Testing in Angular

Resting Easy with the TestBed API

Jeremy Zevin 7/10/2024

How Many People Currently Use Testing In Their Projects?

Be Honest 😇

Overview

Welcome & Agenda Overview

- Introduction ~1min
- Angular Testing Fundamentals ~4mins
- A Simple App ~5mins
- Unit Testing ~15mins
- E2E Testing ~15mins
- Conclusion ~5mins
- Bonus Testing ~5mins*
- Q&A ~5mins

*Time permitting

Team Alchemy

I am not an expert, but rather a testing advocate, I have witnessed the benefits firsthand.

Angular’s Unit Testing Ecosystem is Built-In

Angular provides a comprehensive testing framework out-of-the-box.
Tools like Jasmine for writing tests, Karma for running them, making it easier to start testing without additional setup.

Fundamentals

of Testing in Angular

Angular Testing Fundamentals

- Key concepts of testing in Angular
- Overview of different types of testing: unit, integration, and end-to-end (E2E)
- Benefits of testing ~~in Angular~~

Angular Testing Fundamentals
Key Concepts

- Unit Testing
- Integration Testing
- End-to-End (E2E) Testing
- TestBed API
- Mocks and Spies
- Async Testing
- Component Testing
- Service Testing
- Forms Testing

You get what you inspect,
not what you expect.

Angular Testing Fundamentals

Types

- **Unit Testing**
- **Integration Testing**
- **End-to-End (E2E) Testing**

Angular Testing Fundamentals
Benefits

- Ensuring code quality and reliability
- Early Bug Detection
- Facilitating More Efficient Code Refactoring

Efficiency!

Ensuring code quality and reliability

Benefits

- Consistency: Automated tests ensure code behaves as expected.
- Regression Prevention: Prevent new changes from breaking existing functionality.
- Code Coverage: High test coverage reduces the risk of undetected bugs.
- Documentation: Tests act as documentation for expected behavior.

benefit

benefit

Early Bug Detection

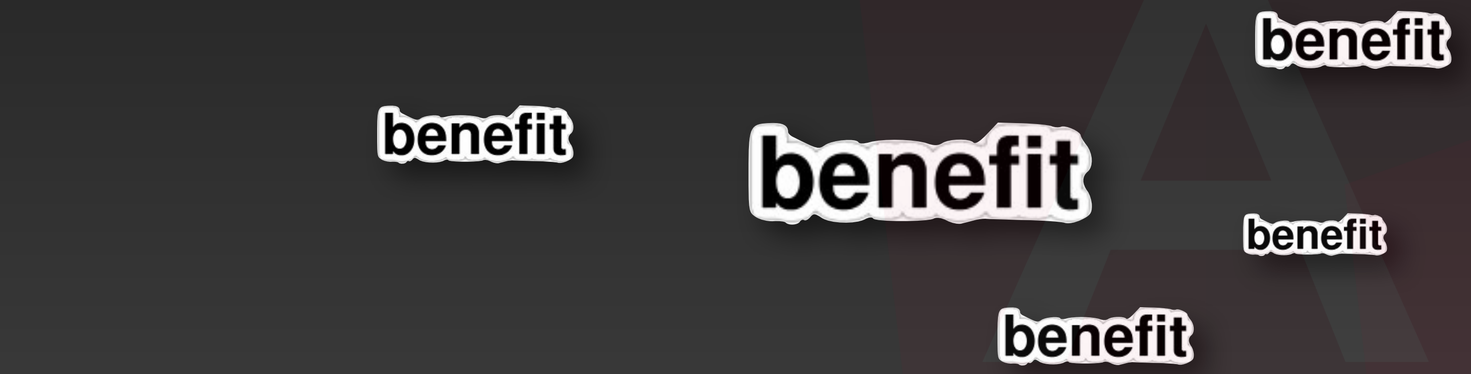
Benefits

- Immediate Feedback: Catch and fix bugs early in the development cycle.
- Isolation of Issues: Easier to identify and address bugs in specific modules or components.
- Continuous Integration: CI/CD pipelines catch bugs as soon as they are introduced.

Facilitating Code Refactoring

Benefits

- Confidence: Tests give confidence to refactor code without introducing new bugs.
- Code Improvement: Ensures refactoring improves maintainability and performance.
- Simplified Maintenance: Tests ensure that new features or changes do not degrade quality.



A Simple App

Demo

A Simple App

Testing Our Simple App

Lets Add Unit Testing!

Angular Unit Testing

Overview

- Unit Testing Basics
- Writing Effective Unit Tests
- Enhancing Test Coverage
- Advanced Unit Testing Techniques

Angular Unit Testing

What is TestBed?

- The TestBed is a powerful unit testing API provided by Angular.

Unit Testing our Simple App

Anatomy of a Test Suite

```
describe(‘SimpleApp', () => {
  // test suite stuffs
});
```

Unit Testing our Simple App

Setup and Teardown

```
describe('SimpleApp', () => {
  beforeEach(() => {
    // setup code
  });

  afterEach(() => {
    // teardown code
  });
});
```

Unit Testing our Simple App

Test Bed Configuration

```
describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [AppComponent],
      providers: [SimpleDataService, provideHttpClient()]
    }).compileComponents();
  });
```

Unit Testing our Simple App

A Basic Test!

```
it('should create the component', () => {
  expect(component).toBeTruthy();
});
```

Unit Testing our Simple App

Where to begin?

- Now that we know how to create the basic blocks lets initialize our TestBed and examine the current state of testing in our Simple App
- Then we can check the coverage we currently have with Istanbul
  - Created with flag --code-coverage

Unit Testing our Simple App

Component Initialization

```
let fixture: ComponentFixture<AppComponent>;
let app: AppComponent;

beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [AppComponent],
    providers: [SimpleDataService, provideHttpClient()]
  }).compileComponents();
  fixture = TestBed.createComponent(AppComponent);
  app = fixture.componentInstance;
});
```

Unit Testing our Simple App

Where to begin?

- We could even mock the dependancies

Unit Testing our Simple App

Mocking Dependencies

```
beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [AppComponent],
      providers: [
        {
          provide: SimpleDataService,
          useValue: jasmine.createSpyObj('SimpleDataService', { getName: of('foo') }),
          // Mock implementation returns an observable
        },
        {
          provide: HttpClient,
          useValue: jasmine.createSpyObj('HttpClient', ['get']),
        },
      ],
    }).compileComponents();
    fixture = TestBed.createComponent(AppComponent);
    app = fixture.componentInstance;
    simpleDataService = TestBed.inject(SimpleDataService) as jasmine.SpyObj<SimpleDataService>;
  });
```

Unit Testing our Simple App
Coverage?

- Let’s check our coverage before going further

Unit Testing our Simple App

Testing Component Methods

```
it('should call getRandomName', () => {
  app.getRandomName();
  expect(simpleDataService.getName).toHaveBeenCalled();
});
```

Unit Testing our Simple App

Check DOM Changes

```
  it('should make DOM changes', () => {
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('h1.display-1 i')).toHaveClass('text-danger');
    fixture.componentInstance.onClickFillName();
    fixture.detectChanges();
    expect(compiled.querySelector('h1.display-1 i')).toHaveClass('text-warning');
  });
```

Unit Testing our Simple App

Check Async Calls

```
it('should fetch data asynchronously', async () => {
  spyOn(component, 'fetchData').and.returnValue(Promise.resolve('data'));
  await component.fetchData();
  expect(component.data).toBe('data');
});
```

Unit Testing our Simple App

Check Our Form Controls

```
it('should have a valid form when required fields are filled' , () => {
  app.simpleFormGroup.setValue({ name: 'foo' });
  expect(app.simpleFormGroup.valid).toBeTrue();
});
```

Unit Testing our Simple App

Check Our Form Controls

```
it('should have an invalid form when required fields are empty' , () => {
  expect(app.simpleFormGroup.valid).toBeFalse();
});

it('should have a valid form when required fields are filled' , () => {
  app.simpleFormGroup.setValue({ name: 'foo' });
  expect(app.simpleFormGroup.valid).toBeTrue();
});
```

Unit Testing our Simple App

Putting It All Together: A Suite of Specs for our Simple App

- AppComponent
  - should call onClickFillName
  - should call getRandomName
  - should have an invalid form when required fields are empty
  - should have a valid form when required fields are filled
  - should make DOM changes
  - should create the app

```
describe('AppComponent', () => {
  let fixture: ComponentFixture<AppComponent>;
  let app: AppComponent;
  let simpleDataService: jasmine.SpyObj<SimpleDataService>;

  beforeEach(async () => {
    await TestBed.configureTestingModule(
  …

  it('should create the app', () => {
    expect(app).toBeTruthy();
  });

  it('should call getRandomName', () => {
    app.getRandomName();
    expect(simpleDataService.getName).toHaveBeenCalled();
  });

  it('should call onClickFillName', () => {
    app.onClickFillName();
    expect(simpleDataService.getName).toHaveBeenCalled();
  });

  it('should make DOM changes', () => {
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('h1.display-1 i')).toHaveClass('text-danger');
    fixture.componentInstance.onClickFillName();
    fixture.detectChanges();
    expect(compiled.querySelector('h1.display-1 i')).toHaveClass('text-warning');
  });

  it('should have an invalid form when required fields are empty' , () => {
    expect(app.simpleFormGroup.valid).toBeFalse();
  });

  it('should have a valid form when required fields are filled' , () => {
    app.simpleFormGroup.setValue({ name: 'foo' });
    expect(app.simpleFormGroup.valid).toBeTrue();
  });
});
```

File	Statements		Branches
app	100%	12/12	
app/core	50%	2/4	

100% Statements 12/12
100% Branches 0/0
100% Functions 5/5
100% Lines 12/12

Best Practices

- Isolate Tests
- Use TestBed for Configuration
- Write Tests for Both Positive and Negative Scenarios
- Use Angular Testing Utilities
- Ensuring tests are fast and reliable

E2E Testing

Is Super Fun!

E2E Testing our Simple App

Playwright Installation

- Playwright isn’t integrated into the Angular Ecosystem yet

E2E Testing our Simple App

Playwright Benefits

- Cross-browser
- Cross-platform
- Cross-language

E2E Testing our Simple App

Playwright Benefits

- Auto-wait
- Cross-platform
- Tracing

Resilient • No flaky tests!

E2E Testing our Simple App

Playwright Benefits

- Full Isolation
- Fast Execution
- Powerful Tooling
  - Codegen
  - Inspector
  - Trace Viewer

E2E Testing our Simple App

Playwright Benefits

- And…

Parallelism

E2E Testing our Simple App

The Basics

```
test.describe('Simple App', () => {
    // similar to unit
});
```

E2E Testing our Simple App

The Basics - Locators & Async

- Playwright has a Locators API

E2E Testing our Simple App

The Basics - Locators & Async

- Playwright has a Locators API

```
await page.getByRole('link', { name: 'Get started' }).click();
```

No need to write custom selectors!

E2E Testing our Simple App

The Basics - Locators & Async

- Playwright has isolated Page context

E2E Testing our Simple App

The Basics - Locators & Async

- Playwright has isolated Page context

```
test('example test', async ({ page }) => {
  // "page" belongs to an isolated BrowserContext, created for this specific test.
});

test('another test', async ({ page }) => {
  // "page" in this second test is completely isolated from the first test.
});
```

E2E Testing our Simple App

The Basics - Test Hooks

- Playwright has similar hooks that we previously learned about in our unit tests

E2E Testing our Simple App

The Basics - Test Hooks

```
test.describe('Simple App', () => {
  test.beforeEach(async ({ page }) => {
    // Perform some setup before each test
  });

  test.afterEach(async ({ page }) => {
    // Perform some teardown after each test
  });

  test(async ({ page }) => {
    // test stuffs
  });
});
```

E2E Testing our Simple App

The Basics

- Now that we have the basics let’s write our first e2e test

E2E Testing our Simple App

The Basics

```
  test('has title', async ({ page }) => {
    await page.goto('http://localhost:4200/');

    // Expect a title "to contain" a substring.
    await expect(page).toHaveTitle(/SimpleApp/);
  });
```

E2E Testing our Simple App

The Basics

```
// 👍
await expect(page).toHaveTitle(/SimpleApp/);

// 👎
expect(await page.toHaveTitle(/SimpleApp/));
```

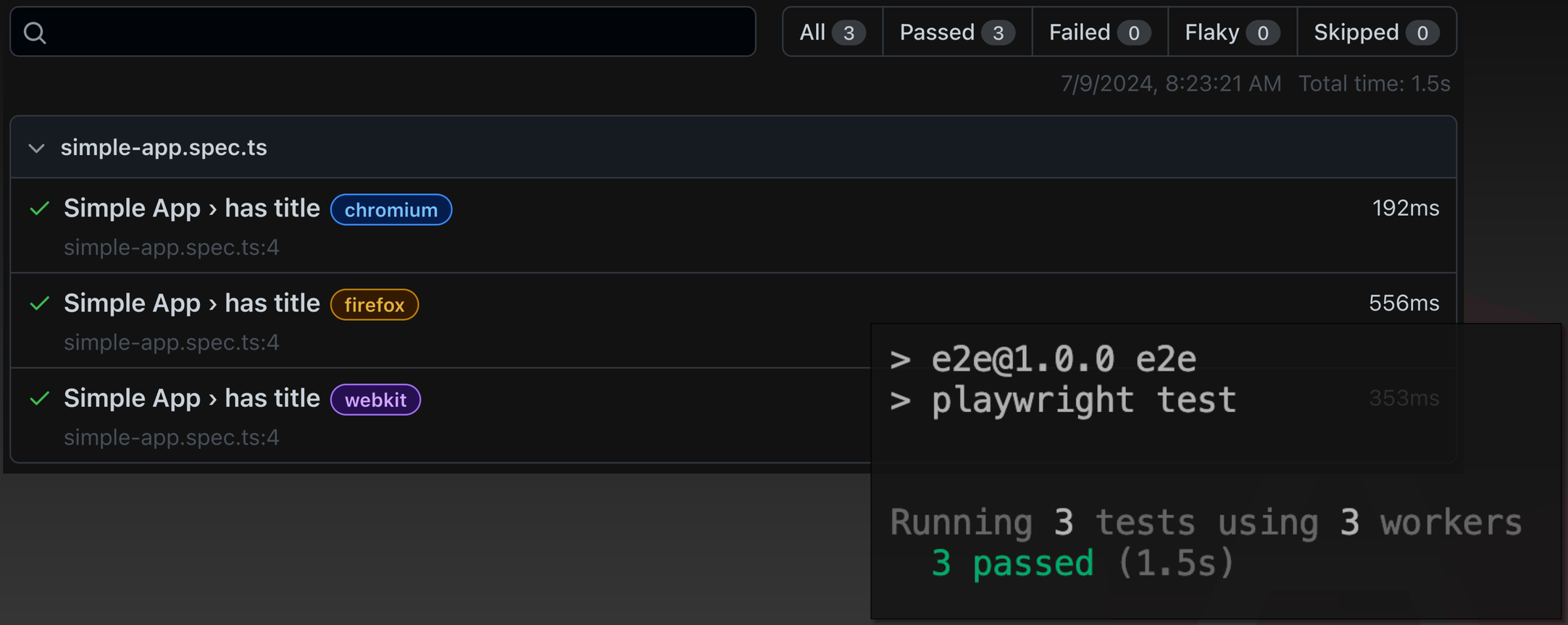
E2E Testing our Simple App

The Basics

- Let’s now run our first e2e test with the CLI
- And add a script to our package.json so we don’t need to install PW -g

E2E Testing our Simple App

The Basics

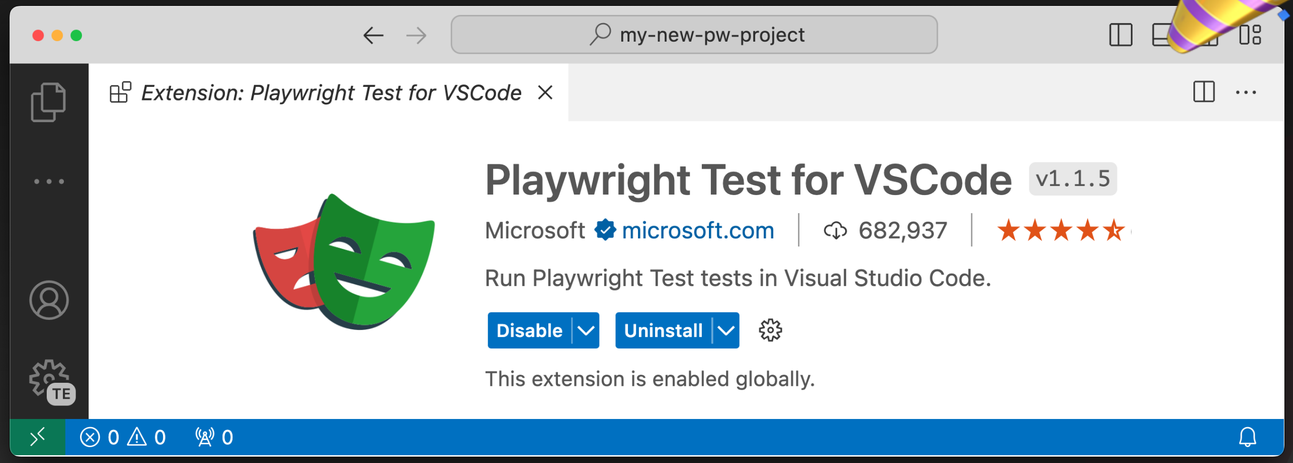


E2E Testing our Simple App

The Basics

- Are there other options for running our e2e tests?

Playwright VSCode Extension



E2E Testing our Simple App

Diving Deeper

- Now, let’s dive deeper into testing our Simple App
- And checkout how the VSCode extension works

E2E Testing our Simple App

Diving Deeper

```
test('face icon is visible and has the correct initial classes', async ({ page }) => {
  await page.goto('http://localhost:4200/');

  // Expect an element "to be visible".
  await expect(page.locator('i').first()).toBeVisible();

  // Expect an element "to have class".
  await expect(page.locator('i').first()).toHaveClass('bi-emoji-frown-fill');
});
```

E2E Testing our Simple App

Diving Deeper

- Uh oh we have some issues with our locators and some other errors
- Let’s use the extension to help us debug, fix and improve our test

E2E Testing our Simple App

Diving Deeper

```
test('face icon is visible and has the correct initial classes', async ({ page }) => {
  await page.goto('http://localhost:4200/');

  const headingIcon = await page.getByRole('heading').first().locator('i');

  // Expect an element "to be visible".
  await expect(headingIcon).toBeVisible();

  // Expect an element "to have class".
  await expect(headingIcon).toHaveClass('bi bi-emoji-frown-fill text-danger');
});
```

E2E Testing our Simple App

Diving Deeper

- Now let’s add several tests to make sure our magic button is working as expected

E2E Testing our Simple App

Diving Deeper

```
  test("clicking on the magic icon changes the face's class", async ({
    page,
  }) => {
    await page.goto('http://localhost:4200/');

    const headingIcon = await page.getByRole('heading').first().locator('i');
    await page.getByRole('button').locator('i.bi-magic').click();

    // Expect an element "to have class".
    await expect(headingIcon).toHaveClass(
      'bi bi-emoji-smile-fill text-warning'
    );
  });
```

E2E Testing our Simple App

Diving Deeper

```
  test('clicking on the magic icon makes the clear button visible', async ({
    page,
  }) => {
    await page.goto('http://localhost:4200/');

    const clearButton = await page.getByRole('button').locator('i.bi-x');
    await page.getByRole('button').locator('i.bi-magic').click();

    // Expect an element "to be visible".
    await expect(clearButton).toBeVisible();
  });
```

E2E Testing our Simple App

Diving Deeper

- Now let’s add several tests to make sure our name input and clear button are working as expected

E2E Testing our Simple App

Diving Deeper

```
test('clicking on the clear button resets the face icon and clears the name input', async ({
  page,
}) => {
  await page.goto('http://localhost:4200/');

  const headingIcon = await page.getByRole('heading').first().locator('i');
  const nameInput = await page.getByRole('textbox').first();

  await page.getByRole('button').locator('i.bi-magic').click();
  await page.getByRole('button').locator('i.bi-x').click();

  // Expect an element "to have class".
  await expect(headingIcon).toHaveClass('bi bi-emoji-frown-fill text-danger');

  // Expect an element "to have text".
  await expect(nameInput).toHaveText('');
});
```

E2E Testing our Simple App

Diving Deeper

```
test('entering a name and clicking on the magic icon changes the face and sets the name', async ({
    page,
  }) => {
    await page.goto('http://localhost:4200/');

    const headingIcon = await page.getByRole('heading').first().locator('i');
    const nameInput = await page.getByPlaceholder('name');

    await nameInput.fill('John');

    // Expect an element "to have class".
    await expect(headingIcon).toHaveClass(
      'bi bi-emoji-smile-fill text-warning'
    );

    // Expect an element "to have text".
    await nameInput.blur();
    await expect(nameInput).toHaveValue('John');
  });
```

E2E Testing our Simple App

Diving Deeper

```
  test('entering a name less than 2 chars should not change the face', async ({
    page,
  }) => {
    await page.goto('http://localhost:4200/');

    const headingIcon = await page.getByRole('heading').first().locator('i');
    const nameInput = await page.getByPlaceholder('name');

    await nameInput.fill('J');

    // Expect an element "to have class".
    await expect(headingIcon).toHaveClass('bi bi-emoji-frown-fill text-danger');
  });
```

E2E Testing our Simple App

Diving Deeper

```
  test('entering a name as 2 spaces should not change the face', async ({
    page,
  }) => {
    await page.goto('http://localhost:4200/');

    const headingIcon = await page.getByRole('heading').first().locator('i');
    const nameInput = await page.getByPlaceholder('name');

    await nameInput.fill('  ');

    // Expect an element "to have class".
    await expect(headingIcon).toHaveClass('bi bi-emoji-frown-fill text-danger');
  });
```

E2E Testing our Simple App

Diving Deeper

- Oops missed something in development, luckily we found this early!




E2E Testing our Simple App

Diving Deeper

- Ok now we’re moving along good now so let’s add something more complex
- Let’s intercept the api call invoked when we click the magic button and mock our own response to make sure our input has the right value

E2E Testing our Simple App

Diving Deeper

```
  test('clicking on the magic icon should call the api', async ({ page }) => {
    await page.route('*/**/api/names', async (route) => {
      await route.fulfill({ json: [{ name: 'jz' }] });
    });
    await page.goto('http://localhost:4200/');

    const headingIcon = await page.getByRole('heading').first().locator('i');
    const nameInput = await page.getByPlaceholder('name');

    await page.getByRole('button').locator('i.bi-magic').click();

    // Expect an element "to have class".
    await expect(headingIcon).toHaveClass(
      'bi bi-emoji-smile-fill text-warning'
    );

    // Expect an element "to have text".
    await expect(nameInput).toHaveValue('jz');
  });
```

E2E Testing our Simple App

Diving Deeper

- Finally, let’s add one more test to show off another great feature of Playwright

Playwright Codegen

Playwright

Playwright Best Practices

- Make tests as isolated as possible
- Avoid testing third-party dependencies
- Use locators
- Use web first assertions
- Use Playwright's Tooling
- Keep your Playwright dependency up to date

In Conclusion

Testing In Angular
In Conclusion

- Key Takeaways
  - Importance of Testing: Ensures code quality, reliability, and facilitates refactoring.
  - Types of Testing: Unit, Integration, and E2E, each serving a unique purpose.
  - Tools and Best Practices: Utilizing Angular’s built-in tools and modern frameworks like Playwright, and leveraging the reporting from these tools.
  - Practical Examples: Demonstrated through everyday tasks and scenarios familiar to developers.

Testing In Angular

In Conclusion

- Next Steps:
  - Start Small: Begin by writing simple unit tests for your components.
  - Explore Advanced Testing: Experiment with integration and E2E testing.
  - Apply Best Practices: Isolate tests, mock dependencies, and use tools like TestBed and Playwright.

Testing In Angular

In Conclusion

- Further Learning:
  - Documentation: Refer to Angular and Playwright official docs for detailed guides.
  - Community: Join Angular and Playwright communities for support and networking.
  - Practice: Continuously practice writing and improving your tests.

https://angular.dev

https://playwright.dev

BONUS DEMO!

Load Testing with Artillery & Playwright

Load Testing
With Artillery & Playwright

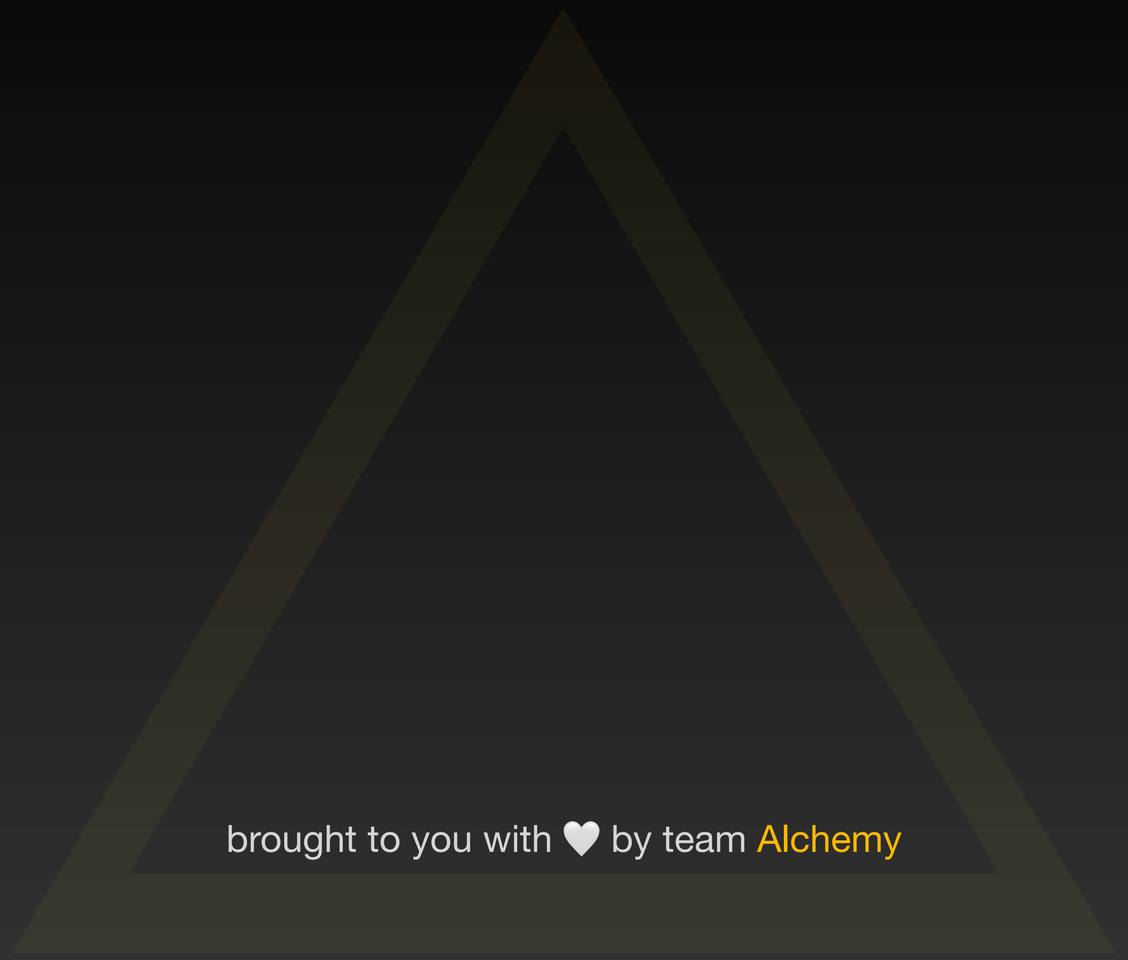
- Introduction to load testing tools
- Basic setup and execution of a load test
- Importance of load testing for performance and scalability

Q&A

Questions?

Thanks!

Thank you for your time and participation!


brought to you with 🤍 by team Alchemy