GUIDA AL TESTING IN APPLICAZIONI REACT

Indice

1. Introduzione
2. Tipi di Test
3. Jest
---3.1 Cos'è Jest
---3.2 Installazione
---3.3 Configurazione
---3.4 Funzionalità principali
---3.5 Esempi pratici
4. React Testing Library
---4.1 Cos'è
---4.2 Installazione
---4.3 API principali
---4.4 Esempi pratici
5. Playwright
---5.1 Cos'è Playwright
---5.2 Installazione
---5.3 Configurazione
---5.4 Comandi
---5.5 Esempi pratici
---5.6 Avanzato: Testing API + Fixtures + POM
------5.6.1 Fixture personalizzate
------5.6.2 POM
------5.6.3 Esempio test API
6. Best Practice
7. Errori comuni
8. Typescript & Testing
---8.1 Integrazione
---8.2 Esempi con Jest
---8.3 Esempi con React Testing Library
---8.4 Esempi con Playwright
---8.5 Errori comuni
---8.6 Conclusione

1. INTRODUZIONE

Quando un applicazione o programma è giunto al termine del suo sviluppo e quindi risulta completa, inizia la sua vita in produzione e della sua manutenzione futura. Per garantire che un'applicazione sia robusta, manutenibile e affidabile, esistono pratiche fondamentali che ogni sviluppatore dovrebbe adottare. Una di queste, spesso paragonata alla documentazione per la sua importanza, è il testing. Scrivere test è una pratica integrante e continua che può portare dei benefici, come :

- Sviluppare con fiducia
I test permettono di modificare il codice esistente, aggiungere delle funzionalità o fare aggiornamenti, con la certezza che se i test passano, le funzionalità principali dell'applicazione non sono state compromesse.

- Previene le regressioni
Quando una modifica al codice rompe una funzionalità che prima funzionava, si parla di "regressione". I test che falliscono possono aiutarci a rilevare dei bug che ci sono sfuggiti.

- Documentazione viva
Una buona suite di test può fungere anche come forma di documentazione. Mostra come un componente o una funzione dovrebbero essere usati o quale output aspettarsi. A differenza della documentazione statica, non può diventare obsoleta rispetto al codice dato che se cambia bisogna aggiornare anche i test.

2. TIPI DI TEST
In questa guida, esploreremo le strategia e gli strumenti essenziali per testare applicazioni React in modo efficace. Partendo dai Test Unitari e Test di Integrazione fino ad arrivare ai Test End-to-End.
Utilizzando librerie standard come Jest, (React) Testing Library e Playwright.

* Unit Test (test unitari) : testano una singola unità di codice come una funzione o un componente React isolato
* Integration Test (test di integrazione) : verificano il funzionamento di più unità che collaborano come un form completo con validazione.
* E2E Test (test end-to-end) : simulano l'interazione dell'utente con l'app, testando il sistema completo.

Per testare le nostre applicazioni React, useremo principalmente due librerie che lavorano in perfetta sinergia, Jest e React Testing Library che ci permettono di scrivere sia test unitari che di integrazione.

3. JEST

3.1 COS'È JEST
Jest è un framework di testing, ci fornisce l'infrastruttura per eseguire i test, creare simulazioni (mock), fare asserzioni (con expect) e controllare la copertura (coverage).

3.2 INSTALLAZIONE
npm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-renderer

Perché serve babel-jest in un progetto React?
Quando usi:
* JSX (<Component />)
* import/export
* optional chaining, nullish coalescing, ecc.

Jest da solo (senza Babel o un altro transpiler) non è in grado di interpretare questo codice, perché Node non supporta tutte queste feature (funzionalità/caratteristiche) in modo nativo o allo stesso modo del browser.

Cosa fa esattamente babel-jest?

Legge la tua configurazione Babel (babel.config.js o simili)
Usa i preset e i plugin (es. @babel/preset-env, @babel/preset-react, @babel/preset-typescript) per trasformare il codice
Restituisce il codice transpile-ato (risultato della trasformazione) a Jest che può eseguirlo e testarlo

Se usi TypeScript, installa anche i tipi:

```
npm install --save-dev @types/jest

npm install --save-dev ts-jest

npm install --save-dev @babel/preset-typescript
```

3.3 CONFIGURAZIONE
Configurare Jest (a livello di root, quindi fuori da /src).

Jest funziona senza configurazione in Create React App, ma in progetti custom (Vite, Webpack, ecc.) è utile creare dei file di configurazione:

* jest.config.js :

```
module.exports = {
 preset: "ts-jest",
 testEnvironment: "jsdom",
 transform: {
   "^.+\\.(js|jsx|ts|tsx)$": "babel-jest",
   "^.+\\.(ts|tsx)$": "ts-jest",
 },
 setupFilesAfterEnv: \["@testing-library/jest-dom", "./jest.setup.js"],
 testPathIgnorePatterns: \[
   "/node\_modules/",
   "<rootDir>/src/backup/",
   "./tests",
 ],
 moduleFileExtensions: \["ts", "tsx", "js", "jsx", "json", "node"],
};
```

* setupTests.js/ts :

```
import '@testing-library/jest-dom';
```

Jest supporta Babel, che verrà utilizzato per trasformare i file in JS validi, in base alla configurazione di Babel. Per impostazione predefinita, la cartella "node_modules" viene ignorata dai trasformatori, per questo motivo abbiamo bisogno di configurare babel.

* babel.config.js :

```
module.exports = {
   presets: \[
     '@babel/preset-env',
     \['@babel/preset-react', {runtime: 'automatic'}],
     '@babel/preset-typescript',
   ],
 };
```

Ora Jest è pronto per eseguire i test. Aggiungi questo script nel file package.json:

```
{
 "scripts": {
   "test": "jest"
 }
}
```

Ora puoi eseguire i test con:

```
npm test
```

Questo comando eseguirà tutti file.test.js/jsx/ts/tsx lanciando ogni singolo test scritto e indicando quali sono passati o errati.

3.4 Funzionalità Principali

* Runner di test: esegue i test e genera report.
* Asserzioni: expect(value).toBe(expectedValue).
* Mocking: jest.fn(), jest.mock() per simulare funzioni e moduli.
* Snapshot: confronta il rendering con una versione salvata.
* Code Coverage: misura la copertura del codice testato (jest --coverage).

Pattern Arrange-Act-Assert (AAA)
Per scrivere test chiari e leggibili, segui il pattern AAA:
* Arrange → Prepara l'ambiente e i dati necessari per il test.
* Act → Esegui l'azione da testare (click, chiamata API, ecc.).
* Assert → Verifica che il risultato sia quello atteso.

3.5 ESEMPI PRATICI

* Test di una funzione semplice :

```
// somma.js
export function somma(a, b) { return a + b; }

// somma.test.js
import { somma } from './somma';
test('somma correttamente due numeri', () => {
 // Arrange: preparo i dati di input
 const a = 2;
 const b = 3;

 // Act: eseguo la funzione da testare
 const result = somma(a, b);

 // Assert: verifico il risultato atteso
 expect(result).toBe(5);
});
```

* Mock di una funzione :

```
test('chiama la funzione di callback', () => {
 // Arrange: creo un mock di funzione
 const mockFn = jest.fn();
 // Act: chiamo la funzione
 mockFn();
 // Assert: verifico che sia stata chiamata
 expect(mockFn).toHaveBeenCalled();
})
```

* Snapshot testing :

```
import { render } from '@testing-library/react';
import MyComponent from './MyComponent';

test('il componente corrisponde allo snapshot', () => {
 const { container } = render(<MyComponent />);
```

```
 expect(container).toMatchSnapshot();
});
```

4. REACT TESTING LIBRARY

4.1 COS'È TESTING LIBRARY

React testing library è una libreria che permette di testare i componenti come se fossero usati da un vero utente, come cliccare pulsanti, cercare testo, leggere un campo ecc.. Questo strumento ti spinge a scrivere test che verificano il comportamento dell'applicazione senza concentrarti sui dettagli interni dei componenti. Utilizza selettori simili a quelli degli utenti (getByText, findByRole). Simula interazioni con fireEvent.

4.2 INSTALLAZIONE

```
npm install --save-dev @testing-library/react @testing-library/dom
```

@testing-library/react: libreria principale per testare componenti React

@testing-library/dom: utile per testare DOM non React

Se usi TypeScript, installa anche i tipi:

```
npm install --save-dev @testing-library/react @testing-library/dom @types/react
@types/react-dom
```

4.3 API PRINCIPALI

- render(component) : renderizza il componente da testare.
- screen : contiene tutte le query :
    - getByText, getByRole, getByTestId
    - findBy* (query asincrona utilizzabile con le fetch)
    - queyBy* (utile per assenza di elementi essendo asincrona)
- fireEvent : simula interazioni reali come click, input, tab ecc...
- waitFor : aspetta aggiornamenti asincroni.

4.4 ESEMPI PRATICI

1.

```jsx
   // Button.test.js
   import React from 'react';
   import { render, screen, fireEvent } from '@testing-library/react';
   import '@testing-library/jest-dom/extend-expect';
   import Button from './Button';
```

```
describe("MyButton", () => {
   test('renders the button with the correct label', () => {
     render(<Button label="Click me" />);
     const buttonElement = screen.getByText(/click me/i);
     expect(buttonElement).toBeInTheDocument();
   });

   test('calls onClick when the button is clicked', () => {
     const handleClick = jest.fn();
     render(<Button label="Click me" onClick={handleClick} />);
     const buttonElement = screen.getByText(/click me/i);
     fireEvent.click(buttonElement);
     expect(handleClick).toHaveBeenCalledTimes(1);
   });
});
```

2.
```jsx
import { render, screen, waitFor } from '@testing-library/react';
import MyComponent from './MyComponent';
import { fetchData } from './api';

jest.mock('./api'); // Mockiamo la funzione API

describe("My Component", () => {
test('mostra i dati dopo la chiamata API', async () => {
  // Arrange: imposto il mock della chiamata API
  fetchData.mockResolvedValue({ data: 'Testo caricato' });

  // Act: renderizzo il componente
  render(<MyComponent />);

  // Assert: verifico che prima mostri un caricamento
  expect(screen.getByText(/caricamento/i)).toBeInTheDocument();

  // Assert: verifico che il testo corretto venga visualizzato dopo la chiamata
API
  await waitFor(() => {
    expect(screen.getByText(/testo caricato/i)).toBeInTheDocument();
  });
});
});
```

5. PLAYWRIGHT

5.1 COS'è PLAYWRIGHT
È una libreria moderna per test end-to-end che permette di simulare il comportamento reale dell'utente su un'applicazione web, interagendo con la UI tramite browser reali (Chromium, Firefox).

5.2 INSTALLAZIONE
L'installazione della libreria può essere fatta in 2 modi :

- MANUALE DA CLI : utilizzando 'npm init playwright@latest' e una volta eseguito il comando sarà possibile scegliere tra javascript o typescript, il nome della cartella test e l'installazione del browser Playwright. Dopodiché playwright scaricherà i browser necessari e creerà i seguenti file :

```
playwright.config.ts
package.json
package-lock.json
tests/
  example.spec.ts
tests-examples/
  demo-todo-app.spec.ts
```

Il file playwright.config è dove puoi aggiungere la configurazione per Playwright, inclusa la modifica dei browser su cui desideri eseguirlo. Se esegui test all'interno di un progetto già esistente, le dipendenze verranno aggiunte direttamente al tuo file package.json.
La cartella tests contiene un test di esempio di base per aiutarti a iniziare a testare. Per un esempio più dettagliato, consulta la tests-examplescartella che contiene i test scritti per testare un'app di attività.

- UTILIZZANDO L'ESTENSIONE : Playwright ha un'estensione per VS Code disponibile per i test con Node.js. Installala dal marketplace di VS Code o dalla scheda estensioni di VS Code.Una volta installato, apri il pannello dei comandi e digita
Install Playwright . Seleziona "Test: Installa Playwright" e scegli i browser su cui desideri eseguire i test. Questi possono essere configurati in seguito nel file playwright.config . La barra laterale di test può essere aperta cliccando sull'icona di test nella barra delle attività. Questo vi darà accesso all'esploratore di test, che vi mostrerà tutti i test del vostro progetto, nonché alla barra laterale di Playwright, che include progetti, impostazioni, strumenti e configurazione. Puoi eseguire un singolo test cliccando sul triangolo verde accanto al blocco di test. Playwright eseguirà ogni riga del test e, al termine, vedrai un segno di spunta verde accanto al blocco di test e il tempo impiegato per eseguirlo.

5.3 CONFIGURAZIONE
Playwright crea un file di configurazione playwright.config.ts/js. Esempio base:

```
import { defineConfig, devices } from "@playwright/test";

/**
 * Read environment variables from file.
 * https://github.com/motdotla/dotenv
 */
// import dotenv from 'dotenv';
```

```
// import path from 'path';
// dotenv.config({ path: path.resolve(__dirname, '.env') });

/**
 * See https://playwright.dev/docs/test-configuration.
 */
export default defineConfig({
  testDir: "./tests",
  /* Run tests in files in parallel */
  fullyParallel: true,
  /* Fail the build on CI if you accidentally left test.only in the source code.
*/
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* Opt out of parallel tests on CI. */
  workers: process.env.CI ? 1 : undefined,
  /* Reporter to use. See https://playwright.dev/docs/test-reporters */
  reporter: "html",
  /* Shared settings for all the projects below. See
https://playwright.dev/docs/api/class-testoptions. */
  use: {
    /* Base URL to use in actions like `await page.goto('/')`. */
    baseURL: "http://localhost:5173/",

    /* Collect trace when retrying the failed test. See
https://playwright.dev/docs/trace-viewer */
    trace: "on-first-retry",
    launchOptions: {
      slowMo: 1000, // Slow down by ms
    },
  },

  /* Configure projects for major browsers */
  projects: [
    {
      name: "chromium",
      use: { ...devices["Desktop Chrome"] },
    },

    {
      name: "firefox",
      use: { ...devices["Desktop Firefox"] },
    },

    {
      name: "webkit",
      use: { ...devices["Desktop Safari"] },
    },
  ],
});
```

Questa configurazione punta alla cartella /tests a livello di root. Sarebbe buona prassi creare un altra cartella all'interno 'e2e' dove inserire tutti i

test.

5.4 COMANDI

Per eseguire tutti i test da CLI:

npx playwright test

Altrimenti utilizzare l'estensione di VSCode e runnare la cartella '/test' o la sottocartella '/tests/e2e'

5.5 ESEMPI PRATICI

Grazie all'estensione di VSCode è possibile generare i test in modo automatico. Per registrare un test, clicca sul pulsante "Record new" dalla barra laterale "Test". Verrà creato un test-1.spec.tsfile e si aprirà una finestra del browser. Nel browser, accedi all'URL che desideri testare e inizia a cliccare. Playwright registrerà le tue azioni e genererà il codice di test direttamente in VS Code. Puoi anche generare asserzioni scegliendo una delle icone nella barra degli strumenti e cliccando su un elemento della pagina su cui vuoi eseguire l'asserzione. È possibile generare le seguenti asserzioni:

'assert visibility'per affermare che un elemento è visibile
'assert text'per affermare che un elemento contiene testo specifico
'assert value'per affermare che un elemento ha un valore specifico
Una volta terminata la registrazione, clicca sul pulsante Annulla o chiudi la finestra del browser. Potrai quindi ispezionare il test-1.spec.tsfile e visualizzare il test generato.

Seleziona un localizzatore e copialo nel tuo file di test cliccando sul pulsante "Pick locator" dalla barra laterale di test. Quindi, nel browser, clicca sull'elemento desiderato e questo verrà visualizzato nella casella "Seleziona localizzatore" in VS Code. Premi "Invio" sulla tastiera per copiare il localizzatore negli appunti e incollarlo in qualsiasi punto del codice. Oppure premi "Esc" per annullare.

```js
import { test } from "@playwright/test";

// Suite di test per il componente Summary
test.describe("Summary component", () => {
  // prima di ogni test, la pagina va all'indirizzo indicato
  test.beforeEach(async ({ page }) => {
    await page.goto("http://localhost:5173/");
  });
  // dopo ogni test chiude la pagina
  test.afterEach(async ({ page }) => {
    await page.close();
  });
```

```
  test("search bar summary grid test", async ({ page }) => {
    await page.getByRole("searchbox", { name: "Cerca applicazione" }).click();
    await page
      .getByRole("searchbox", { name: "Cerca applicazione" })
      .fill("Demone Retail");
    await page
      .getByRole("gridcell", { name: "Demone Retail", exact: true })
      .click();
    await page
      .getByRole("gridcell", { name: "Demone Retail Features" })
      .click();
  });
});
```

5.6 AVANZATO
Playwright consente di testare anche le API REST senza librerie esterne, sfruttando APIRequestContext.

Struttura consigliata:
```
/tests
  ├── /pages        → Classe API utils (POM)
  │   └── APIutils.ts
  ├── /e2e           → Suite di test API
  │   └── APITestSuite.spec.ts
  └── /utils      → Utility comuni (es. mocks, helpers)
      └── apiMocks.ts
  └── /fixtures           → Fixture condivise
      └── api.fixture.ts
```

5.6.1 FIXTURE PERSONALIZZATE
Una fixture è una funzione speciale che prepara uno stato, risorsa o contesto da riutilizzare nei test. Aiuta a centralizzare il setup, evitare duplicazioni e gestire cleanup automatici.

Esempio base :
```api.fixtures.ts
import { test as base } from "@playwright/test";
import api from "../pages/ApiUtils";

type MyFixture = {
  API: api;
};

const fixtures = base.extend<MyFixture>({
  API: async ({ request }, use) => {
    const API = new api(request);
    await use(API);
  },
});
export { fixtures };
```

```

5.6.2 POM
È un pattern architetturale che modella ogni pagina dell'app come una classe contenente localizzatori e azioni. Separando struttura da logica di test, rende i test più leggibili, riutilizzabili e manutenibili.

Esempio :

```ApiUtils.ts
import { APIRequestContext } from "playwright-core";

export default class API {
  private request: APIRequestContext;

  constructor(request: APIRequestContext) {
    this.request = request;
  }

  // costruzione della richiesta ad EtichetteTraduzioni di devservices
  async callApplication(
    operation: string,
    token: string,
    values: Record<string, any> = {}
  ) {
    const endpoint =
      "https://devservices.siacloud.com/retail/EtichetteTraduzioni";
    const body = {
      output_format: "json",
      operation,
      token,
      values,
    };
    try {
      const response = await this.request.post(endpoint, {
        data: body,
      });

      if (!response.ok()) {
        console.error(`API call failed with status ${response.status()}`);
      }

      return response;
    } catch (error) {
      console.error(`Error during API call to ${endpoint}:`, error);
      throw error;
    }
  }
}
```
```

5.6.3 ESEMPIO TEST API

```apiRequest.spec.ts
import { expect } from "@playwright/test";
import { fixtures as test } from "../fixtures/api.fixture";

test.describe("Test API EtichetteTraduzioni", () => {
  const token = "9021s_TOKEN";
  test("getApplications restituisce un array di applicazioni", async ({
    API,
  }) => {
    const response = await API.callApplication(
      "retrieveApplications",
      token,
      []
    );

    expect(response.status()).toBe(200);
  });
});
```

6. BEST PRACTICE

- Scrivi test per il comportamento, non per l'implementazione. Testare l'interfaccia utente in base a ciò che l'utente vede o fa, non su come è implementata.
- Nomina i file in modo coerente: `*.test.ts(x)` o `*.spec.ts(x)`
- Evita test fragili: non usare `getByTestId` se puoi usare `getByRole` o `getByText`.
- Ogni test deve testare **una sola cosa**. Se ci sono troppi `expect()` in un test, è probabile che tu stia testando più comportamenti.
- Dai nomi descrittivi ai test
- Evita il mock eccessivo, soprattutto in test end-to-end o di integrazione.

7. ERRORI COMUNI

- Testare i dettagli interni dei componenti (es. classi CSS, DOM nidificato)
- Usare `await` senza `findBy` o `waitFor`
- Non usare `jest.mock()` quando test interagiscono con API reali
- Usare test troppo generici che non falliscono nemmeno se il componente non funziona più

8. TYPESCRIPT & TESTING

8.1 INTEGRAZIONE

L'uso di TypeScript nei test migliora la qualità del codice in modo significativo, perché:
- Riduce errori a compile-time, evitando test scritti in modo errato (es. mock con proprietà mancanti, props React mancanti o errate).

- Favorisce autocompletamento e refactoring sicuro, aumentando la produttività.
- Garantisce coerenza tipologica tra codice applicativo e test.
- Aiuta a documentare implicitamente le aspettative dei test grazie ai tipi.
- Migliora la manutenibilità e leggibilità del codice di test.

8.2 ESEMPI CON JEST

Test corretto di una funzione :

```sum.ts
export const sum = (a: number, b: number): number => a + b;
```

```sum.test.ts
import { sum } from './sum';

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Mock corretti vs errati.
dato il servizio :

```userService.ts
export const userService = {
  getUser: () => ({ id: 1, name: 'Alice' }),
};
```

Errore comune: mockare getUser restituendo un oggetto parziale.

```test.ts
jest.mock('./userService', () => ({
  userService: {
    getUser: jest.fn(() => ({ id: 1 })), // ✕ mancante 'name'
  },
}));
```

TypeScript segnalerà l’errore, impedendo di far partire il test con dati inconsistenti.
Corretto:

```test.ts
jest.mock('./userService', () => ({
  userService: {
    getUser: jest.fn(() => ({ id: 1, name: 'Mocked User' })),
  },
}));
```

8.3 ESEMPI CON REACT TESTING LIBRARY

Vantaggi:

- Il controllo dei tipi impedisce di dimenticare props obbligatorie quando si renderizza un componente.
- Aiuta a trovare errori nel codice e nei test prima dell'esecuzione.
- Migliora la scrittura di test simulando interazioni realistiche con componenti tipizzati.

Esempio corretto :
``` Button.tsx
import React from 'react';

type ButtonProps = { onClick: () => void };
export const Button: React.FC<ButtonProps> = ({ onClick }) => (
  <button onClick={onClick}>Click me</button>
);
```

```Button.test.tsx
import { render, screen, fireEvent } from '@testing-library/react';
import { Button } from './Button';

test('calls onClick when clicked', () => {
  const handleClick = jest.fn();
  render(<Button onClick={handleClick} />);

  fireEvent.click(screen.getByText('Click me'));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Esempio errore props mancanti:

```Button.test.tsx
import { render, screen, fireEvent } from '@testing-library/react';
import { Button } from './Button';

test('calls onClick when clicked', () => {
  const handleClick = jest.fn();
  render(<Button />);   // ❌ TypeScript segnala che 'onClick' è obbligatorio

  fireEvent.click(screen.getByText('Click me'));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

8.4 ESEMPI CON PLAYWRIGHT

Playwright supporta nativamente TypeScript, migliorando:

- la gestione delle promise con async/await
- la precisione nella selezione degli elementi UI
- la scrittura di test più chiari e manutenibili

Esempio corretto :

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

test('homepage has expected title', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page).toHaveTitle('Example Domain');
});
```

Errori comuni con Playwright e TypeScript
Mancanza di await su funzioni asincrone:
```
test('errore async', ({ page }) => {
  page.goto('https://example.com'); // ❌ manca await
  expect(page.title()).toBe('Example Domain'); // ❌ manca await
});
```

Click su elementi non presenti, che causano fallimenti:
```
await page.click('#non-existent-button'); // ❌ test fallirà se il bottone non c’è
```

Soluzione:
```
const button = page.locator('#non-existent-button');
if (await button.isVisible()) {
  await button.click();
}
```

8.5 ERRORI COMUNI

- Passare tipi errati o incompleti (es. mock senza tutte le proprietà)
- Usare componenti React senza tutte le props obbligatorie
- Dimenticare di usare await nelle chiamate asincrone
- Cercare elementi in test che potrebbero non esistere senza gestire il caso

8.6 CONCLUSIONE

Integrare TypeScript nei test non solo migliora la qualità del codice ma evita molti errori comuni durante lo sviluppo. Questo aumenta la fiducia nel codice testato e riduce i falsi negativi/falsi positivi, facilitando la manutenzione e scalabilità dei test nel tempo.

[autore] Nello Casolla