculator.service';

describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(CalculatorService);
  });

  it('should be created', () => { expect(service).toBeTruthy(); });

  it('should sum two positive numbers correctly', () => {
    expect(service.sum(42,58)).toEqual(100);
  });
});
```

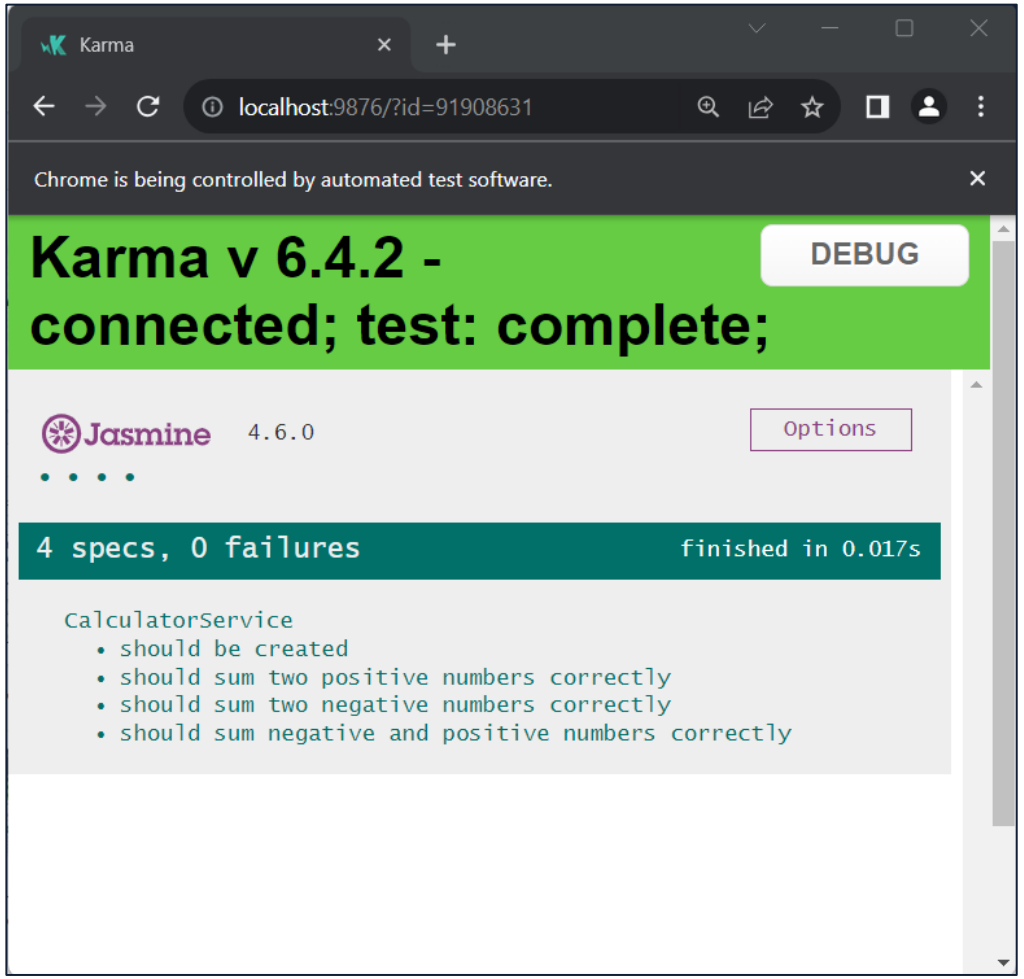
describe() creates a set (a.k.a. **suite**) of specs (or tests). Same idea as a JUnit Test Class. It groups related tests.

TestBed is the most important Angular Testing Utility. Provides methods to create components and services in unit tests.

it() defines a single spec.
Same idea as a **@Test** method in Junit.

UNIT TESTING ANGULAR APPS

- The command **ng test** build the application, and launches the Karma Test Runner
- Karma picks up any spec file and runs it
- Karma also produces a test execution report



TESTING SERVICES

```
import { TestBed } from '@angular/core/testing';
import { AuthService } from './auth.service';

describe('AuthService', () => {
  let service: AuthService;
  beforeEach(() => { service = TestBed.inject(AuthService); });

  it('should detect malformed tokens', () => {
    let malformed = "malformed";
    expect(service.verifyToken(malformed)).toBe(false);
  });

  it('should detect expired tokens', () => {
    let expired = "eyJhbGciO...c9-p0"; //expired on 2000/01/01
    expect(service.verifyToken(expired)).toBe(false);
  });
  //other tests omitted
}
```

TESTING AND DEPENDENCIES

- Typically, the services/components we test will depend on other services/components
- For example, the **RestBackend Service** depends on **HttpClient**
- The **TodoPage Component** depends on the **RestBackend Service**
- In these scenarios, unit testing becomes a little more challenging
- Suppose **TodoPage Component** is not showing the expected To-dos
  - Is it because of a bug in the TodoPage Component?
  - Is it because of a bug in the RestBackend Service?
  - Is it because of a bug in the external REST Backend?
  - Or a combination of all the above?

TESTING AND DEPENDENCIES

- This issue goes far beyond fault localization challenges
- In many cases, we may not want to use real production code units during tests
- Think of a **TemperatureService** that returns the current temperature in Naples by invoking some external REST API.
- What if we want to test what happens in our component when the temperature goes below zero?
  - Do we wait for a few years and run the test as soon as we get an exceptionally cold day?

TESTING IN ISOLATION

- Good unit tests should test a code unit in **isolation**
  - The outcome should not depend on external code units
- How can achieve this when our code units have dependecies?
- The solution is to use **Test Doubles**
- Test Doubles are replacements for production objects that are typically used in tests
- Instead of using the real **TemperatureService**, se use a **TemperatureServiceDouble** which always return a -42°C temperature!

TESTING AND DEPENDENCY INJECTION

- The Dependency Injection mechanism is a precious support when we want to use Test Doubles
- During tests, we can configure the injector to resolve dependencies using Test Doubles instead of the real deal!
- Writing Test Doubles from scratch still takes a good deal of effort
  - Lots of boilerplate code to write
- Luckily, testing frameworks can support us in creating Doubles
- Jasmine, for example, supports the concept of **Spies**
- A **Spy** is a test double that can «mock» any function

JASMINE SPIES

```
let todos = [{id: 1, todo: "foo"}, {id: 2, todo: "bar"}];
restBackendSpy = jasmine.createSpyObj('RestBackendService', ['getTodos']);
restBackendSpy.getTodos.and.returnValue(of(todos));
```

The code above:

- Creates RestBackendService Spy, mocking its **getTodos()** method
- Configures the Spy to always return an **Observable** of a given list of two To-do Items every time the **getTodos()** method is invoked
- If we use the Spy in place of the real RestBackendService, our test will not depend on the Service, nor on the (external) REST API!

TESTING THE TO-DO PAGE COMPONENT

```
it('should properly fetch to-do items', () => {
  let todos = [{id: 1, todo: "foo"}, {id: 2, todo: "bar"}];
  restBackendSpy = jasmine.createSpyObj('RestBackendService', ['getTodos']);
  restBackendSpy.getTodos.and.returnValue(of(todos));
  component.restService = restBackendSpy;

  component.fetchTodos();
  fixture.detectChanges();

  expect(component.todos.length).toEqual(2);
  const element: DebugElement = fixture.debugElement; //get <app-todo-page> elem
  const list = element.query(By.css('ul'));
  const content = list.nativeElement.textContent;
  expect(content).toContain('foo');
  expect(content).toContain('bar');
})
```

COMPUTING CODE COVERAGE

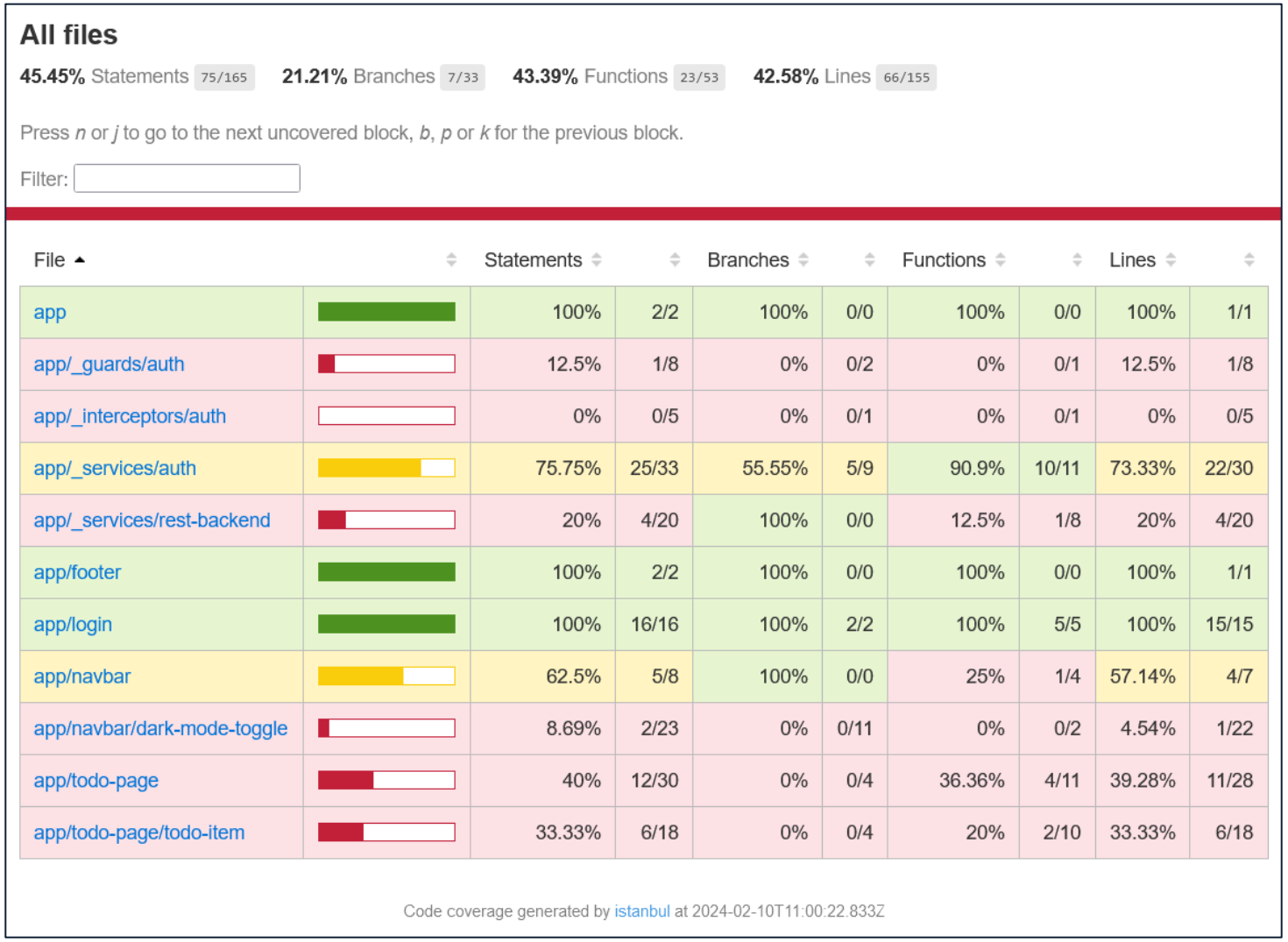
- We can also easily compute **code coverage metrics**
  - A possible indicator of how «well» we tested the app

```
@luigi ➜ D/O/T/W/2/e/2/angular-todo-list $ ng test --no-watch --code-coverage
✓ Browser application bundle generation complete.
...
Chrome 121.0.0.0 (Windows 10): Executed 18 of 18 SUCCESS
TOTAL: 18 SUCCESS

=============================== Coverage summary ===============================
Statements   : 45.45% ( 75/165 )
Branches     : 21.21% ( 7/33 )
Functions    : 43.39% ( 23/53 )
Lines        : 42.58% ( 66/155 )
================================================================================
@luigi ➜ D/O/T/W/2/e/2/angular-todo-list $
```

COMPUTING CODE COVERAGE

- An HTML coverage report is generated in the **/coverage** directory
- We can also inspect each file separately and see which lines of code were covered by tests



All files

45.45% Statements 75/165 **21.21%** Branches 7/33 **43.39%** Functions 23/53 **42.58%** Lines 66/155

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
app	100%	2/2	100%	0/0	100%	0/0	100%	1/1
app/_guards/auth	12.5%	1/8	0%	0/2	0%	0/1	12.5%	1/8
app/_interceptors/auth	0%	0/5	0%	0/1	0%	0/1	0%	0/5
app/_services/auth	75.75%	25/33	55.55%	5/9	90.9%	10/11	73.33%	22/30
app/_services/rest-backend	20%	4/20	100%	0/0	12.5%	1/8	20%	4/20
app/footer	100%	2/2	100%	0/0	100%	0/0	100%	1/1
app/login	100%	16/16	100%	2/2	100%	5/5	100%	15/15
app/navbar	62.5%	5/8	100%	0/0	25%	1/4	57.14%	4/7
app/navbar/dark-mode-toggle	8.69%	2/23	0%	0/11	0%	0/2	4.54%	1/22
app/todo-page	40%	12/30	0%	0/4	36.36%	4/11	39.28%	11/28
app/todo-page/todo-item	33.33%	6/18	0%	0/4	20%	2/10	33.33%	6/18

Code coverage generated by istanbul at 2024-02-10T11:00:22.833Z

END-TO-END TESTING

END–TO–END (E2E) TESTING

Goal:

- Test the system as a **whole**
- From the **point of view** of its intended **end users**

- Each test replicates a **realistic usage flow**
- Interacting with the **external interfaces** of the system
  - As end-users do!

E2E TESTING: REST APIs

When the web app is a REST API

- REST endpoints are the **external interface**
- End users are other programs sending HTTP requests
- E2E tests should send HTTP requests and check that the responses are correct according to specification

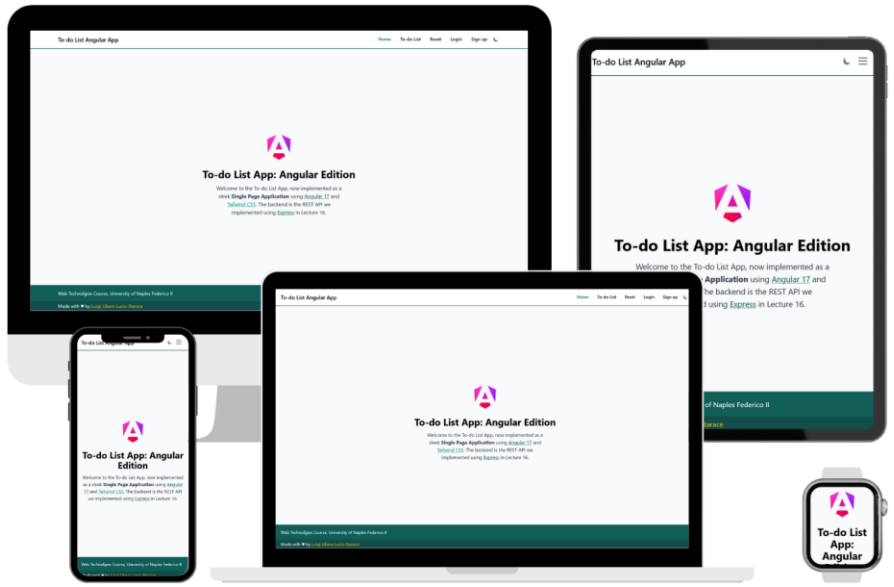
REST API

Url	Method	Description
`/todos`	`GET`	Get all To-dos
`/todos/:id`	`GET`	Get To-do by ID
`/todos`	`POST`	Save To-do
`/todos/:id`	`PUT`	Update To-do by ID
`/todos/:id`	`DELETE`	Delete To-do by ID
`/auth`	`POST`	Login request
`/signup`	`POST`	Create new user

E2E TESTING: WEB APPLICATIONS

When the web app is traditional web app or a SPA

- Web pages are the **external interface**
- End users are humans using a web browser
- E2E tests should interact with the web pages, and check that they change correctly as a result of the interactions
- We will refer to this kind of tests as **E2E web tests**



E2E REST API TESTS: EXAMPLES

Step	Description – Scenario: Successful login
1	Send a POST HTTP request to `/auth` with payload `{usr: "gerry", pwd: "sussman"}`
2	Check that the response has status code `200 OK`
3	Check that the returned value is a JSON of type `{token: string}`
4	Check that response.token is a valid JWT token

Step	Description – Scenario: User inserts a new to-do item
1	Send a POST HTTP request to `/auth` with payload `{usr: "gerry", pwd: "sussman"}`
2	Check that response has status `200 OK`, then extract and save the token in response body
3	Send a POST HTTP request to `/todos` with payload `{todo: "foobar", done: false}` and authorization header set to «`Bearer <TOKEN>`», where <TOKEN> is the value extracted in Step 2
4	Check that the response has status `200 OK`
5	Check that the response body contains a JSON with properties `{todo: "foobar", done: false}`

E2E WEB TESTS: EXAMPLES

Step	Description – Scenario: Successful login
1	Visit login page
2	Insert «gerry» in the username field
3	Insert «sussman» in the password field
4	Click on the «Login» button
5	Check that user is redirected to homepage
6	Check that the navbar contains «Hi, Gerry»

Step	Description – Scenario: User inserts a new to-do item
1	Visit todos page
2	Insert «example» in the to-do input field
3	Click on the «Save to-do» button
4	Check that the «example» to-do item has been added to the list

KEY STEPS IN E2E WEB TESTING

Whether it is done manually or in an automated fashion, E2E testing basically follows the pseudocode listed hereafter:

```
execute(test suite):
    foreach test scenario in test suite:
        foreach step in test scenario:
            select the element to interact with (button, text field, ...)
            interact with it (click, fill, check some property holds, ...)
```

MANUAL AND AUTOMATED E2E TESTING

E2E Testing can be done **manually**, or be **automated**

Manual E2E Testing

- Human testers are given a **script** with a detailed list of steps to follow to replicate user scenarios
- They follow and replicate the script step by step, reporting any issue

Automated

- Testers develop test software (test suites) that can automatically simulate user scenarios
- Test suites can be re-executed multiple times

The images in this slide are AI-generated (by DALL-E 3)

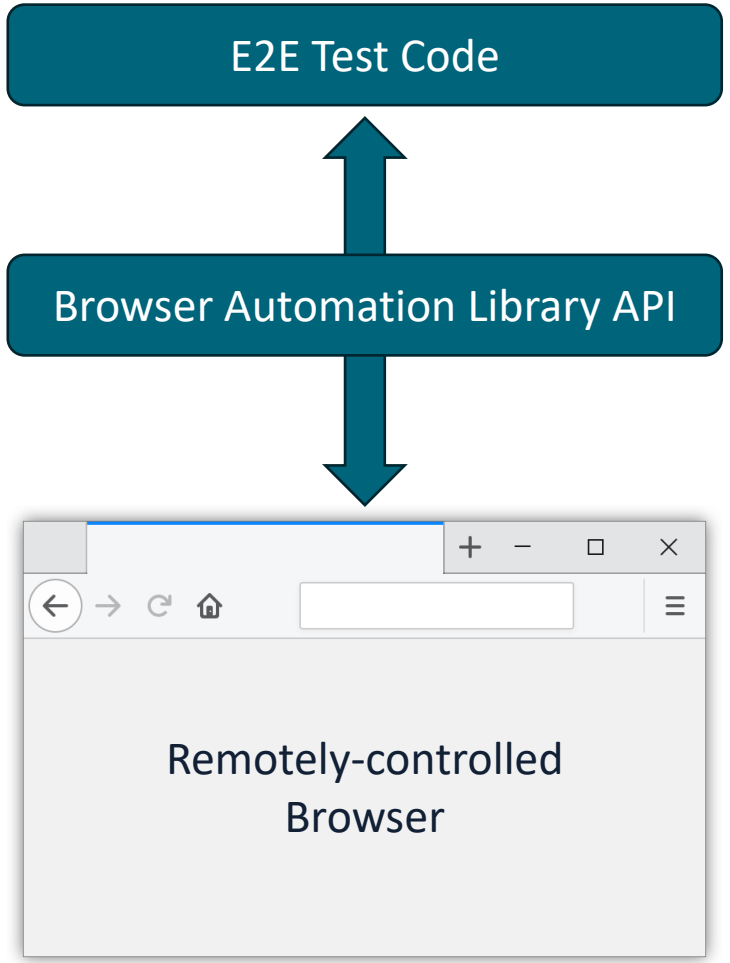
E2E TESTS: MANUAL VS AUTOMATED

- No programming knowledge required
- **Time-consuming**, **tedious**, and **error-prone** activity
- **Does not scale** well…
  - Repeat for each new release of the app
  - Repeat on all major browsers
  - Repeat on different devices

- Programming and testing knowledge required
- Higher initial costs, but the tests can be re-executed multiple times with little additional effort
- Test code needs **maintainance** (i.e., to be updated as the web app evolves)
- Tests may be **flaky** and/or **fragile**

AUTOMATED E2E WEB TESTS

- Automated E2E web tests are basically test code that leverages dedicated libraries to **remotely control** a web browser
  - Navigate to URLs
  - Locate and interact with web page elements
- Thanks to these libraries, E2E test code can simulate user interactions with a web app
- These libraries can be useful not only for testing, but also for other tasks!
  - **Scraping** (extracting data from web sites)
  - **Crawling** (navigating web sites)



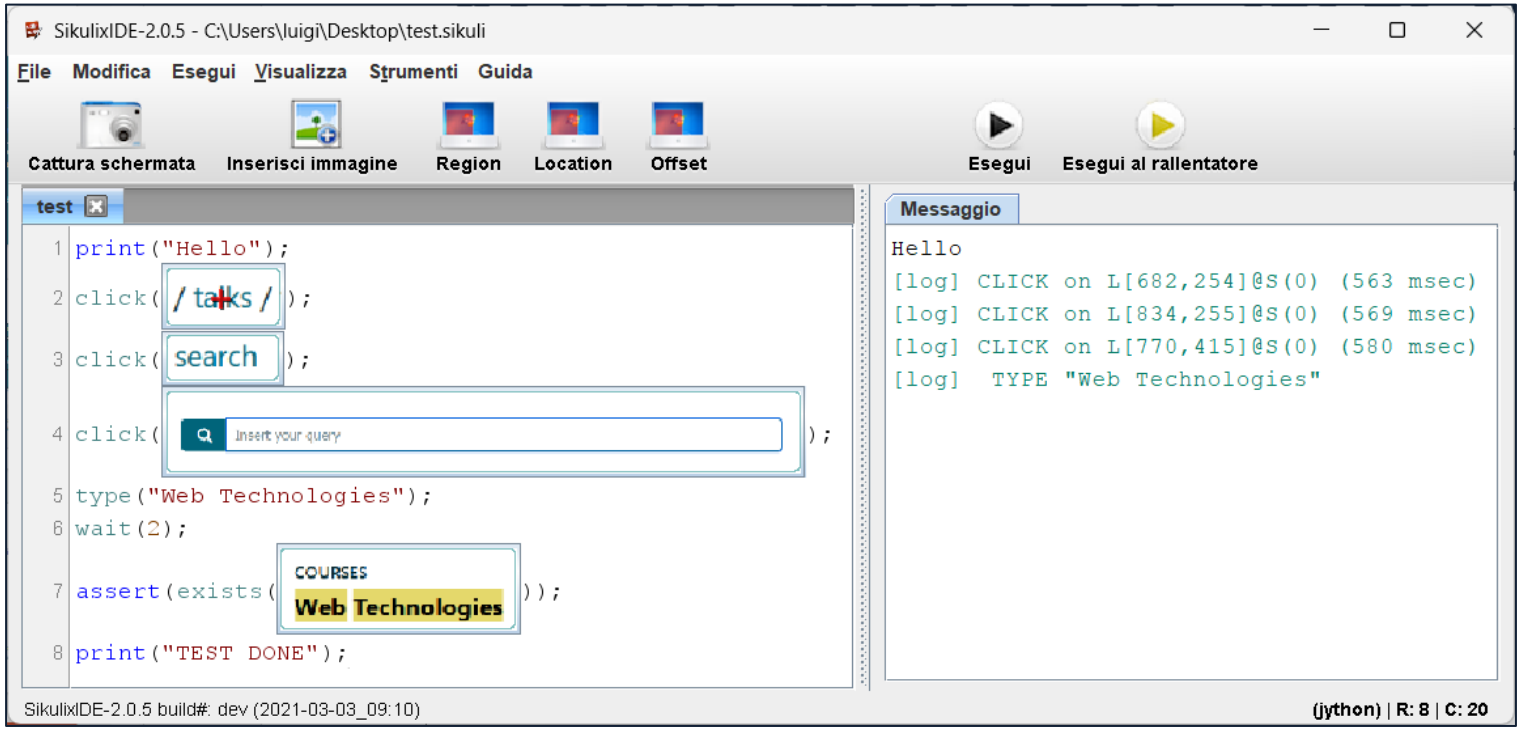
SELECTING WEB PAGE ELEMENTS

Selecting (**locating**) elements with which to interact is a crucial part of E2E web testing. Three main approaches (**locator strategies**) exist:

- **Absolute coordinates**
  - Identify elements based on screen coordinates
  - `click(200, 300);` //click at coordinates 200,300
- **Visual-based locators**
  - Identify elems based on their appearence, using image-matching algorithms
  - `click(`  `);` //click on the element matching the provided image
- **Layout-based locators**
  - Identify elements based on layout properties (e.g.: use CSS selectors!)
  - `click(".item button.buy");` //click on the button.buy contained in a .item

VISUAL WEB TEST SELECTORS

- Tools like SikuliX allow users to write E2E tests using visual locators
- **Live demo time**!



E2E WEB TESTING CHALLENGES

Two key challenges arise when doing E2E web testing

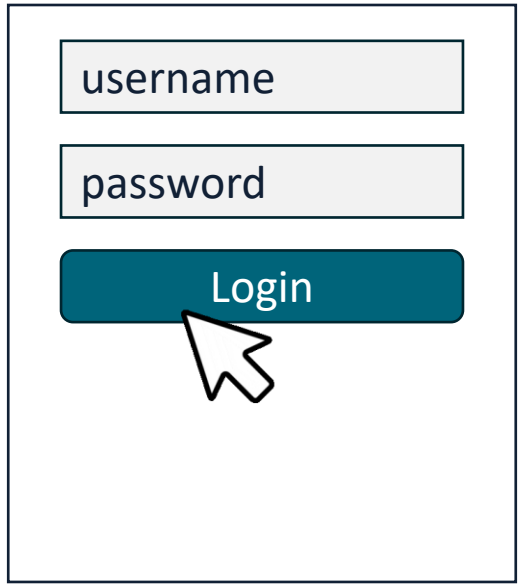
- **Fragility**
- **Flakiness**

E2E WEB TEST FRAGILITY

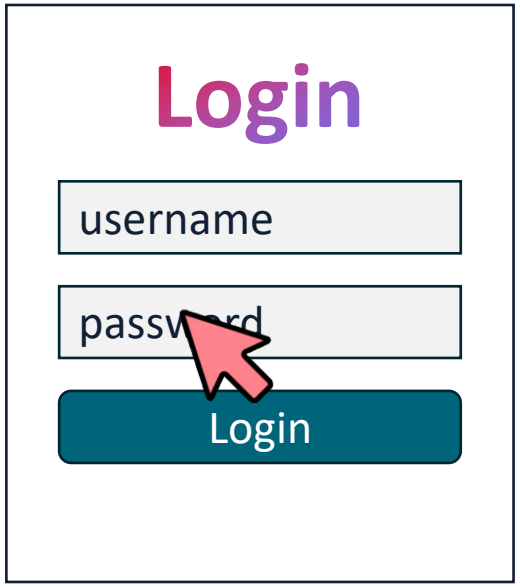
- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click(100, 200);
```

This locator breaks in v. 1.1!
The absolute coordinates do not correspond to the login button anymore!



Web App v. 1.0



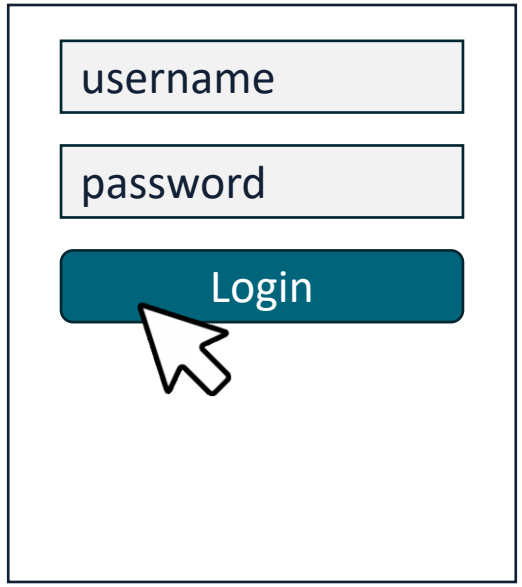
Web App v. 1.1

E2E WEB TEST FRAGILITY

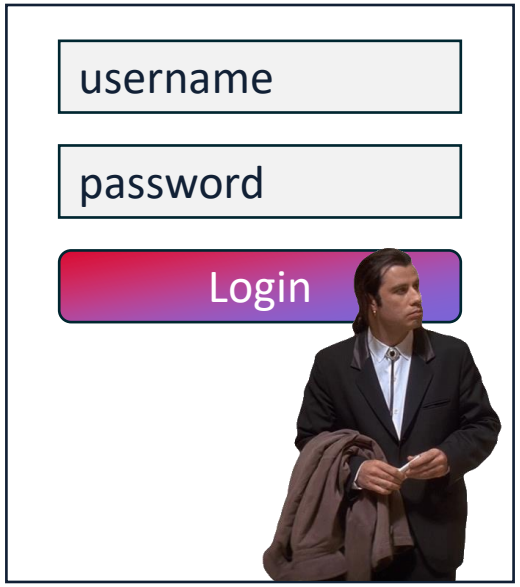
- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click( Login );
```

This locator breaks in v. 1.1!
The specified image no longer matches the appearance of the login button!



Web App v. 1.0



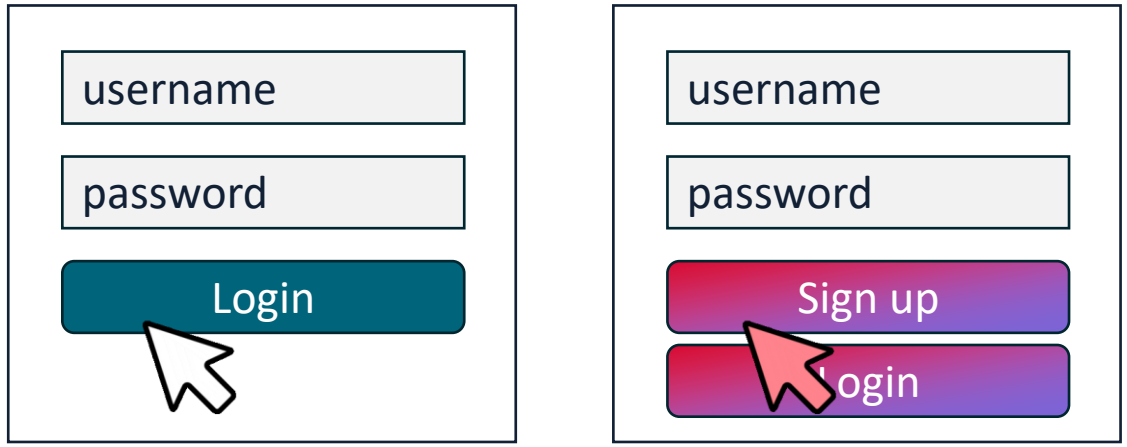
Web App v. 1.1

E2E WEB TEST FRAGILITY

- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click(".form button:first-of-type");
```

This locator breaks in v. 1.1!
A new button is added in .form before the login button. The test will incorrectly click on the new button!



Web App v. 1.0 Web App v. 1.1

LOCATOR STRATEGIES AND FRAGILITY

- Good locators should be not too generic and not too specific
  - If too generic, they might easily select a different element than intended
  - If too specific, even minor changes might break them
  - Experience goes a long way in designing robust locators!
- Coordinate-based locators are the most fragile
- Visual-based locators are robust to layout changes, but fragile w.r.t. changes to the appearence of elements
- Layout-based locators are robust w.r.t. purely visual changes, but more fragile w.r.t. layout changes
- Layout-based locators are the most used in practice

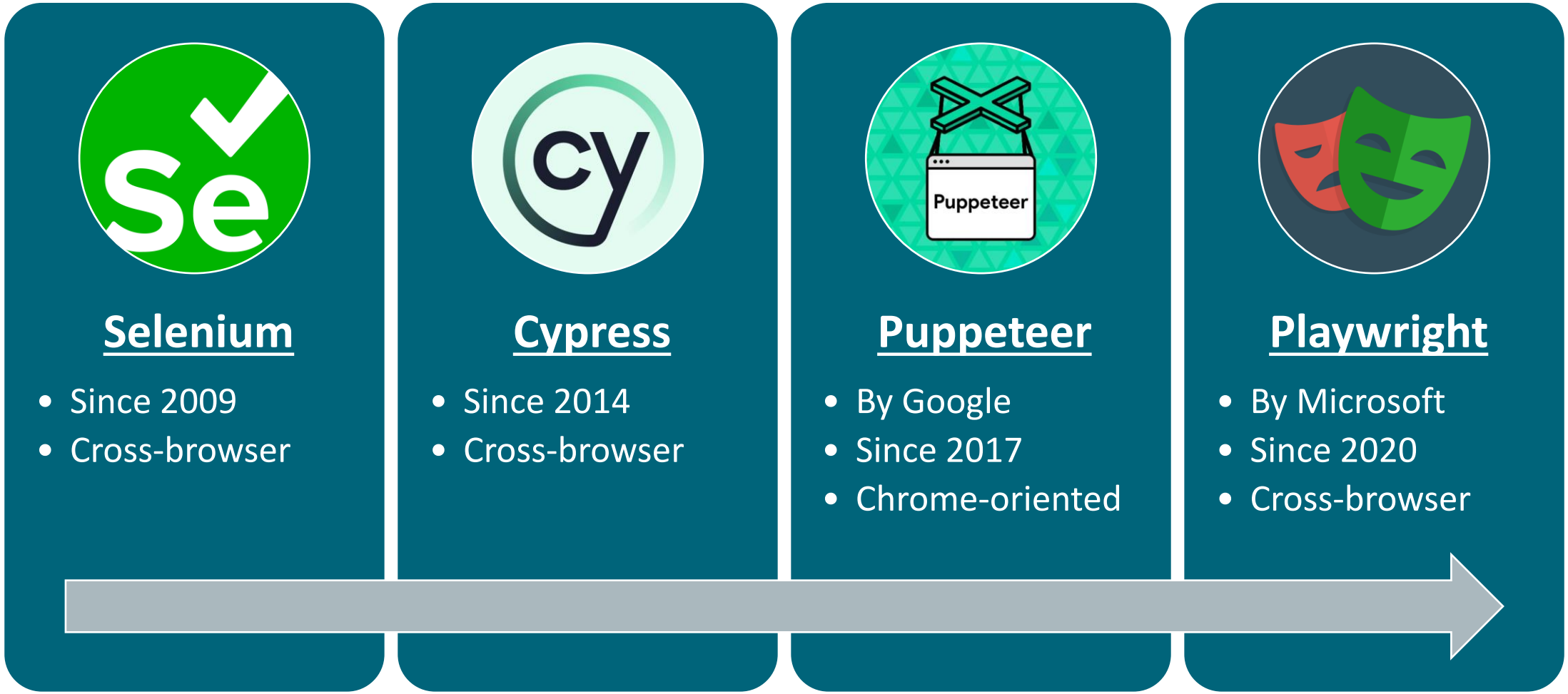
E2E WEB TEST FLAKINESS

- E2E web tests have a tendency to be **flaky**, i.e., behave inconsistently
- The same test may pass or fail intermittently under identical conditions (i.e., no change to the web app or to the test)
- This unpredictability is often caused by the non-determinism related to loading times or fetching data
  - Some times, for example, an external resource takes longer to return the data that needs to be visualized
  - As a consequence, tests might try to access elements which have not been rendered yet, resulting in errors (element not found)
- Flakiness can be mitigated by using appropriate waiting strategies

E2E TEST ISOLATION

- E2E web tests should run in **isolation**
  - **No failure carry-over** (one failing test does not affect subsequent tests)
  - Test **execution order does not matter** (tests can be executed in parallel)
  - Each test can be executed **independently** (easier to **debug**)
- E2E isolation can be achieved in two ways:
  - **Start from scratch**. Each test runs in a brand-new, clean environment
  - **Clean-up between tests**. Make sure the clean and reset the environment between tests

E2E WEB TESTING TOOLS



PLAYWRIGHT

- Open-source browser automation and testing framework
- Developed by Microsoft, first released on 31 January 2020
- **Cross-browser**, **cross-platform**
- **Event-driven**, **Async** approach
- Written in TypeScript
- APIs available in TypeScript, JavaScript, C#, Java, Python
- Slightly less popular than Selenium, Puppeteer and Cypress right now, but **way higher interest** and **retention rates** (State of JS 2022)

INSTALLING PLAYWRIGHT

Several ways to get started (see docs). Most common one is:

```
@luigi → D/O/T/W/2/e/2/playwright-example $ npm init playwright@latest
Need to install the following packages:
create-playwright@1.17.131
Ok to proceed? (y) y
Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'
√ Do you want to use TypeScript or JavaScript? · TypeScript
√ Where to put your end-to-end tests? · tests
√ Add a GitHub Actions workflow? (y/N) · false
√ Install Playwright browsers? (Y/n) · true
Initializing NPM project (npm init -y)...
...
Happy hacking! 🎭
@luigi → D/O/T/W/2/e/2/playwright-example $
```

OUR FIRST PLAYWRIGHT TEST

- In this course, we'll use TypeScript to write Playwright tests
- Tests are defined using the **test()** function
  - Takes as input a **title**, and an **async test function** containing test code
- Playwright provides APIs to remotely **control** web browsers, **locate** elements in web pages, and **interact** with them

```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

PRE–DEFINED PLAYWRIGHT TEST

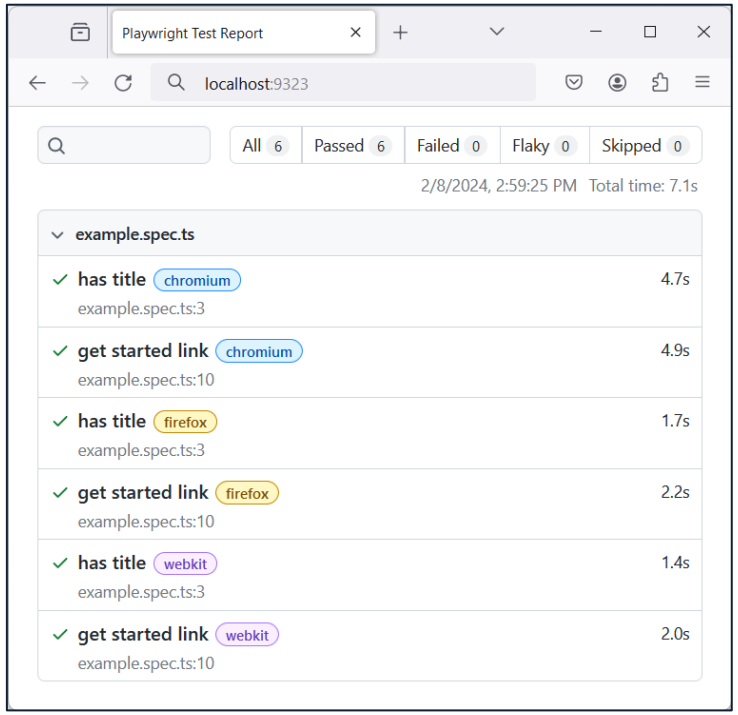
```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();
  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```

RUNNING PLAYWRIGHT TESTS

```
@luigi → playwright-example $ npx playwright test
Running 6 tests using 6 workers
  6 passed (7.1s)
@luigi → playwright-example $ npx playwright show-report
  Serving HTML report at http://localhost:9323.
```



- By default, tests are executed on
  - chromium (Blink engine)
  - firefox (Gecko engine)
  - webkit (engine used by Safari and Apple devices)
- Playwright can also emulate tablet and mobile browsers (Chrome and Safari)
- Platforms on which tests are executed can be configured in the **playwright.config.ts** file

PLAYWRIGHT: TEST ISOLATION

- Playwright achieves **isolation** by starting from scratch
  - Tests are executed in a **clean environment**, not influenced by earlier tests
  - This means that each test runs in a brand-new browser environment, with its own localStorage, sessionStorage, cookies, …
  - This is achieved by using a new **BrowserContext** for each test (you can think of a new BrowserContext as an incognito-like profile)

PLAYWRIGHT: ENVIRONMENT AND FIXTURES

- The environment for each test is defined using **Test Fixtures**
- Tipically, Playwright creates a new **BrowserContext** for each test, and provides a default **Page** in that context
- Think of a Page as an abstraction of a **tab** in a brand-new browser
- The `{page}` argument passed to the test function tells Playwright to setup the Page fixture (e.g.: create a new BrowserContext and a Page in that context)

```
test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

WRITING E2E TESTS WITH PLAYWRIGHT

Three key parts are crucial in writing automated E2E tests

- Locating elements
- Performing actions
- Asserting state against expectations

PLAYWRIGHT: LOCATING ELEMENTS

Playwright supports different ways to locate elements: docs

```
page.getByText(); // locate by text content.
page.getByLabel(); // locate a form control by associated label's text.
page.getByPlaceholder(); // locate an input by placeholder.
page.getByAltText(); // locate an element, usually image, by its text alt.
page.getByTitle(); // locate an element by its title attribute.
page.getByTestId(); // locate an element based on its data-testid
```

It is also possible to use CSS and XPATH locators

```
page.locator("#foo ul [data-foo]")
page.locator("//[@id='foo']//ul//*[@data-foo]")
```

PERFORMING ACTIONS

Once the intended element has been located, it is possible to programmatically interact with it

- It suffices to invoke the adequate method on the located element

```
page.locator("#input").clear();
page.locator("#input").fill("Foo");
page.locator("#input").press("Tab");
page.locator("#checkbox").check();
page.locator("#select-course").selectOption("web technologies");
page.locator("#button").click();
page.locator("#button").dblclick();
```

ASSERTING STATE

Once the intended element has been located, it is also possible to perform test assertions on that element

- To make assertions, call **expect(value or locator)** and then a matcher reflecting the assertion you want to make
- Check out the docs for a complete reference on test assertions

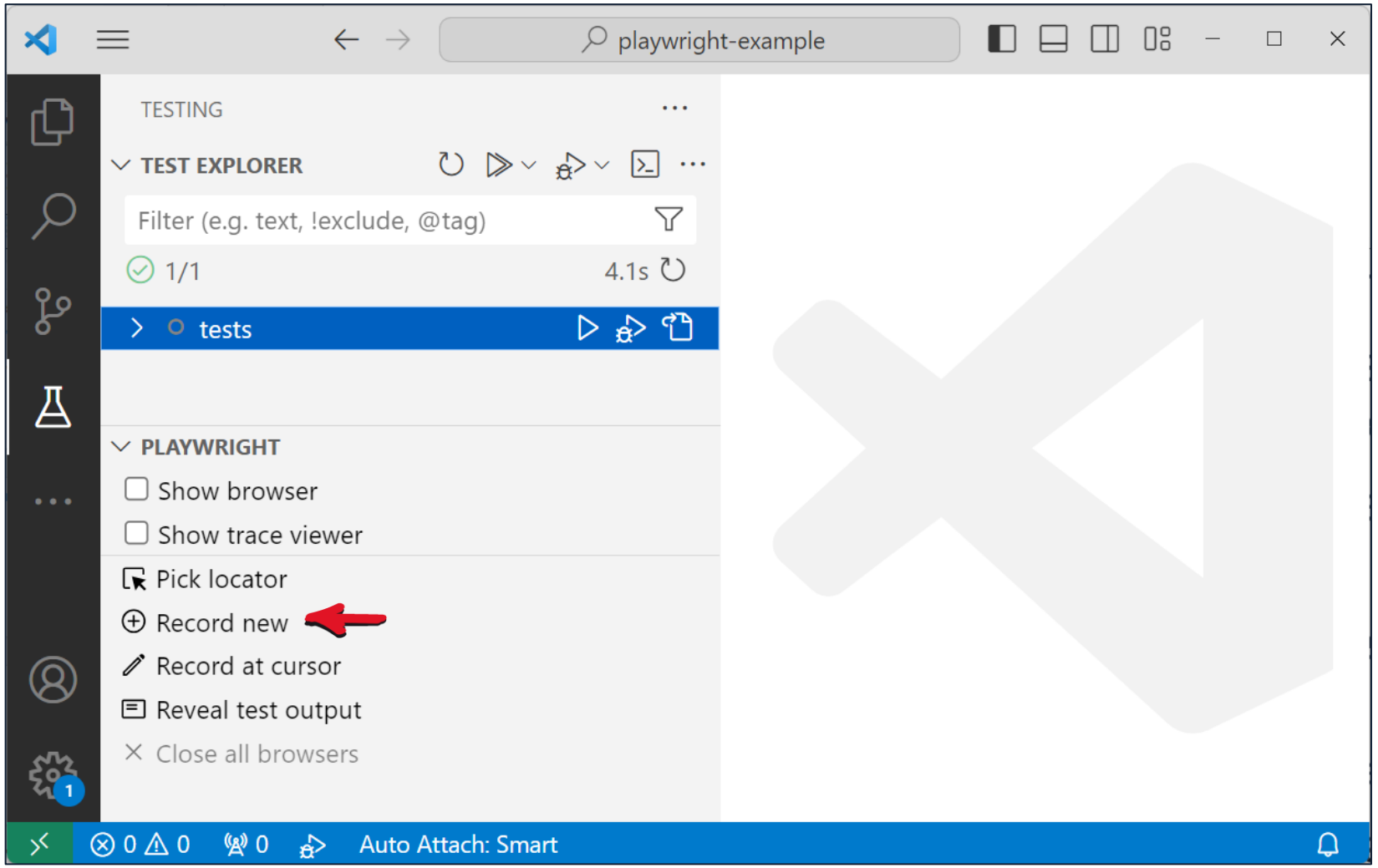
```
expect(page.locator("#foo")).toBeDefined();
expect(page.locator("#foo")).toBeDisabled();
expect(page.locator("#foo")).toBeEmpty();
expect(page.locator("#foo")).toContainText("foo");
expect(page.locator("#foo")).toHaveClass("bar");
expect(page.locator("#foo")).toBeVisible();
expect(page.locator("#foo")).toBeInViewport();
```

CAPTURE AND REPLAY

- Writing tests by hand is a time consuming task
- **Capture and Replay** (C&R) approaches allow to automatically generate E2E test code by «recording» (capturing) user interactions
- C&R allows testers to easily create re-executable E2E tests by simply interacting with the web app
- Being generated automatically, C&R test code may be suboptimal w.r.t. fragility or flakiness
  - We may use it as a starting point nonetheless!
- Playwright supports C&R (through its **Test Generator** feature), so let's see it in action!

PLAYWRIGHT TEST GENERATOR

- The easiest way to get started is to use the VS Code Playwright extension
- In the Testing tab, under the Playwright tab, we can select «**Record new**»



PLAYWRIGHT TEST GENERATOR

- After starting the recording process, a browser window will appear
- We just have to interact with the browser as an end-user
- Playwright will capture all our interactions, generating the code necessary to replicate them
- Built-in heuristics are used to determine the best locators

```
test('should login with correct credentials', async ({ page }) => {
  await page.goto('http://localhost:4200/home');
  await page.getByRole('link', { name: 'Login' }).click();
  await page.getByPlaceholder('Your username').fill('luigi');
  await page.getByPlaceholder('••••••••').fill('luigi');
  await page.getByRole('button', { name: 'Sign in' }).click();
  await expect(page.locator('#navbar-default')).toContainText('Welcome, luigi');
});
```

REFERENCES

- **Angular Testing**
  https://angular.dev/guide/testing
  Relevant parts: Overview, Basics of testing components, Component Testing Scenarios, Testing services, Code coverage
- **Playwright Docs**
  https://playwright.dev/docs/intro
  Relevant parts: Getting started, Guides > Best practices