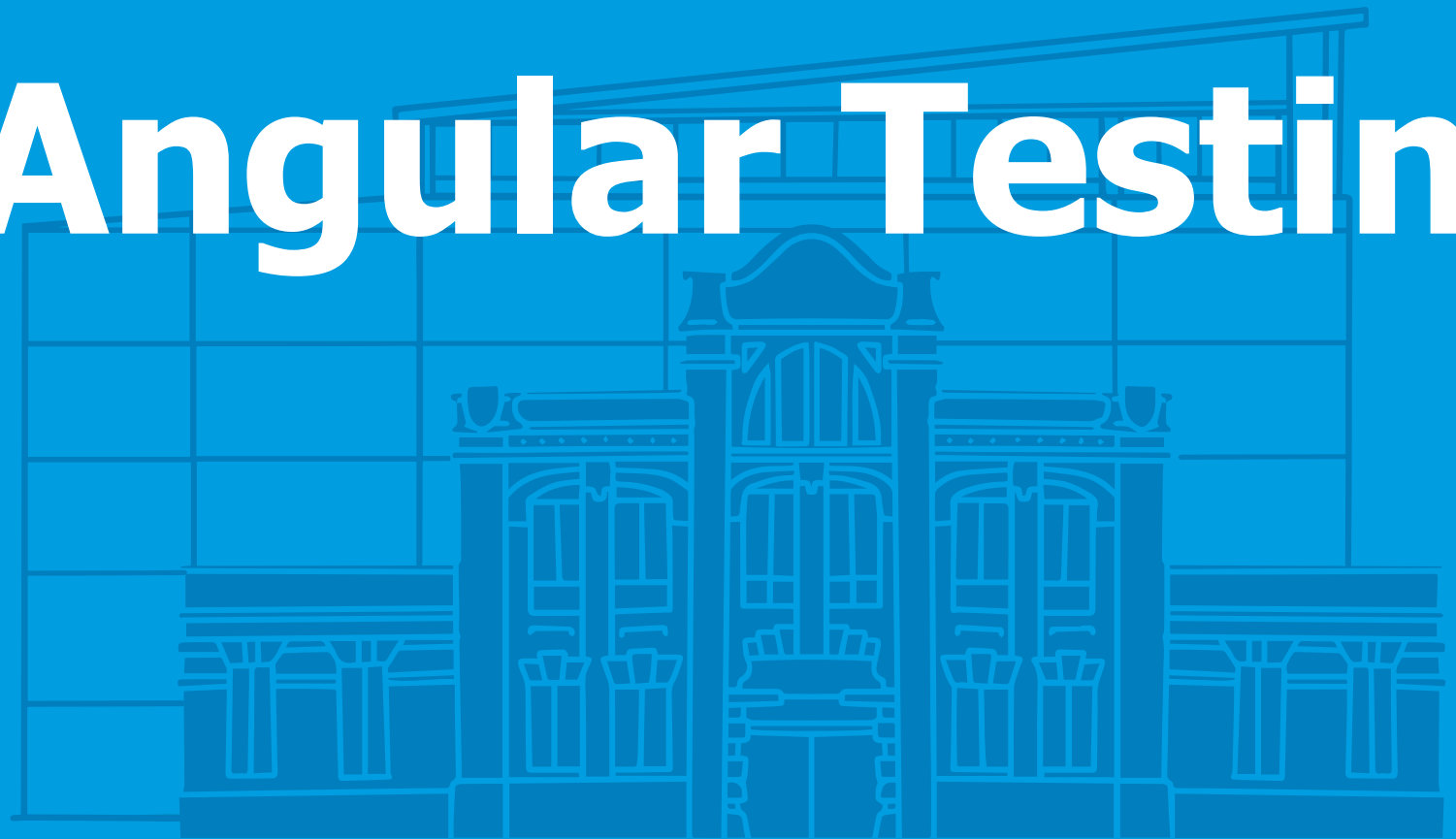
netmind
a bts company

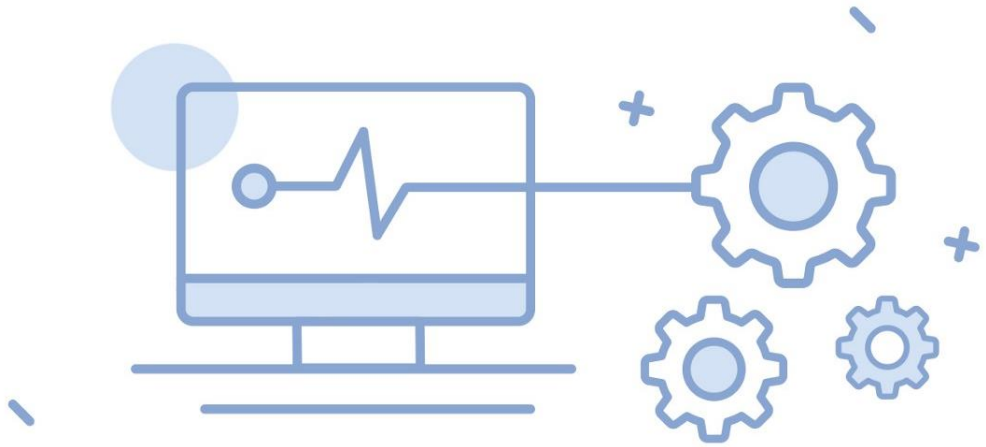
Angular 14 - 21

Angular Testing



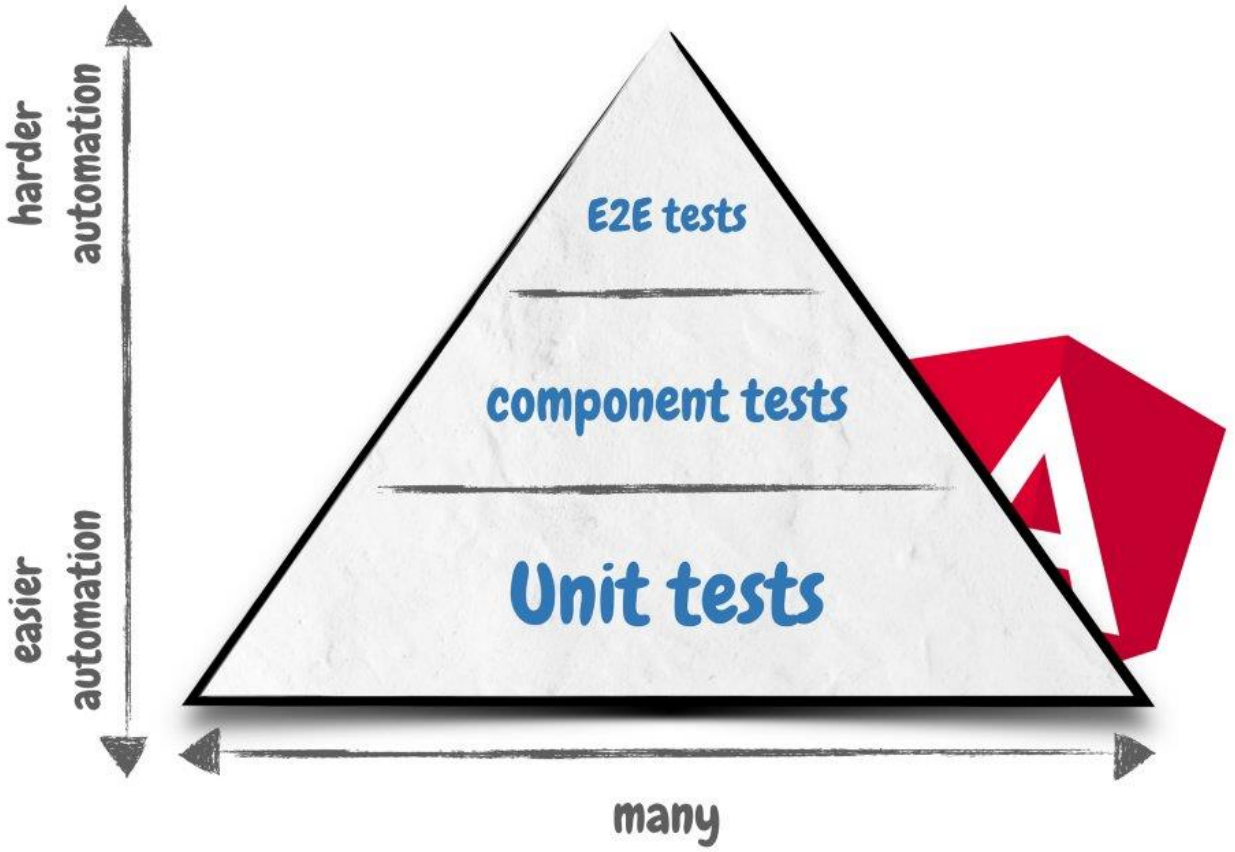
What makes a good test

- **COST-EFFECTIVE:** A valuable test is cost-effective. The test prevents bugs that could ultimately render the application unusable. The test is cheap to write compared to the potential damage it prevents.
- **DESCRIPTIVE:** A valuable test clearly describes how the implementation code should behave.
- **SUCCESS AND ERROR CASES:** A valuable test covers the important scenarios
- **PREVENT BREAKAGE:** A valuable test fails when essential code is changed or deleted.



The Angular testing pyramid

- **Unit tests** test functions
- **Component tests** are technical unit tests but actually they are integration tests - they test the interaction of multiple components
- **E2E Tests** test the most important scenarios



Angular testing principles

- **Testability**
  - **Testable architecture:** all application parts can be tested easily in a similar way..
  - **Well-structured code:** Angular breaks code into smaller chunks that "do one thing and do it well".
- **Dependency injection and faking**
  - **Loose coupling:** Dependency injection turns tight coupling into loose coupling.
  - **Original or fake:** In our test, we can decide how to deal with a dependency:
    - We can either provide an original, **fully-functional implementation**. In this case, we are writing an integration test that includes direct and indirect dependencies.
    - Or we provide a **fake implementation** that does not have side effects.
- **Testing tools**
  - The Angular team already made decisions for us:
  - **Jasmine** as testing framework and **Karma** as test runner.
  - Implementation and test code is bundled with **Webpack**.
  - Application parts are typically tested inside Angular's **TestBed**.
  - For e2e, we can choose between **Protractor** or **Cypress**.

Angular testing principles

- **Running the unit and integration tests**

For running all:

```
ng test
```

For running only one:

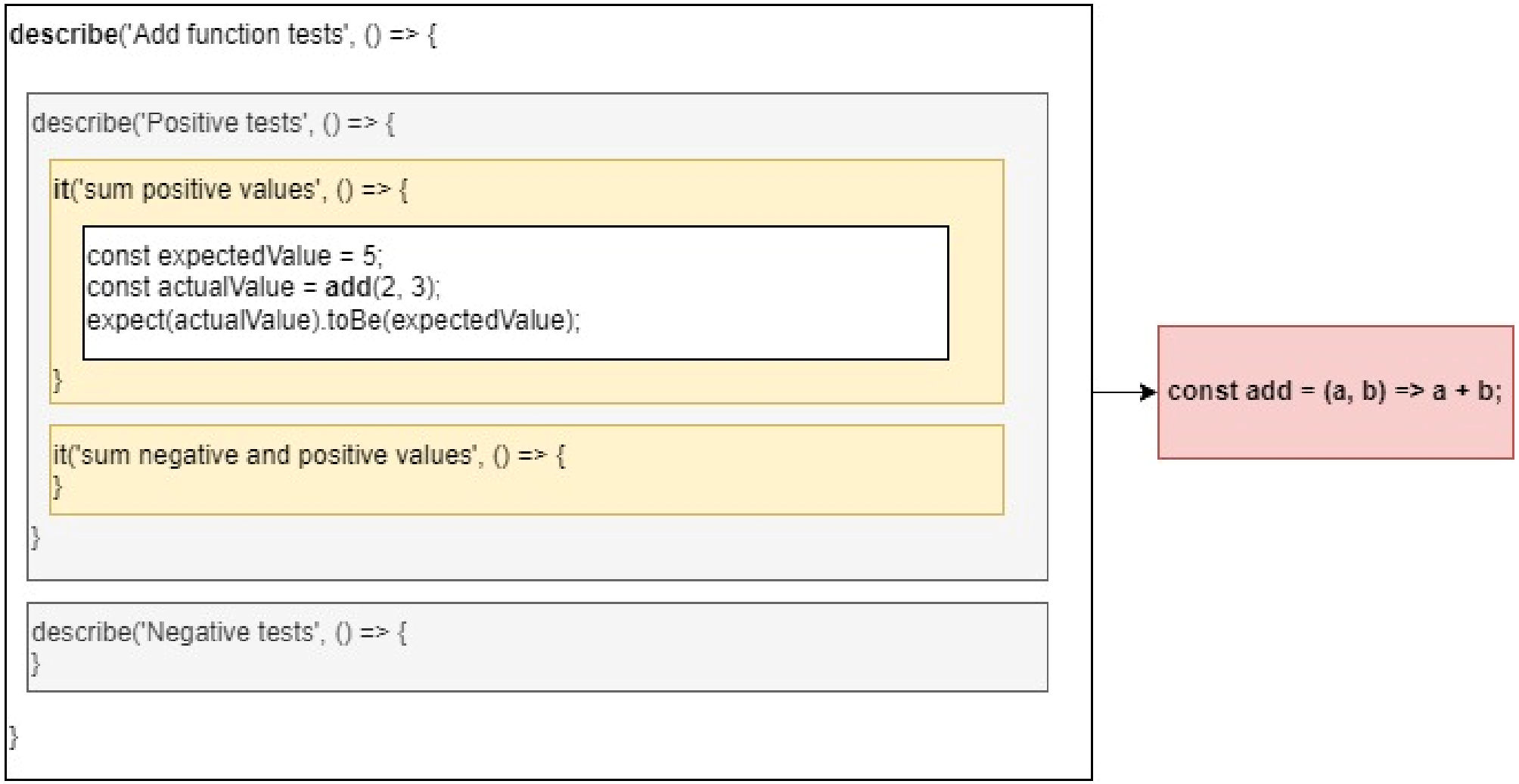
```
ng test --include [path_totest]
```

- ng test uses Webpack to compile the code into a JavaScript bundle. The entry point for the bundle is **src/test.ts**.
- Each **.spec.ts** file represents a test. Typically, it contains at least one Jasmine test suite. The files are co-located with the implementation code.
- ng test launches **Karma**, the test runner. Karma starts a development server at **http://localhost:9876/** that serves the JavaScript bundles compiled by Webpack.
- Test runner launches the browser and navigates to http://localhost:9876/, where the tests result are shown.
  - We enter in a "**red-green**" feedback cycle.
- https://angular.io/cli/test

Test suites with Jasmine

- Angular ships with Jasmine, a JavaScript framework that enables you to write and execute unit and integration tests.
  - https://jasmine.github.io/index.html
- Jasmine consists of three important parts:
  - A library with classes and functions for constructing tests.
  - A test execution engine.
  - A reporting engine that outputs test results in different formats.
- In Jasmine, a test consists of one or more **suites**. They can be nested.
- Each suit consists of one or more specifications, or **specs**. A spec is declared with an **it** block.
- Inside the it block lies the actual **testing code**, that typically consists of three phases:
  - **Arrange** is the preparation and setup phase. For example, the class under test is instantiated. Dependencies are set up. Spies and fakes are created.
  - **Act** is the phase where interaction with the code under test happens. For example, a method is called or an HTML element in the DOM is clicked.
  - **Assert** is the phase where the code behavior is checked and verified. For example, the actual output is compared to the expected output.
    - Jasmine allows us to create expectations in an easier manner using the **expect** function together with a **matcher** (https://jasmine.github.io/api/edge/matchers.html).

Test suites with Jasmine



- You can find most details in the official tutorial: https://jasmine.github.io/tutorials/your_first_suite
- And in this Jasmine cheatsheet: https://devhints.io/jasmine

Test suites with Jasmine

- When writing multiple specs in one suite, usually the Arrange phase is similar or even identical across these specs.
- For this purpose, Jasmine provides four functions: **beforeEach**, **afterEach**, **beforeAll** and **afterAll**. They are called inside of a describe block.

```
describe('Suite description', () => {
  beforeAll(() => {
    console.log('Called before all specs are run');
  });
  afterAll(() => {
    console.log('Called after all specs are run');
  });

  beforeEach(() => {
    console.log('Called before each spec is run');
  });
  afterEach(() => {
    console.log('Called after each spec is run');
  });

  it('Spec 1', () => {
    console.log('Spec 1');
  });
  it('Spec 2', () => {
    console.log('Spec 2');
  });
});
```

Faking dependencies

- When testing a piece of code, you need to decide between an **integration** test and a **unit** test.
  - To recap, the **integration** test **includes ("integrates") the dependencies**.
  - In contrast, the **unit** test **replaces the dependencies with fakes** (also called test doubles, stubs or mocks) in order to isolate the code under test.
- Jasmine provides the **Jasmine spy** for replacing a function dependency:

  `const spy = jasmine.createSpy('name');`

```
// Fake todos and response object
const todos = [
  'shop groceries',
  'mow the lawn',
  'take the cat to the vet'
];
const okResponse = new Response(JSON.stringify(todos), {
  status: 200,
  statusText: 'OK',
});
```

```
class TodoService {
  constructor(
    private fetch = window.fetch.bind(window)
  ) {}

  public async getTodos(): Promise<string[]> {
    const response = await this.fetch('/todos');
    if (!response.ok) {
      throw new Error(
        `HTTP error: ${response.status} ${response.statusText}`
      );
    }
    return await response.json();
  }
}
```

```
describe('TodoService', () => {
  it('gets the to-dos', async () => {
    // Arrange
    const fetchSpy = jasmine.createSpy('fetch')
      .and.returnValue(okResponse);
    const todoService = new TodoService(fetchSpy);

    // Act
    const actualTodos = await todoService.getTodos();

    // Assert
    expect(actualTodos).toEqual(todos);
    expect(fetchSpy).toHaveBeenCalledWith('/todos');
  });
});
```

Faking dependencies

- Sometimes, there is already an object whose method we need to spy on.
- This is especially helpful if the code uses global methods from the browser environment, like window.fetch in the example above.
- For this purpose, we can use the spyOn method:

```
spyOn(object, 'method');
```

```
describe('TodoService', () => {
  it('gets the to-dos', async () => {
    // Arrange
    spyOn(window, 'fetch')
      .and.returnValue(okResponse);
    const todoService = new TodoService();

    // Act
    const actualTodos = await todoService.getTodos();

    // Assert
    expect(actualTodos).toEqual(todos);
    expect(window.fetch).toHaveBeenCalledWith('/todos');
  });
});
```

Testing components

- Several chores are necessary to render a Component in Angular, even the simple counter Component.
- Angular team provides the **TestBed** to ease unit testing.
- The TestBed creates and configures an Angular environment so you can test particular application parts like Components and Services safely and easily.

```
TestBed.configureTestingModule({
  imports: [ /*… */ ],
  declarations: [ /*… */ ],
  providers: [ /*… */ ],
});
```

```
describe('CounterComponent', () => {
  let fixture: ComponentFixture<MyComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [MyComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(MyComponent);
    fixture.detectChanges();
  });

  it('…', () => {
    /* … */
  });
});
```

Testing components

- For accessing elements in the DOM, Angular has another abstraction: The **DebugElement** wraps the native DOM element.
- The fixture's debugElement property returns the Component's host element.

```
const { debugElement } = fixture;
```

- Often it is necessary to unwrap the DebugElement to access the native DOM element inside. Every DebugElement has a nativeElement property:

```
const { nativeElement } = debugElement;
```

```
/* Incomplete! */
describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let debugElement: DebugElement;

  // Arrange
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [CounterComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    fixture.detectChanges();
    debugElement = fixture.debugElement;
  });

  it('increments the count', () => {
    // Act
    const incrementButton = debugElement.query(
      By.css('[data-testid="increment-button"]')
    );
    incrementButton.triggerEventHandler('click', null);

    // Assert
    const countOutput = debugElement.query(
      By.css('[data-testid="count"]')
    );
    expect(countOutput.nativeElement.textContent).toBe('1');
  });
});
```

Testing helper

- A testing helper is a piece of code that makes writing tests easier. It makes test code more concise and more meaningful.
- Since a spec should describe the implementation, a readable spec is better than an obscure, convoluted one.

```
function findEl<T>(
  fixture: ComponentFixture<T>,
  testId: string
): DebugElement {
  return fixture.debugElement.query(
    By.css(`[data-testid="${testId}"]`)
  );
}
```

```
export function click<T>(
  fixture: ComponentFixture<T>,
  testId: string
): void {
  const element = findEl(fixture, testId);
  const event = makeClickEvent(element.nativeElement);
  element.triggerEventHandler('click', event);
}
```

```
export function makeClickEvent(
  target: EventTarget
): Partial<MouseEvent> {
  return {
    preventDefault(): void {},
    stopPropagation(): void {},
    stopImmediatePropagation(): void {},
    type: 'click',
    target,
    currentTarget: target,
    bubbles: true,
    cancelable: true,
    button: 0
  };
}
```

```
export function expectText<T>(
  fixture: ComponentFixture<T>,
  testId: string,
  text: string,
): void {
  const element = findEl(fixture, testId);
  const actualText = element.nativeElement.textContent;
  expect(actualText).toBe(text);
}
```

```
it('decrements the count', () => {
  // Act
  click(fixture, 'decrement-button');
  // Re-render the Component
  fixture.detectChanges();

  // Assert
  expectText(fixture, 'count', '-1');
});
```

Fake input event

- Angular forms cannot observe value changes directly. Instead, **Angular listens for an input event** that the browser fires when a field value changes.
- For compatibility with Template-driven and Reactive Forms, we need to dispatch a fake input event. Such events are also called synthetic events.
- We create a fake input event with new **Event('input')**. To dispatch the event, we use the **dispatchEvent** method of the target element.

```
const resetInputEl = findEl(fixture, 'reset-
input').nativeElement;
resetInputEl.value = '123';
resetInputEl.dispatchEvent(new Event('input'));
```

```
it('resets the count', () => {
  const newCount = '123';

  // Act
  const resetInputEl = findEl(fixture, 'reset-
input').nativeElement;
  // Set field value
  resetInputEl.value = newCount;
  // Dispatch input event
  const event = document.createEvent('Event');
  event.initEvent('input', true, false);
  resetInputEl.dispatchEvent(event);

  // Click on reset button
  click(fixture, 'reset-button');
  // Re-render the Component
  fixture.detectChanges();

  // Assert
  expectText(fixture, 'count', newCount);
});
```

Component state

- We can test the component state related to events in the view.
- For that we will lean on **ngOnChanges**, since it is called whenever a "data-bound property" changes, including Inputs.

```
describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;

  const startCount = 123;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [CounterComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    component.startCount = startCount;
    // Call ngOnChanges, then re-render
    component.ngOnChanges();
    fixture.detectChanges();
  });

  /* … */

  it('shows the start count', () => {
    expectText(fixture, 'count', String(startCount));
  });
});
```

Testing Outputs

- While Inputs pass data from parent to child, Outputs send data from child to parent. In combination, a Component can perform a specific operation just with the required data.
- The Component uses the **emit** method to publish new values. The parent Component uses the **subscribe** method to listen for emitted values.
- In the testing environment, we will do the same.

```
it('emits countChange events on decrement', () => {
  // Arrange
  let actualCount: number | undefined;
  component.countChange.subscribe((count: number) => {
    actualCount = count;
  });

  // Act
  click(fixture, 'decrement-button');

  // Assert
  expect(actualCount).toBe(-1);
});

it('emits countChange events on reset', () => {
  const newCount = '123';

  // Arrange
  let actualCount: number | undefined;
  component.countChange.subscribe((count: number) => {
    actualCount = count;
  });

  // Act
  setFieldValue(fixture, 'reset-input', newCount);
  click(fixture, 'reset-button');

  // Assert
  expect(actualCount).toBe(newCount);
});
```

Testing Components with children

- There are two fundamental ways to test Components with children:
  - A unit test using **shallow** rendering. The child Components are not rendered.
  - An integration test using **deep** rendering. The child Components are rendered.
- When configuring the testing Module, we can specify **schemas** to tell Angular how to deal with elements that are not handled by Directives or Components.

```
await TestBed.configureTestingModule({
  declarations: [HomeComponent],
  schemas: [NO_ERRORS_SCHEMA],
}).compileComponents();
```

- There is a middle ground between a naive unit test and an integration test. Instead of working with empty custom elements, we can render **fake child Components**.

```
@Component({
  selector: 'app-counter',
  template: '',
})
class FakeCounterComponent implements Partial<CounterComponent> {
  @Input()
  public startCount = 0;

  @Output()
  public countChange = new EventEmitter<number>();
}

describe('HomeComponent (faking a child Component)', () => {
  let fixture: ComponentFixture<HomeComponent>;
  let component: HomeComponent;
  let counter: FakeCounterComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [HomeComponent, FakeCounterComponent],
      schemas: [NO_ERRORS_SCHEMA],
    }).compileComponents();

    fixture = TestBed.createComponent(HomeComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();

    const counterEl = fixture.debugElement.query(
      By.directive(FakeCounterComponent)
    );
    counter = counterEl.componentInstance;
  });

  it('renders an independent counter', () => {
    expect(counter).toBeTruthy();
  });

  it('passes a start count', () => {
    expect(counter.startCount).toBe(5);
  });

  it('listens for count changes', () => {
    spyOn(console, 'log');
    const count = 5;
    counter.countChange.emit(count);
    expect(console.log).toHaveBeenCalledWith(
      'countChange event from CounterComponent',
      count,
    );
  });
});
```

Testing Services

- Testing a service is similar to testing a generic class. TestBed is not needed, but instantiate the service object to be tested.

```
describe('CounterService', () => {
  let counterService: CounterService;

  function expectCount(count: number): void {
    let actualCount: number | undefined;
    counterService.getCount().pipe(first())
      .subscribe((actualCount2) => {
        actualCount = actualCount2;
      });
    expect(actualCount).toBe(count);
  }

  beforeEach(() => {
    counterService = new CounterService();
  });

  it('returns the count', () => {
    expectCount(0);
  });

  it('increments the count', () => {
    counterService.increment();
    expectCount(1);
  });

  it('decrements the count', () => {
    counterService.decrement();
    expectCount(-1);
  });

  it('resets the count', () => {
    const newCount = 123;
    counterService.reset(newCount);
    expectCount(newCount);
  });
});
```

Testing Services

- When a service depends on other services, specially on Angular's standard HTTP library, we will need to instantiate the dependencies and subscribe if there are asynchronous events.
- We can inject the depending services in the module using the TestBed.
- **HttpTestingController** will help to deal with requests.

```
const searchTerm = 'dragonfly';
const expectedUrl =
`https://www.flickr.com/services/rest/?tags=${searchTerm}&metho
d=flickr.photos.search&format=json&nojsoncallback=1&tag_mode=al
l&media=photos&per_page=15&extras=tags,date_taken,owner_name,ur
l_q,url_m&api_key=XYZ`;

describe('FlickrService', () => {
  let flickrService: FlickrService;
  let controller: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [FlickrService],
    });
    flickrService = TestBed.inject(FlickrService);
    controller = TestBed.inject(HttpTestingController);
  });

  it('searches for public photos', () => {
    let actualPhotos: Photo[] | undefined;
    flickrService.searchPublicPhotos(searchTerm).subscribe(
      (otherPhotos) => {
        actualPhotos = otherPhotos;
      }
    );

    const request = controller.expectOne(expectedUrl);
    request.flush({ photos: { photo: photos } });
    controller.verify();

    expect(actualPhotos).toEqual(photos);
  });
});
```

Code coverage

- Code coverage, also called test coverage, tells which parts of the code are executed by running the unit and integration tests.
  - Code coverage is typically expressed as percent values, for example, 79% statements, 53% branches, 74% functions, 78% lines.
- In Angular's Karma and Jasmine setup, **Istanbul** is used for measuring test coverage.
  - https://istanbul.js.org/
- To activate Istanbul when running the tests, add the --code-coverage parameter:

```
ng test --code-coverage
```

  - After the tests have completed, Istanbul saves the report in the **coverage directory** located in the Angular project directory.
  - The report is a bunch of HTML files you can open with a browser.
  - Start by opening **coverage/index.html**.

Let's put it into practice: Tasks/Projects App

- Implement tests for the different elements of the application (at least a nested component and service)
    - Use the fake approach and the integration testing approach.
- Get the report of code coverage.

E2e testing Angular

- An end-to-end test mimics how a user interacts with the application. Typically, the test engine launches an ordinary browser and controls it remotely.
- There are two main frameworks to do e2e testing in the context of Angular:
  - **Protractor**, which is based on WebDriver.
  - **Cypress**, which does not use WebDriver.
- Up until Angular version 12, Protractor was the default end-to-end testing framework in projects created with Angular CLI.
  - Since Angular 12, Protractor is deprecated.
  - In new CLI projects, there is no default end-to-end testing solution configured.
  - The main reason for Protractor's deprecation is that it was not maintained for years.
- Instead, now is recommended using **Cypress** for e2e testing Angular applications.
  - https://docs.cypress.io/
  - Cypress is an end-to-end testing framework that is not based on WebDriver.
  - There are no Angular-specific features. Any web site can be tested with Cypress.
  - Cypress is well-maintained and well-documented.
- To add Cypress to the project we can use the next command:

```
ng add @cypress/schematic
```

E2e testing with Cypress

- In the project directory, you will find a sub-directory called cypress. It contains:
  - **tsconfig.json** configuration for all **TypeScript files** specifically in this directory,
  - **e2e** directory for the **end-to-end tests**,
  - **support** directory for **custom commands** and other testing helpers,
  - **fixtures** directory for **test data**.
- The test files reside in the e2e directory. Each test is TypeScript file with the **extension .cy.ts**.
- The tests itself are structured with the test framework **Mocha**. The assertions (also called expectations) are written using **Chai**.
- Mocha structure is similar to Jasmine.

```
describe('… Feature description …', () => {
  beforeEach(() => {
    // Navigate to the page
  });

  it('… User interaction description …', () => {
    // Interact with the page
    // Assert something about the page content
  });
});
```

E2e testing with Cypress

- Cypress commands are methods of the **cy** namespace object. Here, we are using two commands, visit and title.
- You can review the reference for commands, chainers and assertions here:
  - https://docs.cypress.io/api/table-of-contents
- Cheatsheet:
  - https://cheatography.com/aiqbal/cheat-sheets/cypress-io/

```
describe('Counter', () => {
  beforeEach(() => {
    cy.visit('/');
  });

  it('has the correct title', () => {
    cy.title().should('equal', 'Angular Workshop: Counters');
  });

  it('increments the count', () => {
    cy.get('[data-testid="count"]').should('have.text', '5');
    cy.get('[data-testid="increment-button"]').click();
    cy.get('[data-testid="count"]').should('have.text', '6');
  });

  it('decrements the count', () => {
    cy.get('[data-testid="decrement-button"]').click();
    cy.get('[data-testid="count"]').should('have.text', '4');
  });

  it('resets the count', () => {
    cy.get('[data-testid="reset-input"]').type('123');
    cy.get('[data-testid="reset-button"]').click();
    cy.get('[data-testid="count"]').should('have.text', '123');
  });
});
```

- Cypress has two shell commands to run the end-to-end tests:

🔗 **Non-interactive test runner:** Runs the tests in a “headless” browser. This means the browser window is not visible.

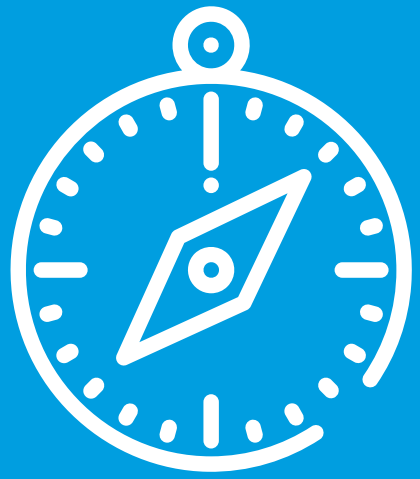
```
npx cypress run
```

🔗 **Interactive test runner:** Opens a window where you can select which browser to use and which tests to run.

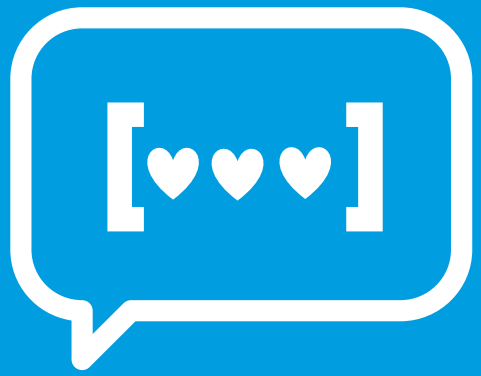
```
npx cypress open
```

Let's put it into practice: Tasks/Projects App

- Implement e2e tests for the project list, for deleting a task and for creating a new project.



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: