Testarea unei aplicatii mobile

Calculator App - React Native

Echipa A18

Sandu Eduard Alexandru

Alexe Vasile Paul

Nitoi Antonio

Componenta proiectului

1. Comparatia dintre 2 framework-uri de testare pentru aplicatii mobile in React Native: Jest & Mocha.

2. Utilizarea unui framework pentru a evidentia mai multe tipuri de teste (Jest).

Jest vs Mocha

- Am ales sa comparam framework-urile Jest si Mocha deoarece am observat ca acesta este un subiect dezbatut si fac parte din cele mai comune framework-uri de JavaScript.

- Jest este foarte potrivit "out-of-the-box" si intuitiv pentru teste simpliste insa nu atat de permisiv.

- Mocha este un cadru de testare mai flexibil și mai modular – permite testarea unor aspecte mai complexe.

Jest vs Mocha: Exemplu Unit Testing

Jest

```
// sum.js
function sum(a, b) {
  return a + b;
}
module.exports = sum;

// sum.test.js
const sum = require('./sum');

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

Mocha

```
// sum.js
function sum(a, b) {
  return a + b;
}
module.exports = sum;

// sum.test.js
const assert = require('assert');
const sum = require('./sum');

describe('Sum', () => {
  it('should return 3 when adding 1 and 2', () => {
    assert.strictEqual(sum(1, 2), 3);
  });
});
```

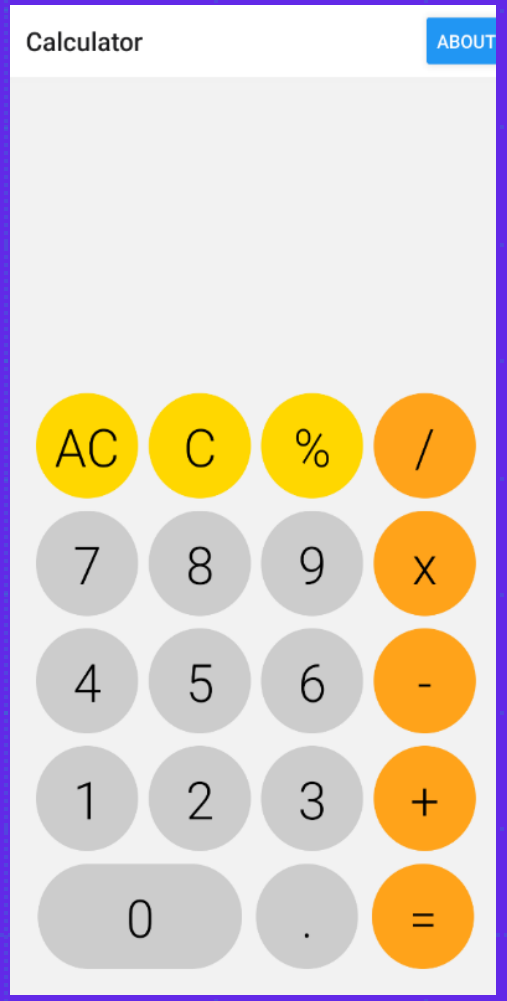
Avantajele Framework-urilor

- **Jest** este foarte usor de configurat si dispune de **snapshot testing, unit testing** si o **rulare rapida a testelor.**

- **Mocha** este ceva mai complex, dispune de foarte multe librarii externe (ex. Sinon) cu care putem face **mocking, stubbing** pe langa ceea ce poate face Jest.

Implementarea testelor in Jest

- Am folosit o aplicatie deja implementata (Un calculator pentru telefonul mobil) in React Native pentru a evidentia mai multe tipuri de teste de functionalitate prin **Unit tests, UI tests** si **Mutation tests.**



Unit tests in Jest

```
test("AC Button works correctly", async () => {
  const { getByTestId, getByText } = render(<Home />);
  const textInput = getByTestId("text-input");

  // update text input to have a number to check if it clears when pressing AC Button

  await fireEvent.changeText(textInput, "4873789598275");

  // get AC Button element and fire clicking event

  const button = getByText("AC");
  await fireEvent.press(button); // uncomment this line to see proper efect of test working

  // Now we have to check if the content of the text input is empty

  expect(textInput.props.value).toEqual('');

});
```

Am implementat aici un test pentru verificarea functionalitatii butonului de AC (Sterge tot). Am facut astfel de teste pentru toate obiectele din DOM-ul componentei. Aici populam input-ul cu un numar ca mai apoi sa mockuim o apasare de buton.

UI Snapshot tests in Jest

```
test("Text box component matches snapshot", () => {
  const tree = renderer.create(<TextBox> 1234 </TextBox>).toJSON();
  expect(tree).toMatchSnapshot();
});

test("Button component matches snapshot", () => {
  const tree = renderer.create(<Button />).toJSON();
  expect(tree).toMatchSnapshot();
});

test("Home screen matches snapshot", () => {
  const tree = renderer.create(<Home />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Folosind feature-ul out-of-the-box al Jest de snapshot testing, putem compara componentele cu o versiune "offline" a aplicatiei ca model de compatibilitate. Astfel, aceste teste ne asigura ca partea de UI a aplicatiei nu se schimba fara ca noi sa efectuam schimbari bine intentionate.

Mutation testing folosind Stryker & Jest

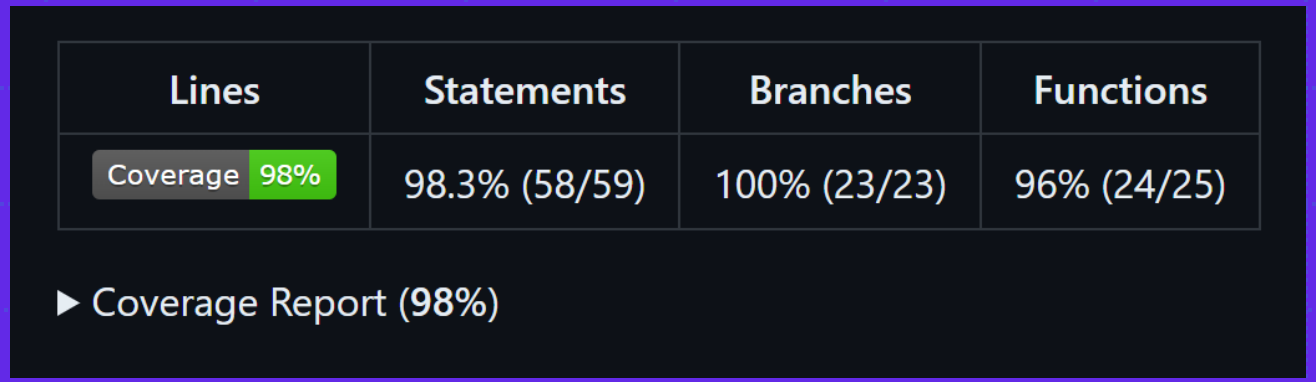
```
[Survived] ConditionalExpression
components/Home.js:34:41
-       if (last === "x" || last === "+" || last === "-" || last === "/") {
+       if (last === "x" || last === "+" || false || last === "/") {
Tests ran:
    Unit Testing Suite Div button works correctly
    Unit Testing Suite Multiply button works correctly
    Unit Testing Suite Minus button works correctly
  and 2 more tests!
```

File	% score	# killed	# timeout	# survived	# no cov	# errors
All files	84.09	126	22	24	4	0
About.js	83.33	10	0	1	1	0
AboutButton.js	0.00	0	0	0	3	0
Button.js	100.00	4	7	0	0	0
Home.js	83.69	104	14	23	0	0
TextBox.js	100.00	8	1	0	0	0

Am folosit unealta Stryker pentru a genera mutatii pentru componentele pe care dorim sa le testam in profunzime - aceste mutatii vor modifica la intamplare parti din componente iar mai apoi va rula testele scrie folosind Jest si va realiza un coverage report nou pentru fiecare componenta.

Github actions workflow pentru coverage report

- Jest are optiunea de a genera un raport de coverage in momentul rularii testelor pentru a se asigura ca toata aplicatia a fost testata, linie cu linie. Astfel, am scris un workflow folosind github actions care sa ruleze automat testele Jest la orice commit pe branch-ul **main.** Acest workflow va actualiza si un raport de coverage in readme-ul aplicatiei la commit.



Lines	Statements	Branches	Functions
Coverage 98%	98.3% (58/59)	100% (23/23)	96% (24/25)

▶ Coverage Report (**98%**)