Testing Tools Comparison

Evaluating System, UI, and Unit Testing Tools for Web Development Stacks

Overview

- Why test?
- Testing level
- Comparison various tools on each Level
  - Jest vs. Vitest vs. Mocha, Junit vs. TestNG
  - TestBed, HttpTestingController, Quarkustest, Testcontainers, RestAssured
  - Cypress vs. Selenium vs. Playwright, Cucumber
- Tips/ best practices
- setup recommendation for the project
- demoproject

Why test?

- Prevent bugs and verify code behaves as expected
- Identify issues quickly before reaching production
- Encourage modular, maintainable, and clean code
- Performance, Compatibility(different browsers), Security
- Test-Driven Development (TDD)

Testing level unittest

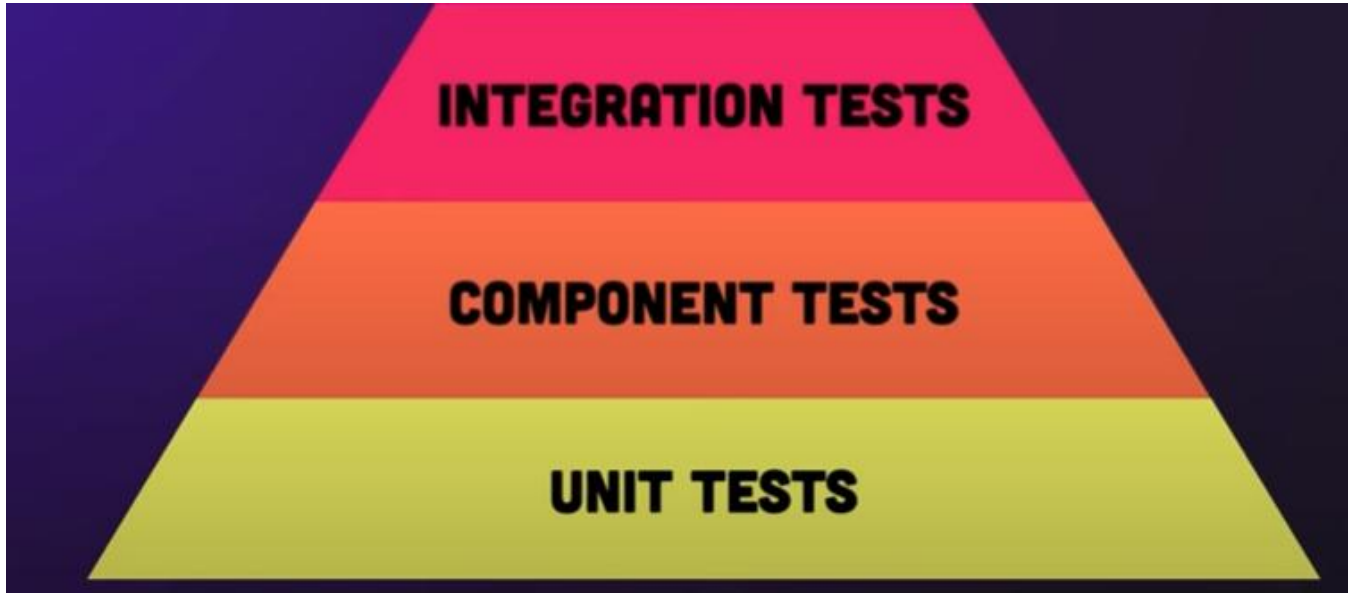
- write unit tests for all the methods and functions in our code to make sure that our code is working at the lowest level
- Aiming for 100% code coverage
- Mostly not complex and fast

Testing level componenttest

- Testing components in isolation(mock out the database and any other external components that get called)
- application is working as you expect given certain inputs
- Component tests make sure that all those units that you tested in the previous level also work when you put them together

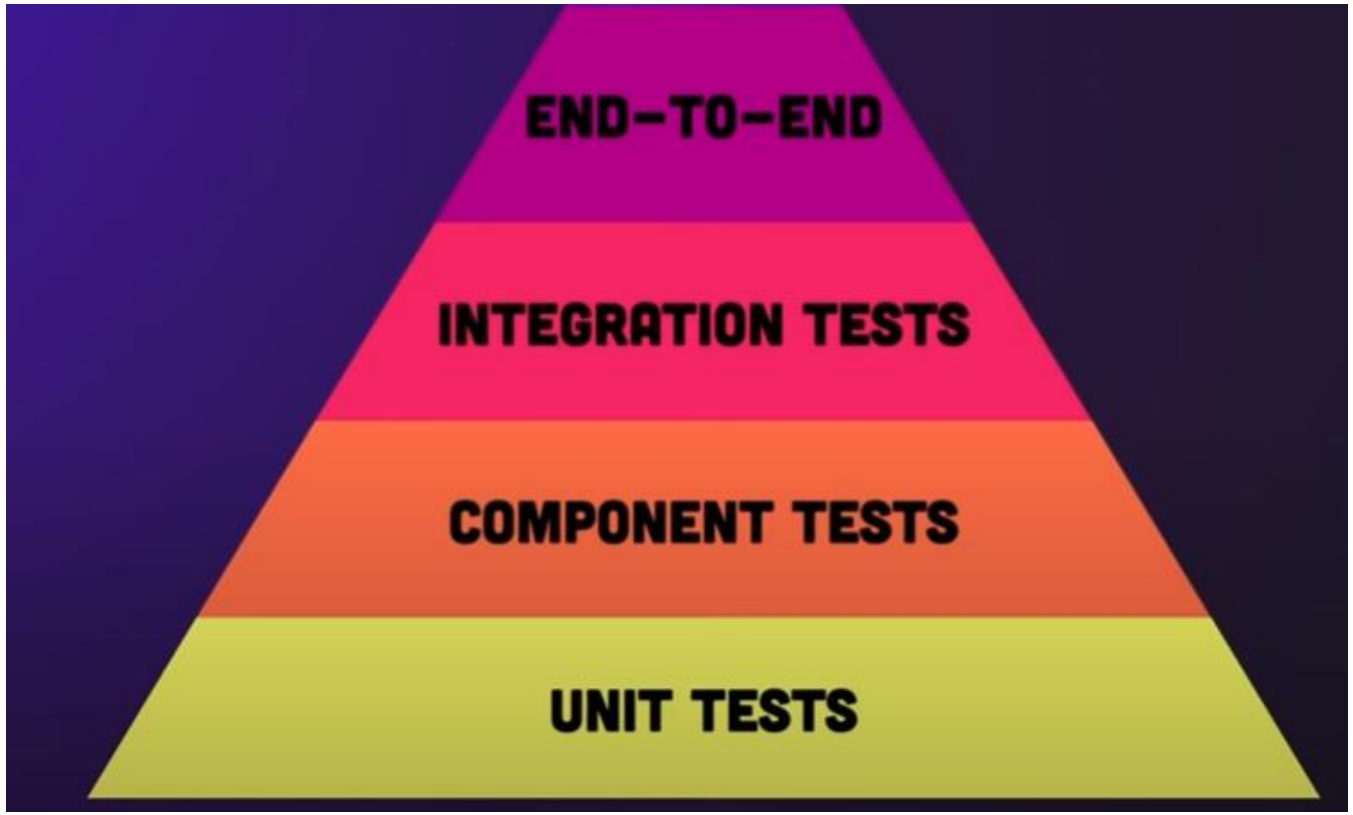
Testing level Integrationtest

- test the integration between components
- Tested Elements: Interfaces, data flow, communication between modules



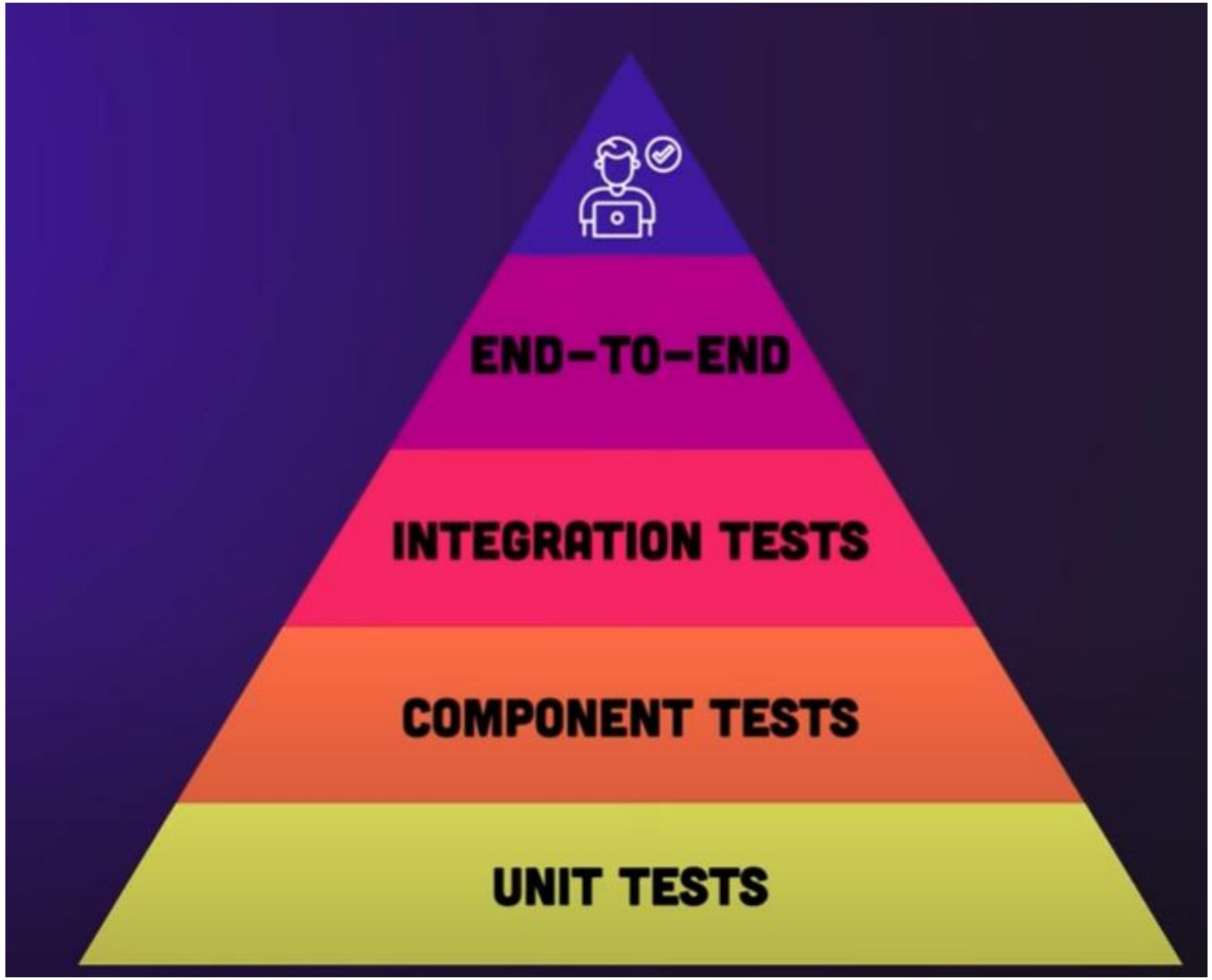
Testing level E2E

- validates the complete workflow of an application from start to finish
- Ensures the entire system behaves as expected, including frontend, backend, and any integrated services
- Tested Elements: User interfaces (UI), APIs and backend systems, Databases and third-party integrations



Testing level

- Some features might be too hard to automate or not worth the time in doing it
- If you find a bug in your application it is always better to find it lower down the pyramid than near the top, else you hunt through the logs and try and work out where the application went wrong



Unittests

Jest

- installation: npm install --save-dev jest (ts-jest @types/jest typescript)
- Test execution: npx jest
- Well-suited for React projects (officially supported by Create React App)

Pro	contra
Very fast unit tests via parallel execution and snapshot testing	not natively designed for TypeScript
Out-of-the-box configuration with built-in test runner, mocking, and assertions	Can slow down in very large codebases with thousands of tests
Large community and strong documentation	

Jest

```ts
export function add(a: number, b: number): number {  Show usages
    return a + b;
}

export function multiply(a: number, b: number): number {  Show usages
    return a * b;
}
```

```ts
import { add, multiply } from './math';

describe( name: 'math functions', () : void => {
    test( name: 'add adds two numbers', () : void => {
        const result : number = add( a: 2,  b: 3);
        expect(result).toBe( expected: 5);
    });

    test( name: 'multiply multiplies two numbers', () : void => {
        const result : number = multiply( a: 4,  b: 5);
        expect(result).toBe( expected: 20);
    });
});
```

Mocks

- Simulated objects that mimic real-world behavior to isolate dependencies
- Avoid external systems (APIs, databases, timers, etc.)
- jest.fn() (Mock implementation that mimics the behavior of a function. This allows calls and arguments to the function to be monitored without changing the original function. E.g. toHaveBeenCalledWith checks whether the function has been called with specific arguments.)

Mocks

```
15  describe( name: 'mock math functions', () : void => {
16      test( name: 'mock add function', () : void => {
17          const mockAdd : Mock<number, [a: number, b: number], any> = jest.fn(add);
18          const result : number = mockAdd( a: 2, b: 3);
19          expect(result).toBe( expected: 5);
20          expect(mockAdd).toHaveBeenCalledWith(2, 3);
21          expect(mockAdd).toHaveBeenCalled();
22      });
23
24      test( name: 'mock multiply function', () : void => {
25          const mockMultiply : Mock<number, [a: number, b: number], any> = jest.fn(multiply);
26          const result : number = mockMultiply( a: 4, b: 5);
27          expect(result).toBe( expected: 20);
28          expect(mockMultiply).toHaveBeenCalledWith(4, 5);
29          expect(mockMultiply).toHaveBeenCalled();
30      });
31
32  }
33 );
```

Mocha

- installation: npm install --save-dev mocha ts-node typescript @types/mocha
- Test execution: npx mocha
- Good fit for legacy Node.js applications

simple, flexible, fun

pro	cons
flexible due to modular architecture (choose your own assertion/mocking libs)	No built-in assertions or mocking → requires additional setup
Broad Ecosystem Support strong community support, with many plugins and integrations available	More configuration, not newbie friendly
	Slightly slower

Vitest

- installation: npm install -D vitest
- Test execution: npx vitest
- Seamless integration with Vite projects (Vue, React)
- Tests must contain .test or .spec in their file name

pro	contra
Better support for modern JavaScript/TypeScript features	Younger -> smaller community
Developed TypeScript-first (Type Testing via expect-type)	Snapshot testing and watch mode not as mature
Extremly fast, Fast Watch Mode & Debugging	

Vitest

```ts
import { assert, expect, test } from 'vitest'
import { squared } from './basic'


test( name: 'Math.sqrt()', () :void => {  new *
  expect(Math.sqrt( x: 4)).toBe( expected: 2)
  expect(Math.sqrt( x: 144)).toBe( expected: 12)
  expect(Math.sqrt( x: 2)).toBe(Math.SQRT2)
})

test( name: 'Squared', () :void => {  new *
  expect(squared( n: 2)).toBe( expected: 4)
  expect(squared( n: 12)).toBe( expected: 144)
})

test( name: 'JSON', () :void => {  new *
  const input = {
    foo: 'hello',
    bar: 'world',
  }

  const output :string = JSON.stringify(input)

  expect(output).eq( value: '{"foo":"hello","bar":"world"}')
  assert.deepEqual(JSON.parse(output), input,  message: 'matches original')
})
```

Junit vs. TestNG

- Java-Testframeworks
- Mostly support Unit-und Integrationtests
- Annotion based testing (@Test, @BeforeEach, ...)
- Junit is available in modern Java projects, while TestNG must be explicitly included

functions

Feature	Junit(5.x)	TestNG
Annotations	@Test, @BeforeEach	@Test, @BeforeMethod
Testconfigs	Modern and easy	More flexible
Dependency Injection	Only extensions (@Mock)	✅
Parallel execution	Only with extern runner	✅
Community	Very active	ok
Testorder	no	✅ priority
Data-driven tests	@ParameterizedTest	@DataProvider
Assertions	Assertions.assertEquals()	Assert.assertEquals()

Integrationtests

Angular TestBed

- Interaction with DOM, events, and binding
- Services and mocks can be integrated specifically
- Always use async/await + compileComponents() for templated components
- Use fixture.detectChanges() to trigger lifecycle + DOM
- Use TestBed.inject() for services instead of new MyService(...)
- For multiple test suites: TestBed.resetTestingModule() in afterEach
- Use mocks for external dependencies (e.g., HTTP)

HttpTestingController

- Test class for intercepting HttpClient requests
- HttpClientTestingModule allows you to easily mock HTTP requests by providing the HttpTestingController service
- first need to import and provide them in your TestBed and the service we are testing
- Advantages:
  - Controls HTTP calls in the test
  - No real backend required
- Example:
  - req = httpMock.expectOne('/api/data'); req.flush(mockData);
- Testing Angular services and components with HTTP dependencies

Quarkustest

- @QuarkusTest: Full integration test with Quarkus runtime
- tests run within the Quarkus test container, so CDI, configuration, REST clients, database connections, etc. are available
- Ideal for Cloud-Native Java(Lightweight and optimized for Kubernetes & microservices)
- Dependency Injection in Tests (Tests can inject beans like in real runtime)
- @NativeImageTest: Test native compiled Quarkus apps

RestAssured

- Java DSL for testing REST APIs
- Ideal for integration testing with Quarkus
- Advantages:
  - Readable, flexible, checks status, headers, and body
- Example:
  - given().get("/api/configs").then().statusCode(200);
- Test entire endpoints and API flows

Testcontainers

- Java library for Docker containers in tests (e.g. DB, RabbitMQ)
- Perfect for testing with RabbitMQ, real DBs, etc.
- Advantages:
  - Realistic environment, no mocks required
  - Same setup locally & in CI
- Takes a long time and more complicated setup
- Example:

```
static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>( "postgres:16-alpine" );
```

E2E-/Systemtests

Cypress

- Installation: npm install cypress --save-dev
- npx cypress open

pro	contra
real browser environment, fast and stable	No native support for multi-tab or multi-window testing
Intuitive API and visual testing UI for debugging	Runs only within the browser context → limits some test cases
Automatic waiting and time-travel snapshots, easy debug	Only Supports Chromium-Based Browsers,No Real Mobile Device Testing
Comes with built-in test runner, assertions, mocking, stubbing, and reporting	

Playwright

- npm init playwright@latest
- npx playwright codegen url
- npx playwright test -ui/ npx playwright test --headed

pro	contra
E2E testing across multiple browsers (Chromium, Firefox, WebKit)	Not completly intuitive
multi-tab, multi-context, and mobile emulation	Debugging not as visually friendly
Fast and modern	Smaller community
No need for additional frameworks – Playwright Test handles assertions, reporting, and parallel execution.	

Selenium

Pro	contra
Long-standing and widely used with broad language and browser support	Slow
cross-browser testing	Maintenance-heavy
Large ecosystem and plugin support	Outdated API
Works with Java, Python, C#, JavaScript, and more	Poor integration with modern JavaScript frameworks

Cucumber

- Tests in natural language through gherkin syntax
- Step definition in js or ts
- Written in Gherkin feature files
- Structure: Given … When … Then …
- Can be combined with other test runners (e.g. Selenium, Cypress, Playwright)

pro	contra
Bridges communication between devs, testers, and business stakeholders	Maintenance gets complex with a large number of scenarios
readable syntax	Less technical: harder to debug
	Limited flexibility for dynamic or data-driven testing

Cucumber function + stepdefinition

```
function isItFriday(today) : string {
    if (today === "Friday") {
        return "TGIF";
    } else {
        return "Nope";
    }
}
```

```
12  Given('today is {string}', function (givenDay) : void {
13      this.today = givenDay;
14  });
15
16  When('I ask whether it\'s Friday yet', function () : void {
17      this.actualAnswer = isItFriday(this.today);
18  });
19
20  Then('I should be told {string}', function (expectedAnswer) : void {
21      assert.strictEqual(this.actualAnswer, expectedAnswer);
22  });
```

Cucumber featurefile

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario Outline: Today is or is not Friday
    Given today is "<day>"
    When I ask whether it's Friday yet
    Then I should be told "<answer>"

    Examples:
      | day            | answer |
      | Friday         | TGIF   |
      | Sunday         | Nope   |
      | anything else! | Nope   |
      | Monday         | Nope   |
```

Tips/ best practices

- Start Unit tests for core logic, later add integration and E2E tests
- Write 1–2 tests per feature directly while coding
- simple coverage reports (e.g. with --coverage)
- Tests in separate folders (__tests__, *.spec.ts) + clear naming conventions
- Automated integration of tests in GitHub Actions, GitLab CI, or similar

Stay In Sync

- Quarkus backend:
  - Junit + Mockito (Unit testing)
  - Junit + Quarkustest +(for rabbitMQ Testcontainers) (Integrationtesting)
  - RestAssured (System, backendE2E)
- Angular-Frontend:
  - Vitest(Unit testing)
  - Vitest + TestBed + HttpTestingController (Integrationtesting)
  - Cypress (E2E testing)

Demoprojekt with Quarkus and Angular

Conclusion

Tool	Unit	Integration	E2E	Ease of Setup	Maintainability	Performance	Flexibility	Community Support	TypeScript Support
Jest	✅	⚠️ (limited)	❌	✅	✅	✅	⚠️ (JS-focused)	✅ ✅	✅ ✅
Mocha	✅	⚠️ (needs extras)	❌	⚠️ (manual config)	⚠️ (boilerplate)	✅	✅	✅	⚠️ (requires setup)
Vitest	✅ ✅	⚠️ (with mocks)	❌	✅ ✅	✅	✅ ✅	⚠️ (Vite-based)	⚠️ (young ecosystem)	✅ ✅
Cypress	❌	⚠️ (via APIs)	✅ ✅	✅	✅	✅ (browser-based)	⚠️ (browser only)	✅ ✅	✅ ✅
Selenium	❌	⚠️ (possible)	✅	❌ (complex)	❌ (flaky tests)	❌	✅ ✅	✅	⚠️ (indirect)
Playwright	❌	⚠️ (via setup)	✅ ✅	✅	✅	✅ ✅	✅ ✅	✅	✅ ✅

Sources

- https://mochajs.org/
- https://jestjs.io/
- https://vitest.dev/
- https://www.cypress.io/
- https://playwright.dev/
- https://www.selenium.dev/
- https://cucumber.io/
- https://www.youtube.com/watch?v=YaXJeUkBe4Y&list=PLYlYyzg8PyTO2BhJcrAGYNkn1mD_cblWc
- https://www.youtube.com/watch?v=gA-uNj2FgdM&list=PLYlYyzg8PyTO2BhJcrAGYNkn1mD_cblWc&index=5
- https://www.youtube.com/watch?v=IDjF6-s1hGk&list=PLYlYyzg8PyTO2BhJcrAGYNkn1mD_cblWc&index=2
- https://www.youtube.com/watch?v=7f-71kYhK00
- https://angular.dev/
- https://testcontainers.com/guides/
- https://quarkus.io/guides/getting-started-testing