UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAŞI
FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

GENERARE AUTOMATĂ DE UNIT TESTE PENTRU APLICAȚII REACT

propusă de

Oana Juravle

Sesiunea: *iulie, 2019*

Coordonator ştiinţific
Conf. Dr. Mădălina Răschip

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAŞI
FACULTATEA DE INFORMATICĂ

GENERARE AUTOMATĂ DE UNIT TESTE PENTRU APLICAȚII REACT

Oana Juravle

Sesiunea: *iulie, 2019*

Coordonator ştiinţific
Conf. Dr. Mădălina Răschip

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAŢIE privind originalitatea conţinutului lucrării de licenţă

Subsemntatul(a) ..…… domiciliul în …….. născut(ă) la data de ……..…., identificat prin CNP ……..……..., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de ……. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „Generare Automată de Unit Teste pentru Aplicații React”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iaşi, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent Oana Juravle

_____

(semnătura în original)

Introducere

În contextul dezvoltării web al zilelor noastre testarea joacă un rol foarte important în dezvoltarea aplicaţiilor. Însă este dificil pentru un programator să scrie întreaga funcţionalitate a unei aplicaţii fără a-i scăpa din vedere vreun detaliu şi pe termen lung scrierea de unit teste ajunge să consume mult din timpul şi resursele alocate unei aplicaţii. În timp, toate acestea fac din proiectarea de unit teste o sarcină plictisitoare şi de evitat printre programatori.

Proiectul prezentat în continuare aduce o optimizare a acestui proces de testare, prin generarea automată de unit teste necesare pentru o aplicaţie scrisă folosind ReactJS. Pe parcursul implementării am avut în vedere cele mai comune funcţionalităţi dintr-un proiect, pentru a avea rezultate satisfăcătoare indiferent de scopul proiectului în care este folosit acest program. În cadrul unei aplicaţii unit testele pot fi adăugate atât pe partea de server, pentru a testa *endpoint-urile* accesate de exemplu atunci când este încărcată o listă de date în pagină, cât şi pe partea de client pentru a testa interacţiunea dintre utilizator şi aplicaţie. Am ales ca acest proiect să fie orientat pe testarea care trebuie efectuată pentru a asigura buna funcţionare a aplicaţiei la interacţiunea cu utilizatorul, mai exact pe modul în care răspunde aplicaţia la acţiuni de tipul *onClick, onChange, onSubmit*. Schimbările vizuale sunt dificil de testat prin metoda abordată în generarea automată de teste prezentată în această lucrare deoarece programul nu oferă atât de mult control asupra conţinutului unei pagini însă este capabil să facă diferite presupuneri asupra acţiunilor care determină aceste schimbări. Ulterior, programatorul poate revizui rezultatul obţinut în urma generării şi să adauge afirmaţiile legate de schimbările vizuale care apar în conţinut. De exemplu, dacă avem un buton care deschide un modal, aplicaţia poate să identifice butonul şi să testeze ca funcţia corespunzătoare a fost într-adevăr apelată la *onClick* (deci la nivel de cod implementarea este corectă) însă poate fi util şi să verificăm dacă modalul este afişat în pagină. În acest caz, fişierul rezultat poate fi editat manual pentru a adăuga această verificare la finalul testului corespunzător.

În continuare, în primul capitol vor fi prezentate câteva informaţii generale referitoare la librăria ReactJS pentru a oferi contextul pentru cele ce urmează. De asemenea, voi vorbi despre unit teste, împreună cu avantajele şi limitările lor, dar şi despre paşii care trebuie urmaţi în crearea unit testelor pentru a avea o suită de teste care oferă garanţia că acoperă cele mai importante părţi ale aplicaţiei. Spre finalul capitolului sunt prezentate o serie de aplicaţii existente la momentul actual care încearcă să uşureze procesul de testare, urmate de un subcapitol care cuprinde contribuţiile proprii. Cel de-al doilea capitol al lucrării prezintă tehnologiile care au stat la baza implementării aplicaţiei. În capitolul trei este descrisă metoda abordată pentru generarea automată a unit testelor. Capitolul patru prezintă analiza rezultatelor aplicaţiei în practică.

1. Descrierea problemei

1.1. Ce este ReactJS?

ReactJS[1] este o librărie de Javascript utilizată pentru a crea interfeţe/aplicaţii web care nu necesită reîncărcarea paginii web în momentul în care datele se modifică (s*ingle-page apps*). Acest lucru face ca ReactJS să fie una dintre cele mai puternice librării la momentul actual, fiind în aceleaşi timp simplu de înţeles şi uşor de testat. A fost publicată în 2013 de echipa celor de la Facebook.

ReactJS are la bază componente[2]. O componentă este un modul care returnează un anumit rezultat. Conceptual, o componentă este o funcţie javascript: acceptă un set de date de intrare(în acest caz numite "props") şi returnează un element React. Aceasta poate conţine, la rândul său, una sau mai multe subcomponente. Componentele permit structurarea aplicaţiei în părţi independente şi reutilizabile, de exemplu: butoane, formulare, tabele, etc.

În ReactJS componentele pot fi definite în două moduri:

- Componente funcţionale (*Functional Components*)
  După cum sugerează şi numele, aceste componente sunt doar funcţii care acceptă un singur obiect drept argument - *props* şi returnează un element React. În cadrul componentelor funcţionale nu pot fi definite alte metode. De asemenea, nu există conceptul de stare a componentei(*state*).

- Componente clase (*Class Components*)
  Componentele definite în acest fel au câteva atribute adiţionale, printre care conceptul de stare şi cel de "ciclu de viaţă" al componentei(*component lifecycle*). Acesta este compus dintr-o serie de metode speciale care oferă control asupra componentei şi care se execută automat în diferite etape ale procesului - iniţializare, încărcare în DOM(*mounting*), actualizare, ştergere din DOM(*unmounting)*. Orice componentă trebuie să implementeze o metodă numita "render" care, pornind de la datele de intrare, returnează ceea ce urmează să fie încărcat pe ecranul utilizatorului.

Un alt element central în React este reprezentat de props - parametri folosiţi pentru a trimite date de la o componentă la alta, dar numai de la componenta părinte la componenta copil(reciproca nu este valabilă). Aceste valori pot fi de orice tip(de exemplu: string, boolean, object, array). Un alt lucru care merită menţionat despre *props* este că valorile acestora pot fi accesate doar pentru a fi citite, nu şi modificate.

De asemenea, există conceptul de state. *State-ul* este tot un obiect, similar celui de *props*. Diferenţa principală dintre cele două este dată de faptul ca *props* sunt transmise componentei, în timp ce *state-ul* este iniţializat şi modificat doar în interiorul componentei în

[1] https://reactjs.org/
[2] https://reactjs.org/docs/components-and-props.html

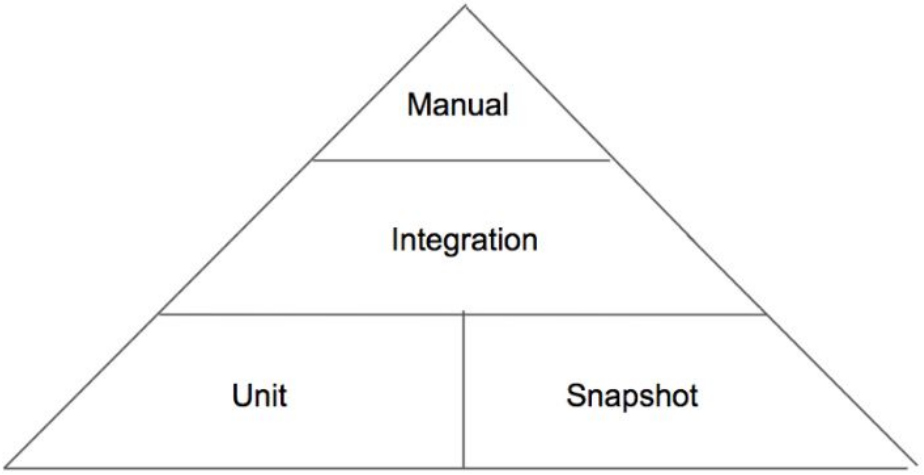
care este definit. În acest obiect sunt păstrate date care se pot schimba - de exemplu, datele dintr-un formular sau date filtrate în urma unei căutări/sortări.

1.2. Unit testing

1.2.1. Ce sunt unit testele?

Unit testele reprezintă primul nivel pentru testarea unei aplicaţii - în cadrul unit testelor sunt testate izolat mici funcţionalităţi /componente specifice ale aplicaţiei[3].

După cum se poate vedea în imaginea de mai jos, mai multe unit teste dar şi snapshot-uri pot fi folosite pentru validarea unui test de integrare şi mai multe teste de integrare sunt folosite pentru validarea unui test manual. Aşadar, unit testele joacă un rol foarte important în asigurarea corectitudinii generale a aplicaţiei şi chiar unii programatori - autori ai unor articole de inspiraţie pentru această lucrare - sunt de părere că este obligatoriu să începi cu testarea bazată pe unit teste/snapshot-uri şi să foloseşti testele de integrare doar dacă este necesar. Unul din motivele care susţin această afirmaţie este faptul că pentru a începe să adaugi teste de integrare trebuie să aştepţi până când întreaga funcţionalitate este completă, pe când unit testele sunt implementate progresiv, odată cu fiecare parte de logică adăugată.



Piramida de testare[4]

Unit testele aparţin abordării de testare numite “White Boxing” deoarece testele sunt adăugate după ce funcţionalitatea a fost deja implementată. Din aceste cauze, unit testele sunt în cele mai multe cazuri scrise de către programatorii care au implementat respectiva parte de funcţionalitate, folosind date de testare separate de cele folosite de echipa de testare.

Pentru a avea garanţia ca unit testele acoperă într-adevăr părţile importante din componenta testată, în general se foloseşte o altă tehnică de White Boxing, şi anume “Code Coverage”-ul. Analiza *coverage* identifică ce funcţionalităţi nu au fost incluse în suita de teste

[3] https://www.guru99.com/unit-testing-guide.html#1
[4] https://hackernoon.com/using-jest-and-enzyme-for-testing-react-apps-4d3e1543bf0d

corespunzătoare. Aceasta oferă informaţii referitoare la *Statement Coverage* - verifică dacă toate liniile de cod sunt parcurse de suita de teste cel puţin o dată, informaţii referitoare la *Branch Coverage* - verifică dacă sunt acoperite toate ramurile *if/else* precum şi celelalte expresii conditionale, etc.

1.2.2. Care sunt avantajele unit testelor?

Unit testele ajută la detectarea schimbărilor apărute în proiect (de exemplu, defecte de implementare) şi la fixarea lor imediată, cu un cost minim, fără a impacta alte părţi ale proiectului. Conform studiului "TDD--The Art of Fearless Programming"[5] publicat în IEEE Software, rata de depistare a defectelor în producţie poate fi scăzută cu 40%-80% pentru proiecte care au la bază o structură bine definită de unit teste.

De asemenea, acestea asigură faptul ca funcţionalitatea curentă se păstrează în cazul în care codul iniţial suferă refactorizări. Cât timp testele din suita corespunzătoare logicii care este în proces de schimbare trec, programatorul poate avea siguranţa că aplicaţia îşi păstrează funcţionalitatea. În acelaşi timp, prin unit teste se poate descoperi că o parte din logică a fost omisă. Descoperită la acest pas, funcţionalitatea lipsă poate fi adăugată înainte ca respectivul task să ajungă în etapa de testare.

Unit testele pot servi drept documentaţie pentru aplicaţia în dezvoltare. Dat fiind că o suită completă de teste trebuie să includă toate cerinţele pentru o anumită parte de aplicaţie, oferă o mai bună înţelegere a codului. Descrierea detaliată a testelor dintr-o suită este foarte utilă în acest caz.

1.2.3. Care sunt limitările unit testelor?

Unit testele necesare componentelor funcţionale sunt uşor de scris şi asigură acoperire totală deoarece o componentă funcţională returnează, pentru acelaşi set de date de intrare, acelaşi rezultat şi nu există efecte secundare. Aşadar un singur test care verifică corectitudinea rezultatului pe baza obiectului *props* primit este suficient. Însă de multe ori componentele, oricât de bine modularizate ar fi, ajung să aibă o structură mai complexă de atât (*class components*).

Pe măsură ce proiectul se dezvoltă, creşte şi numărul de unit teste necesare, acestea devenind tot mai complexe şi costisitoare atât ca şi timp cât şi ca şi cerinţe financiare. Cerinţele proiectului pot include menţinerea nivelului de *code coverage* destul de mare(80-90%), uneori ajungând şi la 100%. Acest lucru înseamnă o scădere a ratei de implementare pentru funcţionalităţi noi, dat fiind că programatorii trebuie să acorde timp suplimentar pentru scrierea unit testelor.

Un alt dezavantaj este faptul că pentru funcţionalităţi comune însă folosite în componente diferite aceeaşi logică de testare este folosită în locuri multiple. Putem lua ca exemplu testarea unui formular - în general, paşii pentru validarea unui formular presupun

[5] https://ieeexplore.ieee.org/document/4163024

populareа câmpurilor din formular şi verificarea că aceste valori sunt salvate la trimiterea formularului.

1.2.4. Ce trebuie testat?[6]

În React regula de bază atunci când sunt adăugate unit testele pentru o componentă este ca testarea să se orienteze pe tot ce nu este static. Cu alte cuvinte:

- Dat fiind un set de date de intrare(reprezentat în acest caz de *state/props*), să se testeze ce anume este returnat de componentă;
- Dată fiind acţiunea unui utilizator, să se testeze răspunsul componentei, de exemplu: este modificat *state-ul* sau este apelată o metodă a componentei respective ori trimisă de componenta părinte prin intermediul *prop-urilor*.

Orice altceva poate fi omis, având un motiv întemeiat. De exemplu:

- Librăriile puse la dispoziţie de diferite pachete: în general, librăriile au deja propriile teste implementate pentru funcţionalitatea pusă la dispoziţie;
- Constantele: aşa cum a fost menţionat mai sus, unit testele ar trebui să verifice doar partea de logică a componentei care poate suferi modificări;
- Alte componente care sunt folosite în cadrul celei testate: fiecare componentă trebuie să aibă propriul fişier de test, astfel încât o componentă părinte nu este responsabilă pentru testarea corectitudinii a unei componente copil.

1.3. Implementări existente

La momentul curent nu există prea multe librării care să genereze unit teste pentru o anumită funcţionalitate implementată. Pentru tehnologii precum Java şi Python există câteva unelte care oferă garanţia faptului că testele obţinute dau rezultate bune şi sunt folosite chiar în cadrul diverselor companii. Acest lucru este posibil deoarece Java şi Python sunt limbaje folosite în general pe backend, iar rezultatele funcţiilor implementate sunt destul de predictibile - primesc un set de date de intrare şi returnează un set de date de ieşire care eventual este folosit în cadrul altor funcţii. Spre deosebire de Java şi Python, metodele definite în Javascript şi în mod special React se reflectă în interfaţa utilizatorului, ceea ce duce la o multitudine de rezultate posibile. Din această cauză este dificil să se facă presupuneri legate de conţinutul unei pagini web prin intermediul unor teste generate automat.

În continuare sunt prezentate câte o librărie de generare automată de unit teste pentru Java şi Python, una pentru Javascript şi una pentru React.

---

6 https://www.freecodecamp.org/news/components-testing-in-react-what-and-how-to-test-with-jest-and-enzyme-7c1cace99de5/

Randoop[7]

Randoop este un generator de teste pentru Java. Este folosit în diferite pachete, ca de exemplu java.util, în diverse proiecte *open-source* dar şi în companii ca ABB şi Microsoft, pentru a descoperi greşeli de implementare sau logică existente în codul curent dar şi defecte care pot să apară în implementările viitoare. Testele sunt generate aleatoriu pe o perioadă de timp setată de tester şi returnate într-un format JUnit. Generarea de teste se face prin verificarea structurii clasei testate, mai exact a constructorilor şi a metodelor publicate în interfaţa acesteia. Se bazează pe faptul că, în timpul setat de tester, prin parsarea aleatorie a codului, vor fi acoperite toate secvenţele de cod esenţiale. În urma acestei operaţii sunt generate două fişiere - unul conţinând testele care au picat în timpul execuţiei, şi deci pot indica potenţiale erori în clasele testate, şi un alt fişier în care sunt trecute testele care au trecut în urma execuţiei de mai sus şi pot fi folosite pentru validări ulterioare.

Auger[8]

Auger este un generator de teste pentru Python. Cu ajutorul acestei librării, testele sunt generate pe măsură ce se adaugă codul, împreună cu datele de test necesare, astfel[9]: pentru fiecare funcţie definită în cadrul modulului, Auger extrage valorile argumentelor şi a rezultatului returnat, după care este simulat un apel al acelei funcţii cu argumentele şi valoarea returnată înlocuite cu valorile de test corespunzătoare; pentru funcţiile apelate din diverse librării este extrasă valoarea returnată în urma execuţiei astfel încât în test valoarea returnată de acelaşi apel al funcţiei să poată fi înlocuită corespunzător.

JS Test Gen[10]

JS Test Gen este un pachet care generează template-uri de unit teste pentru funcţii Javascript. Se foloseşte de Babel[11] pentru a identifica modulele care sunt exportate din fişierul testat. Odată identificate, generează template-urile de test formatate cu ajutorul unui pachet numit prettier[12]. Din păcate, această librărie generează doar template-uri generale pentru funcţiile exportate dintr-un fişier, nu face niciun fel de presupunere asupra rezultatului actual obţinut în urma execuţiei acestora.

[7] https://randoop.github.io/randoop/
[8] https://github.com/laffra/auger
[9] http://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html
[10] https://github.com/js-test-gen
[11] https://babeljs.io/
[12] https://prettier.io/

```
export const sum = (x, y) => x + y;
```

```
describe("sum", () => {
  it("should fail auto generated test", () => {
    expect(sum().toBe(false));
  });
});
```

Rezultat obţinut folosind JS Test Gen

Snappify[13]

Snappify este o librărie creată pentru testarea componentelor scrise cu React şi TypeScript. Snappify se foloseşte de faptul că, folosind TypeScript, există o interfaţă care descrie tipul *prop-urilor* folosite în componenta respectivă. Pe baza acestor informaţii sunt generate valori aleatoare corespunzătoare *prop-urilor* şi apoi se creează testele folosind Snapshots. O idee similară este abordată şi în librăria *generator-react-jest-tests*[14]. Deşi soluţia propusă în această librărie are potenţial, în proiectele reale este nevoie de teste care să verifice mai riguros funcţionalitatea dată, iar testarea bazată doar pe snapshot-uri construite din combinaţii ale *prop-urilor* primite de un element nu asigură neapărat funcţionarea corectă a aplicaţiei, ci doar că între rulări succesive ale testelor nu au apărut modificări în structura elementului respectiv.

```
test("Button case #1", () => {
  const tree = renderer
    .create(
      <Button
        // some randomly generated values for props
        className="value"
        idDisabled={true}
        onClick={() => {}}
        children={<div />}
      />
    )
    .toJSON();

  expect(tree).toMatchSnapshot();
});
// Other test cases that contain randomly generated values for props go here...
```

Rezultat obţinut folosind Snappify

[13] https://github.com/denisraslov/snappify
[14] https://www.npmjs.com/package/generator-react-jest-tests

1.4. Contribuţii proprii

După cum se poate observa mai sus, la momentul actual nu există o librărie care să vină cu o serie de verificări care să asigure ca funcţionalitatea dată este acoperită corect sau complet prin unit teste. Programul prezentat în această lucrare face un pas înainte oferind o suită de teste care acoperă cele mai comune cazuri de testare pentru o componentă React dată, indiferent de librăriile folosite pentru scrierea ei. Mai exact, oferă suport pentru:

- verificarea comportamentului componentei atunci când sunt folosite doar *prop-urile default* dar şi pentru încărcarea componentei cu o listă de *prop-uri* setate de cel care a implementat componenta respectivă. În acest fel şi programatorul are control asupra datelor de test care vor fi folosite, ceea ce duce la un nivel mai ridicat de încredere în testele generate;
- testează acţiunea evenimentelor de *onClick și redirect* pentru orice element. Prin testarea acestor evenimente, se garantează faptul că la nivel de cod totul funcţionează conform implementării și că interfața aplicației răspunde corespunzător în interacţiunea cu utilizatorul;
- validarea formularelor prin verificări făcute atât pentru câmpuri valide cât și pentru cazul în care formularul ar putea avea erori;
- returnarea unui raport al încercărilor de creare a unor teste care, din diferite motive, nu au putut fi adăugate cu succes, împreună cu motivul pentru care încercările au eşuat. În acest fel programatorul este notificat de faptul că este posibil ca aplicaţia să returneze o suită de teste care acoperă o parte mai mare din implementare doar că, de exemplu, lipsește un identificator pentru elementul testat.

2. Tehnologii folosite

Jest

Jest[15] este una dintre cele mai folosite librării de testare la momentul actual şi este dezvoltată de Facebook. Am ales această librărie pentru proiectul meu din motive de performanţă dar şi pentru că oferă o sintaxă foarte intuitivă şi vine cu toate pachetele necesare astfel încât nu e nevoie să se instaleze librării adiţionale pentru a avea o suită de teste eficiente. De asemenea, are nevoie de foarte puţine configurări pentru a fi integrată în cadrul proiectului şi oferă posibilitatea testării folosind snapshot-uri.

După cum spuneam, sintaxa folosită este foarte intuitivă iar acesta este un lucru bun în general în unit teste, şi în special în această aplicaţie - dat fiind că testele sunt generate automat, au nevoie de o descriere cât mai clară, care să ajute programatorul să vadă care sunt părţile de logică acoperite. Acest lucru se face cu ajutorul a trei metode[16] oferite de librăria Jest:

- *describe()*
  describe() este o metodă opţională folosită pentru a grupa un set de teste şi conţine un text ce reprezintă o descriere a ceea ce se testează în suită;
- *test() / it()*
  test() este similar cu *describe()*. Conţine un text reprezentând descrierea testului curent. *it()* este un alias pentru *test()*;
- *expect(); toEqual()*
  expect() cuprinde presupunerile făcute în cadrul testului. *expect()* este imediat urmat de *toEqual()* - o funcţie a cărei evaluare trebuie să fie mereu adevărată, în caz contrar întreaga execuţie a suitei se încheie şi este returnată eroarea care a cauzat oprirea rulării testelor. *toEqual()* face parte dintr-o listă mult mai mare de comparatori, printre care şi *toContain(), toBeCalled(), toHaveBeenCalledWith()*, etc.

De asemenea, Jest oferă posibilitatea de a executa părţi de logică înainte/după fiecare test, respectiv înainte/după întreaga suită de teste, printr-o sintaxă de felul:

[15] https://jestjs.io/
[16] https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8

```
describe("Component Unit Tests", () => {
  beforeEach(() => {
    initializeUsersList();
  });

  it("users list has 4 results", () => {
    expect(getUsers().length).toEqual(4);
  });
});
```

Exemplu de test scris cu ajutorul librăriei Jest

Snapshots

Snapshot-ul[17] reprezintă un tip de testare ce aparţine librăriei Jest şi care verifică, la fiecare execuţie a testelor, că nu apar schimbări neaşteptate în interfaţa utilizatorului.

Atunci când testele rulează pentru prima dată, dacă funcţia *toMatchSnapshots()* este întâlnită într-unul din teste - de cele mai multe ori după o metodă *expect()* - Jest va genera automat un fişier de shapshot într-un director __snapshots__ din cadrul celui de __tests__.

```
describe("Component Snapshot", () => {
  let tree;
  it("renders correctly", () => {
    tree = renderer.create(<Component />).toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Exemplu de generare a Snapshot-urilor

În fişier va fi salvată o reprezentare a componentei încărcate. Acesta poate fi validat sau editat pentru a asigura corectitudinea datelor iniţial salvate.

La următoarele rulări ale testelor, Jest va încărca snapshot-urile deja existente şi le va compara cu cele returnate de testul curent. Eventualele diferenţe vor fi afişate în terminal. De exemplu, dacă în implementarea curentă a fost adăugat un buton nou, atunci când se execută testele, se va returna în consolă o eroare cu privire la faptul că cele două versiuni de snapshot nu corespund, împreună cu diferenţele găsite. Acestea sunt mai apoi evaluate şi dacă într-adevăr schimbările au fost intenţionate se actualizează fişierul de snapshot.

Acesta este doar unul din cazurile în care snapshot-urile pot fi folosite. Un alt caz în care snapshot-urile se dovedesc a fi utile este pentru compararea obiectelor.

[17] https://jestjs.io/docs/en/snapshot-testing

Enzyme

Enzyme[18] este o librărie dezvoltată de Airbnb specific creată pentru testarea componentelor React. Principalul scop al librăriei este de a construi o interfață mai simplă pentru scrierea unit testelor. În acest sens, Enzyme oferă trei funcţii prin care o componentă poate fi inclusă în teste:

- *mount()*
  această metodă este folosită atunci când este util să fie încărcată întreaga componentă împreună cu componentele copil corespunzătoare sau este nevoie să fie accesate ori modificate valorile din props(inclusiv *defaultProps*) - atât pentru componenta dată cât şi pentru componentele copil;
- *shallow()*
  această metodă încarcă doar componenta, ignorând toate componentele copil. Nu poate accesa valorile din props(inclusiv *defaultProps*) însă poate testa efectul pe care îl au aceste valori în componenta dată;
- *render()*
  această metodă generează codul HTML corespunzător componentei React, iar analiza testelor se face pe baza codului HTML.

```
describe("Component Unit Tests", () => {
  let component;
  beforeEach(() => {
    component = mount(<Component />);
  });

  afterEach(() => {
    component.unmount();
  });

  it("renders correctly", () => {
    expect(component.length).toBe(1);
  });
});
```

Exemplu de folosire a metodei mount

În afara acestor metode, Enzyme pune la dispozitie şi alte metode, de exemplu:

- pentru a simula evenimente - *onClick*, *onChange*, *onKeyDown*, etc: *component.find(selector).simulate(event),* unde selectorul poate fi de exemplu un selector css;
- pentru a modifica valorile din *props*: *component.setProps(newProps)*.

[18] https://airbnb.io/enzyme/

3. react-unit-test-generator

3.1. Ideea

Dată fiind o componentă, programul o va analiza şi va extrage din ea atributele necesare elementelor care urmează a fi testate.

Spre exemplu, dacă componenta conţine un tabel sortabil de utilizatori şi un buton pentru adăugarea unui utilizator nou, putem extrage id-urile tabelului/butonului pentru a ne folosi de aceste valori în teste la identificarea elementelor în cadrul componentei. Dacă butonul redirecţionează spre altă pagină a aplicaţiei putem lua şi valoarea noului URL pentru a testa şi această funcţionalitate. Aşadar, o suită completă de teste pentru acest exemplu ar putea conţine:

- un test pentru a verifica dacă componenta a fost încărcată în pagină;
- un test pentru a verifica dacă se face corect sortarea elementelor din tabel;
- un test pentru a verifica funcţionalitatea butonului de adăugare a unui utilizator nou.

3.2. Structura proiectului

Proiectul este structurat în două directoare principale: *src şi dist.* La acestea se adaugă directorul *node_modules*. Pe lângă acestea, în rădăcina proiectului se găsesc şi diferite fişiere de configurare despre care voi vorbi mai jos.

- Directorul src

  Aici se află toată logica pentru generarea template-urilor, printre care: fişierul de intrare(index.js), fişierul în care se află funcţia responsabilă de generarea arborelui de identificatori(createIdentifiersMap.js), precum şi alte două subdirectoare: templates şi helpers. După cum ii spune şi numele, primul subdirector cuprinde toate template-urile din care este alcătuit fişierul de teste returnat. Aceste template-uri sunt organizate, la rândul lor, în alte subdirectoare, pe baza funcţionalităţilor pentru care sunt folosite: logica pentru testare butoanelor se află în directorul “buttons”, cea pentru testarea formularelor în directorul “forms”, etc. Cel de-al doilea director - helpers - conţine funcţii ajutătoare care sunt folosite în mai multe locuri. În acest fel se respectă principul “Don’t repeat yourself” (DRY) care spune ca logica ce se repetă într-un proiect trebuie abstractizată.
- Directorul dist

  Acest director cuprinde codul compilat dintr-un director dat(în cazul nostru src) şi este adăugat automat prin rularea unei comenzi care trebuie inclusă în *package.json* sub opţiunea “scripts” sau executată direct din terminal. Compilarea codului reprezintă un pas esenţial pentru a ne asigura ca aplicaţia noastră va putea fi integrată într-un proiect.
- Directorul node_modules

  Şi acest director este creat şi actualizat automat de fiecare dată când se adaugă un pachet nou în aplicaţia dată, prin executarea unei comenzi “*npm install*

<NumePachet>". Această comandă descarcă pachetul cerut în directorul node_modules. În acest fel, atunci când este folosită, de exemplu, o funcţie dintr-un pachet instalat, definiţia ei va exista în directorul node_modules şi va fi executată corect. Acest director nu este adăugat în repository-ul publicat pe GitHub deoarece lista de pachete de care are nevoie proiectul este cuprinsă în package.json şi prin executarea unei comenzi "*npm install*" fiecare pachet este descărcat şi adăugat în acest director.

- package.json

  Acest fişier este creat încă de la început, atunci când se iniţializează aplicaţia printr-o comanda "*npm init*". În urma executării acestei comenzi sunt adăugate o serie de câmpuri în fişierul package.json, dintre care:
  - numele aplicaţiei: react-unit-test-generator
  - versiunea curentă a aplicaţiei
  - un câmp "main" corespunzător fişierului de intrare în programul nostru (în acest caz este index.js din directorul dist)
  - un câmp "scripts" unde am definit comanda pentru crearea directorului dist
  - un câmp "dependencies"
  - un câmp "devDependencies"

  "dependencies" şi "devDependencies" sunt obiecte care fac referire la pachetele instalate în proiect, sub forma "*<NumePachet>: <VersiuneInstalată>*". Diferenţa dintre cele doua obiecte este dată de scopul pentru care au fost instalate pachetele: dacă un pachet este folosit doar pe parcursul dezvoltării aplicaţiei sau pentru testarea acesteia, va fi adăugat în obiectul "devDependencies"(de exemplu: jest, enzyme); dacă programul este construit pe baza unui pachet (de exemplu: react, react-test-renderer) atunci acesta trebuie adăugat în obiectul "dependencies".
- babel.config.js
- jest.config.js

3.3. Publicarea aplicaţiei ca pachet pe npm

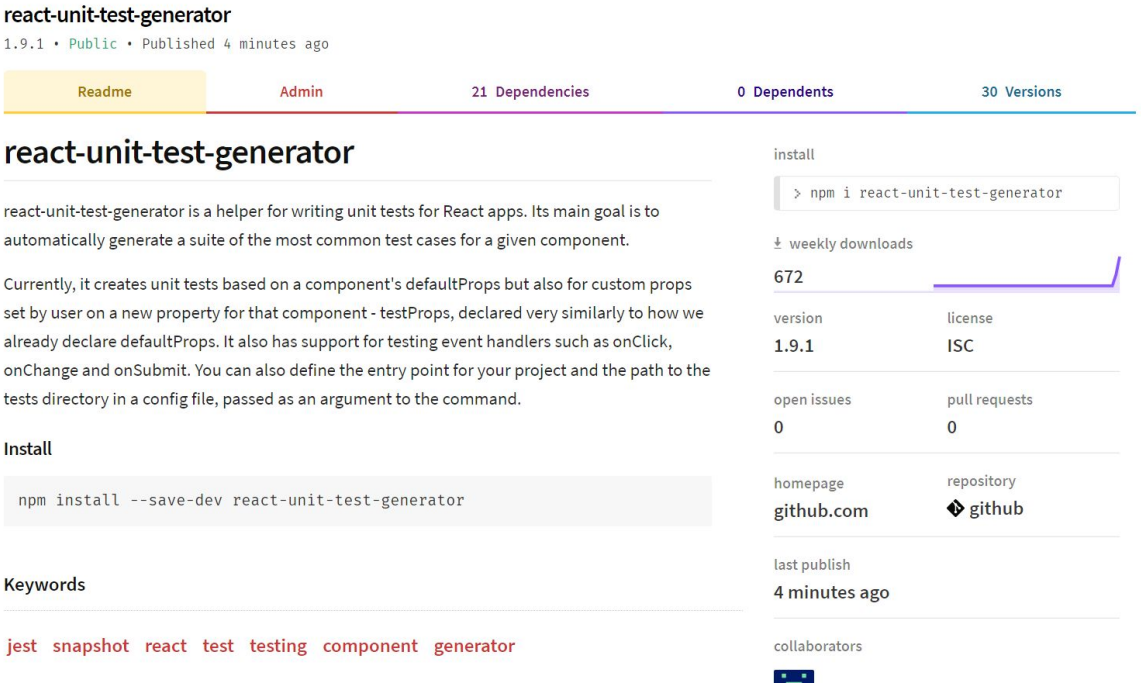
Aplicaţia prezentată este creată cu scopul de a oferi suport în scrierea unit testelor. Aşadar, nu este implementată pentru a fi folosită ca o aplicaţie de sine stătătoare ci în cadrul unui proiect care foloseşte React. Pentru a atinge acest scop, am încercat să abstractizez cât mai mult logica din aplicaţie astfel încât aceasta să fie cât mai independentă de structura şi tehnologiile utilizate în cadrul unui proiect.

Modul cel mai uşor prin care o astfel de aplicaţie ajunge să fie folosită într-un proiect real este prin publicarea ei ca şi pachet pe *npm*[19] (Node Package Manager), în acest fel devenind accesibilă pentru oricine doreşte să o utilizeze. npm este cel mai mare registru software, folosit pentru a publica un pachet nou sau pentru a descărca pachetele folosite într-un anumit proiect. Aşadar, am decis să public aplicaţia prezentată în primul rând pentru

[19] https://www.npmjs.com/

a-i demonstra utilitatea într-un mediu practic şi în al doilea rând pentru a-i creşte gradul de accesibilitate aşa încât să poată fi folosită de oricine doreşte să o includă în proiectele proprii.

La momentul scrierii acestei lucrări, pachetul se află la versiunea 1.9.1 şi are un număr de 672 descărcări pe săptămână.



react-unit-test-generator ca pachet pe npm

Acest număr nu reprezintă numărul de descărcări unice al pachetului. Numărul de descărcări este incrementat de fiecare dată când este publicată o nouă versiune a pachetului. De asemenea, tot la publicarea unei noi versiuni, pachetul este descărcat de roboţi care îl analizează şi se asigură că este sigur, dar şi de portaluri care descarcă pachetul pentru a-l expune pe propriul lor domeniu. Cu toate acestea, putem aproxima că, deşi pachetul nu era la o versiune finală, pe parcursul implementării a atras atenţia unor dezvoltatori care au considerat ca le-ar fi util un astfel de pachet în proiectele lor.

3.4. Descrierea soluţiei

3.4.1. Setup

Dat fiind că aplicaţia se află pe registrul npm este necesar mai întâi ca aceasta să fie descărcată printr-o comandă

“npm install --save-dev react-unit-test-generator”

Deoarece pachetul este folosit pentru generarea unit testelor şi nu influenţează comportamentul aplicaţiei expuse utilizatorului, este recomandat ca acesta să fie instalat în

obiectul *devDependencies* din package.json. Pentru aceasta am adăugat opţiunea --*save-dev* pentru descărcarea pachetului.

Următorul pas pentru a putea folosi pachetul constă în adăugarea unei comenzi sub câmpul "scripts" din fişierul package.json corespunzător aplicaţiei, care va rula fişierul dat ca valoare a câmpului "main" din fişierul package.json al pachetului. Această comandă trebuie să fie de forma:

"*<NumeComanda>: react-unit-test-generator"*

Prin execuţia pachetului ca şi comandă în acest fel se creează două variabile în obiectul global *process.env* de care ne vom folosi pentru a obţine informaţii despre aplicaţia care foloseşte acest pachet - npm_package_name şi despre fişierul pentru care se vor genera unit testele - npm_config_fileName.

Această comandă poate primi un argument opţional, adăugat în scriptul din fişierul package.json: --*config*=<*NumeFişierDeConfigurare*>. Fişierul de configurare trebuie să fie adăugat în rădăcina aplicaţiei şi să conţină o valoare *json.* În această valoare vor fi specificate:

- calea către directorul din care programul poate să acceseze componentele pentru care va genera teste: { *"entry": "./root/src" }*
- calea către directorul în care va crea fişierul cu testele returnate: { "*destination": "./root/tests"}*

Dacă acest argument nu este adăugat, se consideră ca aplicaţia are structura cea mai comună, în care componentele se găsesc într-un director *src* şi testele trebuie adăugate într-un director *tests*, ambele în rădăcina aplicaţiei. Dacă este adăugat, se verifică dacă acest fişier există şi dacă conţinutul său poate fi accesat. În caz de eroare, aceasta este afişată în terminal şi programul îşi opreşte execuţia.

3.4.2. Identificarea fişierului pentru care se vor genera testele

După cum am menţionat mai sus, programul îşi va începe execuţia printr-o comandă adăugată în fişierul package.json şi executată de la linia de comandă sub forma:

"*npm run <NumeComandă> --fileName=<NumeFişier>*"

Parametrul pe care îl primeşte acest script - *NumeFişier* - este obligatoriu şi poate fi numele fişierului care conţine definiţia componentei sau calea(completă ori parţială) către fişierul respectiv.

La acest pas este validat fişierul. Aici, dar şi în cazurile prezentate mai jos, am ales să înlocuiesc mesajele de eroare legate de liniile din cod care au returnat eroarea respectivă cu mesaje care să ofere informaţii utile în corectarea ei. Aşadar, se verifică dacă comanda a fost executată cu argumentul corespunzător numelui fişierului; în caz contrar, un mesaj specific va fi afişat în terminal.

După ce se validează numele fişierului de configurare (dacă este cazul) şi calea către fişierul din datele de intrare, programul parcurge recursiv toate fişierele din directorul dat ca şi punct de intrare în aplicaţie şi returnează calea către acele fişiere a căror cale se termină cu argumentul citit din terminal. Spre exemplu, pentru rularea unei comenzi "npm run generate-tests --fileName=Users" o posibilă listă de rezultate ar putea conţine '/components/Users.js,', 'src/components/AllUsers.js' etc, dar nu va include un rezultat de felul "src/components/UsersIndex.js".

Dacă algoritmul recursiv de identificare a fişierului respectiv nu a returnat cel puţin un element, utilizatorul va fi informat despre acest lucru.

Pentru lista de fişiere returnată de acest algoritm, înainte de începerea execuţiei programului pentru generarea fişierului de unit teste corespunzător, se face încă o verificare legată de permisiunile acestuia; dat fiind ca programul trebuie să citească din fişierul respectiv definiţia componentei, este important să avem drepturile necesare.

3.4.3. Crearea fişierului destinaţie

Următorul pas este cel de creare a fişierului destinaţie, care se va adăuga în directorul de teste şi va avea acelaşi nume ca şi componenta testată, urmat de extensia *.test.js* (dacă fişierul de test cu acest nume există deja atunci conţinutul său va fi suprascris).

3.4.4. Crearea arborelui de identificatori

Pentru a demonstra faptul ca programul nu este dependent de o anumită structură sau sintaxă, în componentele prezentate elementele sunt definite atât direct, ca şi tag-uri de HTML, cât şi prin intermediul unei librării. Pentru exemplificare, am folosit librăriile *material-ui*[20] şi *semantic-ui-react*. Am ales aceste librării deoarece ambele aduc o sintaxă diferită, stiluri şi nivele de elemente suplimentare. Spre exemplu, componenta "Message" din *semantic-ui-react* transformată în HTML va arăta ca în exemplul de mai jos:

```
<Message
  success
  icon="thumbs up"
  header="Nice job!"
  content="Your profile is complete"
/>
```

```
<div class="ui success message">
  <i class="thumbs up icon" />
  <div class="content">
    <div class="header">Nice job!</div>
    Your profile is complete
  </div>
</div>
```

Diferenţe de sintaxă între semantic-ui-react şi HTML[21]

[20] https://material-ui.com/
[21] https://react.semantic-ui.com/

În continuare, rezultatele prezentate sunt obţinute utilizând librăria *semantic-ui-react* deoarece librăria *material-ui* are un comportament similar. Diferenţa principală dintre cele două librării se referă la structura (ierarhia) folosită pentru încapsularea diferitelor elemente.

Pentru algoritmul responsabil de extragerea elementelor relevante în cadrul testelor am încercat diferite abordări.

Prima abordare presupunea citirea fişierului în care este definită componenta şi parcurgerea lui astfel: ştiind ca fiecare componentă declarată ca şi clasă trebuie să conţină o metodă numita *render* - responsabilă cu returnarea conţinutului care va fi afişat în pagină - se poate ignora restul codului iar toate verificările care urmează să fie făcute doar pe cuprinsul acestei metode. Pentru acest lucru însă trebuie făcută riguros delimitarea funcţiei; chiar dacă în general metoda *render* începe şi se termină cu acolade, în interiorul ei există şi alte seturi de acolade de care trebuie ţinut cont. Acest lucru este valabil mai apoi pentru fiecare metodă adăugată la evenimentele de tip *onClick* sau *onChange* - cu atât mai mult cu cât definiţia acestora poate fi făcuta şi *inline*. Pentru a stabili unde începe şi se termină fiecare element este nevoie de o verificare similară, şi apoi apare şi problema elementelor definite ca şi tag-uri de HTML şi componente luate din librăria menţionată mai sus. Luând toate aceste lucruri în calcul plus problema parcurgerii fişierului în primul rând, am decis să ma îndrept spre altă soluţie.

O altă metodă - cu care am hotărât să merg mai departe în implementarea algoritmului - presupune folosirea unei pachet oferit de React, *react-test-renderer*[22]. React-test-renderer este un pachet destul de popular la momentul curent (> 2 milioane descărcări săptămânal) ce oferă o modalitate de a transforma o componentă React într-un obiect Javascript care nu depinde de DOM. Singurul dezavantaj al acestei abordări este faptul ca pentru acelaşi element din pagină, în funcţie de cum a fost încărcat - doar HTML sau printr-o componentă din semantic-ui-react - obiectul rezultat poate arăta diferit:

```
<button
  data-testid="test-button"
  onClick={this.handleClick}>
  Example Button
</button>
```

```
{ type: 'button',
  props:
   { 'data-testid': 'test-button',
     onClick: [Function: bound handleClick] },
  children: [ 'Example Button' ] }
```

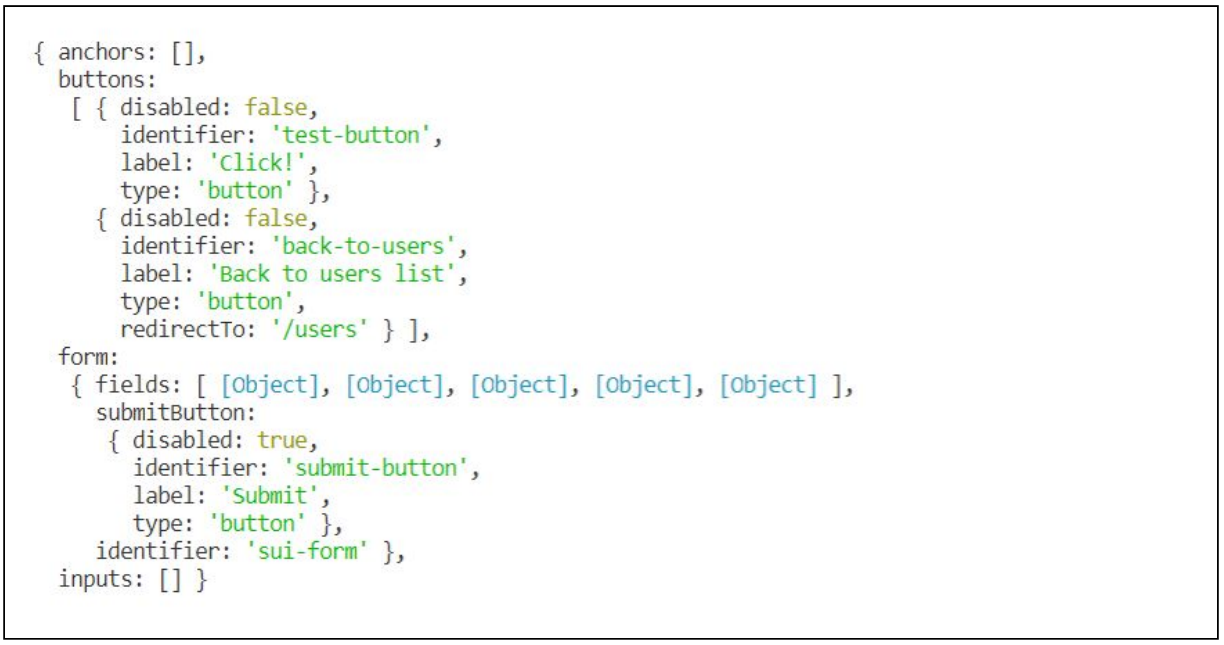
Element vs obiectul rezultat în urma transformării pentru un tag de HTML

[22] https://reactjs.org/docs/test-renderer.html

Element vs obiectul rezultat în urma transformării pentru o componentă din semantic-ui-react

După cum se poate observa mai sus, chiar dacă în pagina de browser cele două butoane se vor comporta exact la fel, există câteva diferenţe între rezultatul obţinut în urma transformării lor în obiecte, dintre care cea mai importantă pentru algoritmul nostru este valoarea proprietăţii *onClick*: în primul caz numele funcţiei apelate la click este menţionat, pe când în cel de-al doilea caz aceasta apare ca o funcţie anonimă. Din această cauză am decis ca numele funcţiei apelate să nu fie extras în această etapă de construire a arborelui de identificatori, ci direct în cadrul testului.

Cu menţiunea făcută mai sus, obiectul rezultat este parcurs şi se extrag elementele relevante. Pentru parcurgerea obiectului se foloseşte algoritmul de Depth First Traversal, pe baza căruia se realizează o serie de verificări. Acest tip de parcurgere este potrivit pentru algoritmul prezentat deoarece permite vizitarea tuturor nodurilor copil astfel încât să nu se altereze ierarhia arborelui generat. În acest fel se cunoaşte la fiecare pas lista de părinti ai nodului curent şi se poate stabili categoria în care trebuie încadrat. De exemplu, un buton care aparţine unui formular va fi tratat diferit faţă de restul butoanelor din aplicaţie. Similar, un buton cuprins într-un tag HTML *"<a href...>"* va trebui să preia de la părintele său acţiunea declanşată la click. În final, în funcţie de elementele identificate în componentă, arborele de identificatori va avea o structura asemănătoare cu aceasta:



Arborele de identificatori al unui formular

Dat fiind că în cadrul testului căutarea elementelor se face pe baza unui identificator, acest atribut trebuie specificat încă din componentă, atunci când este definit elementul. Identificatorul general recunoscut în cadrul algoritmului este “data-testid-ul”.

3.4.5. Crearea template-urilor pentru teste

Fişierul de test final este compus pe parcurs din multiple template-uri. Există template-uri mai specifice, dar şi template-uri generale, cum ar fi lista de librării şi fişiere care trebuie importată la începutul fiecărui fişier de test sau sintaxa fiecărui început de suită de teste:

```
return prettier.format(
  `
  describe(Automated Generated Tests for ${Component.name}', () => {
    let component;
    ${testDefaultProps(Component.defaultProps, defaultTestProps)}

    describe('With custom props', () => {
      beforeEach(() => {
        component = mount(
          <MemoryRouter>
            <Component ${templateProps} />
          </MemoryRouter>
        ).find('${Component.name}');
      });
      ${testRender()}
      ${testButtonsBehaviour(
        component,
        testRendererInstance,
        definedTestProps,
        buttonIdentifiers,
      )}
      ${testAnchorsBehaviour(identifiers)}
      ${testFormFields(
        component,
        testRendererInstance,
        definedTestProps,
        formatTemplateProps(defaultTestProps),
        identifiers,
      )}
    });
  });
`,
  { parser: 'babel' },
);
```

Template-ul general pentru suita de teste

Fragmentul de cod de mai sus reprezintă punctul de pornire al template-urilor care vor fi adăugate. După cum se poate observa, fişierul rezultat va avea următoarea structură: avem blocul cel mai exterior, care cuprinde toate testele generate automat pentru componenta dată, denumit "*Automated Generated Tests for <NumeComponentă>*". Primul test adăugat va fi cel pentru verificarea încărcării corecte a componentei folosind doar valorile *default* pentru *props*. Imediat după acest test se adaugă o nouă suită, pentru a grupa testele generate folosind pentru *props* o combinaţie de valori *default* şi valori trimise de programator. Deoarece suita va conţine mai multe teste, componenta este reîncărcată înainte de execuţia fiecărui test. În acest fel ne asigurăm că rezultatul unui test anterior nu influenţează execuţia testelor care urmează. Odată încărcată componenta, se începe adăugarea de teste. Template-ul returnat este formatat folosind librăria *prettier*.

3.4.6. Teste generate

După cum am menţionat într-un capitol anterior, programul prezentat ţine cont de două lucruri atunci când generează unit testele:

a. Verificarea faptului că este încărcată componenta pe baza *prop-urilor*

Programul returnează două teste pentru această situaţie, în funcţie de valorile pentru *props* pe care le primeşte. Există proprietăţi primite de la componenta părinte care sunt obligatorii pentru funcţionarea corectă a componentei copil. Acestea sunt marcate corespunzător şi nu au nevoie de o valoare *default* definită în cadrul componentei copil.

Pentru valorile trimise de componenta părinte care pot lipsi în anumite cazuri sunt adăugate valori *default*. Să luam ca exemplu o componentă care returnează un formular. Acest formular este folosit atât pentru a adăuga un utilizator nou cât şi pentru a edita unul existent. În acest caz, metoda apelată la trimiterea formularului este obligatorie, însă detaliile despre utilizator lipsesc în cazul în care suntem pe formularul de creare şi nu pe cel de editare. Aşadar, aceste valori nu sunt trimise mereu de la componenta părinte, şi este adăugată o valoare *default* în componenta curentă.

În teste, primul caz verifică dacă componenta este încărcată corect folosind, pe lângă valorile obligatorii, valorile *default*. În cel de-al doilea caz, componenta este încărcată cu un set de date definite în locul celor *default*.

Am luat în calcul două posibilităţi pentru generarea valorilor de *props* de care are nevoie componenta în teste - generarea aleatorie a unor valori, ţinând cont de tipul lor descris în obiectul *propTypes* atribuit componentei - string, boolean, array, etc. sau folosirea unor date de testare din partea programatorului. Am ales cea de-a doua variantă din următorul motiv: unit testele pot fi foarte specifice, şi în unele situaţii se doreşte testarea unor valori exacte pentru a face un test cu adevărat util iar datele generate aleator nu oferă garanţia ca acoperă aceste cazuri speciale. Acest lucru nu este neapărat o limitare a aplicaţiei prezentate - şi unit testele scrise manual au nevoie de aceste valori de testare date de către programator. Am considerat că cel mai bun loc pentru definirea acestor date este chiar în cadrul fişierului în care se găseşte şi componenta, într-un obiect atribuit acesteia:

```
UsersIndex.testProps = {
  users: [
    {
      firstName: "Josh",
      id: 1,
      lastName: "Carter",
      email: "josh.carter@test.co",
      archived: false
    },
    {
      firstName: "Jamie",
      id: 2,
      lastName: "Smith",
      email: "jamie.smith@test.co",
      archived: true
    }
  ]
};
```

Definirea datelor de testare

În cadrul testului care verifică funcţionalitatea componentei folosind valorile default este necesar să fie trimise şi cele obligatorii. Pentru a construi acest set de date am pornit de la următoarea idee: obiectul *testProps* conţine valori definite pentru *prop-urile* obligatorii, însă poate să conţină valori şi pentru o parte din cele opţionale. Aşadar obiectul final de test va avea la baza valorile din *testProps*, însă pentru cheile comune din *testProps* şi *defaultProps* cele din urmă vor suprascrie valorile din *testProps*, şi în plus, vor fi adăugate eventualele valori care sunt declarate doar în *defaultProps*.

```
function setDefaultTestProps(testProps = {}, defaultProps = {}) {
  let defaultTestProps = testProps
    ? cloneDeep(Object.assign(cloneDeep(testProps), cloneDeep(defaultProps)))
    : defaultProps;
  return defaultTestProps;
}
```

Construirea obiectului *props* pentru testarea valorilor default

Pentru crearea obiectului *props* pentru testele care se folosesc de valorile definite de programator (în obiectul *testProps*) se foloseşte reciproca expresiei prezentate mai sus.

b. Verificarea răspunsului componentei la acţiunile utilizatorului

Testarea diverselor evenimente care sunt declanşate în aplicaţie este a doua zonă în care programul oferă suport şi s-au avut în vedere în special evenimentele de tip *onClick şi onChange*.

Pentru *onClick* se folosesc metodele propuse de Enzyme. Paşii care trebuie urmaţi pentru a verifica răspunsul componentei la *onClick* sunt diferiţi în funcţie de modul în care

este declarată metoda - fie în cadrul componentei, fie în componenta părinte. Programul descris acoperă ambele situaţii:

1. Testarea metodelor trimise prin *props* şi apelate la *onClick*

Dacă metoda este definită în componenta părinte acest lucru trebuie să se reflecte în lista de *prop-uri* (*defaultProps* - dacă poate să lipsească sau *testProps* - dacă este marcată ca fiind obligatorie în cadrul componentei). Paşii pentru generarea testului în acest caz sunt:

- Crearea unei funcţii de test care să o înlocuiască pe cea originală
  Jest oferă o metodă, *jest.fn(),* care permite înlocuirea funcţiilor asupra cărora nu avem control cu obiecte care pot fi inspectate şi controlate.
- Înlocuirea metodei în componentă
  Acest lucru se face prin reîncărcarea componentei, de data aceasta cu valoarea de test în locul celei iniţiale. Astfel, de oriunde va fi refolosită, funcţia de test va fi apelată în locul celei originale.
- Identificarea elementului care declanşează acţiunea
  De cele mai multe ori elementul responsabil pentru acţiune este fie un buton, fie o ancora. Pentru a simula evenimentul original ne folosim de două din metodele propuse de Enzyme, *find()* şi *simulate()*: *component.find(selector).simulate("click");*
- Verificarea faptului ca răspunsul componentei este cel aşteptat
  În acest pas algoritmul verifică dacă metoda apelată la *onClick* a fost apelată.

```
function renderTestSuite(element, testProps, boundedMethod, action, hasPositiveAssertion) {
  const mockFunction = getMethodMockName(boundedMethod);
  return `
  it('tests the "${element.label}" button click', () => {
    const ${mockFunction} = jest.fn();
    ${mountReactComponentWithMocks(testProps, boundedMethod)}
    ${clickButton(element.identifier, action)}
    ${returnAssertion(mockFunction, hasPositiveAssertion)}
  });
  `;
}
```

Template pentru testarea funcţiilor trimise prin *props*

În fragmentul de mai sus sunt urmaţi exact paşii descrişi: este înlocuită funcţia originală cu cea de test - pentru a evidenţia faptul ca metoda folosită mai departe este diferită de cea iniţială am schimbat numele funcţiei de test aşa încât să înceapă cu cuvântul "mock". De exemplu, handleClick va fi înlocuită de o metodă mockHandleClick. Acest lucru este făcut în "getMethodMockName". Apoi este reîncărcată componenta, se simulează click-ul şi se adaugă rezultatele aşteptate. După cum se poate observa, "returnAssertion" primeşte ca şi parametru o valoare booleana - hasPositiveAssertion. Dacă este setată pe *true*, ne aşteptăm ca metoda să fie apelată; în caz contrar se fac presupuneri legate de motivul pentru care nu a fost apelată. Acest lucru este util în cazul formularelor, de exemplu, pentru a verifica dacă este valid sau invalid. Acest caz va fi descris mai târziu.

Template-ul de mai sus va genera într-un final un test care trece cu succes, asemănător cu cel din exemplul următor:

```
it('tests the "Handle Click" button click', () => {
  const mockHandleClick = jest.fn();
  component = mount(
    <MemoryRouter>
      <Component
        currentUser={{}}
        history={{ push: jest.fn() }}
        handleClick={mockHandleClick}
      />
    </MemoryRouter>
  );

  const button = component.find('button[data-testid="test-button"]');
  button.simulate("click");

  expect(mockHandleClick).toBeCalled();
});
```

Test generat pentru testarea metodelor trimise prin *props*

2. Testarea metodelor definite în cadrul componentei şi apelate la *onClick*

Pentru testarea acestui caz am ales o abordare uşor diferită. Şi în această situaţie avem nevoie de o metodă care permite controlul asupra funcţiei apelate. Cu toate ca varianta prezentată mai sus, care foloseşte *jest.fn()* pentru a obţine acest lucru, ar fi funcţionat şi aici, am considerat că mai potrivită ar fi o altă metodă oferită de Jest, şi anume *spyOn()*. Diferenţa dintre cele doua metode este că spyOn păstrează implementarea originală a funcţiei. În acest fel putem duce testul cu un pas mai departe: nu doar că verificăm dacă metoda a fost apelată ci putem face şi alte presupuneri, de exemplu să ne asigurăm ca obiectul *state* al componentei a fost modificat corespunzător după execuţia funcţiei. Acest lucru nu ar fi fost posibil cu *jest.fn()* deoarece această metodă oferă suport doar pentru suprascrierea funcţiei originale. Aşadar, paşii de urmat în acest caz sunt:

- Crearea unui "spion" pentru metoda de pe instanţa componentei
  Acest spion este ataşat direct metodei de pe prototipul componentei.
- Reîncărcarea componentei
  Pentru a "activa" spionul definit mai sus este necesară reîncărcarea componentei, de data aceasta fără a modifica nimic altceva în valorile pe care le consumă.
- Identificarea elementului care declanşează acţiunea
  Aceeaşi metodă aplicată pentru testarea funcţiilor primite prin props este folosită şi aici: elementul este identificat folosind *find()* iar evenimentul este simulat prin *simulate("click")*.
- Verificarea faptului ca răspunsul componentei este cel aşteptat
  Din nou, se verifică dacă metoda definită la *onClick* a fost apelată. Datorită metodei de testare folosită, funcţia originală chiar a fost executată în cadrul testului, aşadar putem

face şi alte presupuneri, legate de forma *state-ului*. Pentru a testa schimbările din state o posibilă soluţie ar fi fost compararea valorilor din state înainte şi după execuţia funcţiei testate, sau chiar şi doar a acelor valori care au fost modificate în cadrul testului. Problema cu această soluţie este ca programul nu are suficient control asupra noilor valori din state pentru a face această verificare eficientă. Asa ca am ales să mă folosesc de snapshot-uri pentru a valida elementele din state la acest pas. În acest fel, atunci când se generează testele este adăugat şi un fişier de snapshot care conţine elementele din *state* cu valorile lor actualizate, dacă este cazul. La următoarele execuţii ale testului respectiv se compară noul snapshot cu cel existent şi dacă există diferenţe atunci vom şti că ceva s-a schimbat în logica aplicaţiei.

```
function renderTestSuite(testProps, element, method, action, hasPositiveAssertion) {
  const templateProps = formatTemplateProps(testProps) || '';
  return `
  it('tests the "${element.label}" button click', () => {
    let spy;
    ${mockMethod(method, action, true)}
    ${mountComponent(templateProps)}
    ${clickButton(element.identifier, action)}
    ${returnAssertion(hasPositiveAssertion)}
    ${checkForStateUpdate()}
  });
  `;
}
```

Template pentru testarea metodelor din instanţa componentei

Fragmentul de cod de mai sus cuprinde etapele prezentate. Funcţiile folosite sunt similare cu cele pentru testarea metodelor trimise prin *props*, cu menţiunea că "mockMethod" va returna spionul ataşat componentei, şi nu o asignare folosind *jest.fn()*:

spy = jest.spyOn(Component.prototype, <numeMetodă>)

"checkForStateUpdate" este metoda responsabilă cu crearea snapshot-ului cu valorile din *state*, printr-o comandă de felul:

expect(component.find(Component).instance().state).toMatchSnapshot()

Template-ul de mai sus va genera într-un final un test care trece cu succes, asemănător cu cel din exemplul următor:

```
it('tests the "Handle Click" button click', () => {
  let spy;
  spy = jest.spyOn(Component.prototype, "handleClick");
  component = mount(
    <MemoryRouter>
      <Component currentUser={{}} history={{ push: jest.fn() }} />
    </MemoryRouter>
  );

  const button = component.find('button[data-testid="test-button"]');
  button.simulate("click");

  expect(spy).toBeCalled();
  expect(component.find(Component).instance().state).toMatchSnapshot();
});
```

Test generat pentru testarea metodelor din instanţa componentei

3. Testarea răspunsului componentei la *onChange*

Evenimentul de tip *onChange* poate să apară în diferite parţi ale aplicaţiei, atunci când utilizatorul face o schimbare a valorilor din câmpurile unui formular. Dat fiind ca formularele sunt destul de întâlnite în cadrul aplicaţiilor, pentru logarea sau crearea unui nou cont, pentru adăugarea unei entităţi noi într-o listă, completarea datelor pentru înscrieri online la diferite evenimente, etc. am considerat ca acesta este un caz destul de des întâlnit pentru a merita automatizarea procesului de creare a testelor necesare. Chiar dacă acum programul nu acoperă toate situaţiile, reuşeşte să completeze câmpurile unui formular şi să verifice dacă în urma acestor schimbări formularul este valid sau nu.

Încă de la crearea arborelui de identificatori, elementele unui formular sunt separate sub o cheie comună. În acest fel nu este nevoie să fie parcurse din nou toate elementele şi extragerea celor relevante. Pentru a decide care elemente aparţin unui formular, în timpul parcurgerii Depth First algoritmul actualizează mereu lista părinţilor pentru un element. Dacă se întâlneşte un element de tip *input* care are în această listă un formular atunci este adăugat în lista de câmpuri ale acelui formular. Similar, un buton care are ca şi părinte un formular şi este de tip *submit* nu va fi adăugat în lista generală de butoane identificate în componentă, ci în lista elementelor corespunzătoare formularului.

Programul returnează o nouă suită de teste care cuprinde două teste care se axează pe câmpurile obligatorii din formular: unul care verifică dacă, pentru datele introduse în câmpurile obligatorii, se apelează cu succes la *submit* metoda corespunzătoare, şi unul care verifică comportamentul componentei atunci când există erori de validare.

Încă de la începutul suitei se ia în calcul tipul metodei apelată la *onSubmit*. Aşa cum am prezentat în capitolele anterioare, testele au nevoie de o structură diferită dacă la *submit* este apelată o funcţie trimisă prin *props* sau din instanţa componentei. Pentru optimizarea rezultatului, template-ul pentru încărcarea componentei împreună cu valoarea de test a funcţiei (dacă este primită prin *props*) sau crearea spionului (dacă este vorba de o metodă definită în cadrul componentei) este adăugat o singură dată, la începutul suitei, cu ajutorul metodei *beforeEach()* din Enzyme.

```
return `
  describe('Form validation', () => {
    let field;
    ${isInstanceMethod ? 'let spy;' : `let ${mockFunction}`}
    beforeEach(() => {
      ${
        isInstanceMethod
          ? mockMethod(boundedMethod, isInstanceMethod)
          : `${mockFunction} = jest.fn();`
      }
      ${
        isInstanceMethod
          ? mountComponent(templateProps)
          : mountReactComponentWithMocks(testProps, boundedMethod)
      }
    });
```

Template pentru încărcarea formularului înaintea fiecărui test

Pentru testarea validității formularului se parcurge lista de câmpuri din arborele de identificatori și fiecare câmp este populat corespunzător, de exemplu: dacă inputul este de tip text, se populează cu un string "test", dacă este de tip email, se populează cu un string "value@test.co" iar dacă inputul este un checkbox, valoarea sa este schimbată cu opusul valorii curente. În acest fel se asigură corectitudinea datelor introduse. Următorul pas este simularea evenimentului de *submit* al formularului și verificarea că metoda corespunzătoare a fost apelată cu succes. Acest lucru se realizează prin metodele prezentate anterior în cadrul capitolelor de testare a metodelor definite în cadrul componentei părinte sau a componentei curente, așa că nu voi trece din nou prin pașii respectivi.

Dacă există câmpuri obligatorii în formular se adaugă și testul pentru a verifica comportamentul în cazul în care acesta este invalid. Pentru a obține un formular invalid, programul parcurge lista de câmpuri din arborele de identificatori și pentru fiecare câmp obligatoriu este simulat un eveniment de tip *onFocus* și unul de *onBlur*. Aceste acțiuni ar trebui să activeze erorile din formular pentru a face în continuare presupunerile legate de starea formularului. În acest caz, un test pentru a verifica dacă metoda corespunzătoare nu a fost apelată la *onSubmit* nu este util deoarece acea metodă va fi mereu apelată înaintea unei ultime validări a formularului, însă fără a se execută definiția ei. Așadar, am decis ca în acest test să adaug presupuneri legate de butonul de *submit* și de erorile care ar putea exista în pagină. În primul rând, programul verifică dacă inițial butonul de *submit* era inactiv, și dacă da, se adaugă în test presupunerea că și acum trebuie să fie în aceeași stare. Dacă butonul era activ la încărcarea componentei nu se adaugă această verificare deoarece cu siguranță va fi activ și după încercarea de a trimite un formular invalid. În acest caz, cel mai probabil în pagină au fost afișate erorile de validare, așa că se adaugă o verificare legată de acest lucru. Problema este că există modalități diferite de a trata erorile care ajung în interfața utilizatorului: unele aplicații pot să afișeze câte o eroare pentru fiecare câmp obligatoriu, pe când altele pot afișa o singură eroare pentru întreg formularul. Având în vedere acest lucru, am ajuns la idea că, în orice caz, există cel puțin o eroare în pagina și am decis să mă folosesc de

acest lucru adăugând în test presupunerea făcută: la finalul testului se verifică dacă există în componentă cel puţin un element care să aibă clasa *“error”*.

3.4.7. Rata de acoperire a codului prin testele generate

După ce testele generate au fost adăugate în fişier, programul mai are un singur pas de executat: rularea testelor obţinute, pentru a stabili rata de acoperire a codului prin analiza *coverage*. Desigur că această etapa ar fi putut fi omisă, lăsând la latitudinea programatorului decizia de a executa testele imediat după generare. Am considerat însă că în majoritatea cazurilor acest pas este util, aşa că l-am inclus în logica internă a aplicaţiei.

Deoarece comanda de start a programului corespunde execuţiei directe a unui script din pachet, rularea testelor nu a putut fi făcută printr-o comandă din terminal, aşa cum se întaâmplă în toate cazurile când programatorul execută testele direct din aplicaţia React. Pentru a face totuşi acest lucru posibil, am abordat o metodă mai puţin întâlnită: am importat librăria Jest în fişierul meu şi am apelat direct o funcţie internă din cadrul acesteia:

jest.runCli(<opţiuniDeConfigurare>, <caleaCatreProiect>)

Dacă în cazul rulării librăriei direct de la linia de comandă sau printr-un script putem adăuga calea către un fişier de configurare pentru Jest sub forma unui argument de felul *--config=jest.config.json,*, aici aceste opţiuni trebuie trimise ca şi parametru direct al funcţiei, împreună cu calea directorului în care se găseşte proiectul din care se execută testele.

Am ales să trimit doar opţiunile de configurare necesare pentru a rula strict fişierele returnate în cadrul execuţiei curente a programului, pentru a pune mai bine în evidenţă rezultatele obţinute.

4. Rezultate obţinute

4.1. Rezultate

Pentru a testa utilitatea pachetului în practică am creat o aplicaţie React care simulează o aplicaţie de gestionare a unei liste de utilizatori. Pentru aceasta, am adăugat o pagină principală, unde sunt listaţi sub formă de tabel toţi utilizatorii înregistraţi. Această pagină conţine de asemenea un câmp de căutare şi un buton pentru adăugarea unui nou utilizator. La selectarea butonului de adăugare sau a numelui unui utilizator aplicaţia redirecţionează spre un formular. Câmpurile din acest formular sunt goale dacă este vorba de formularul de adăugare a unui utilizator, respectiv populate pentru formularul de editare.

Aplicaţia este implementată folosind versiunea curentă a librăriei React la momentul scrierii lucrării (16.8.6), foloseşte librăria *react-router-dom*[23] pentru schimbarea rutelor din cadrul aplicaţiei şi librăria *semantic-ui-react* ca sursă a componentelor folosite. Aşadar, deşi aplicaţia este folosită doar pentru a demonstra utilitatea pachetului, este construită folosind unele dintre cele mai populare librării utilizate în aplicaţiile React din producţie. Nu am implementat şi integrarea cu backend-ul deoarece acest lucru este în afara scopului prezentării: pachetul descris în această lucrare generează unit teste pentru interfaţa unei aplicaţii; orice se întâmplă în interacţiunea cu partea de backend după ce utilizatorul dă click, de exemplu, pe butonul de trimitere formular este tratat separat de alte unit teste, cu o structură mult mai uşor de controlat.

În continuare sunt prezentate rezultatele obţinute după generarea unit testelor pentru fiecare din cele două pagini.

Pagina de listare:

Pagina de listare este compusă din elemente uşor de identificat de aplicaţia noastră, de exemplu: componente <*Link* /> din librăria semantic-ui-react pentru lista de utilizatori din tabel şi o componentă <*Button* /> care are ca şi părinte o componentă <*Link* />. În acelaşi timp, se testează comportamentul componentei cu *prop-uri default* pentru lista de utilizatori dar şi cu o listă de utilizatori trimisă prin *testProps*. După cum se poate observa mai jos, testele create au fost suficiente pentru a acoperi întreaga logică a acestei pagini:

[23] https://reacttraining.com/react-router/web/guides/quick-start

- teste generate:

```
PASS tests/UsersIndex.test.js
  Automated Generated Tests
    With default props
      √ renders correctly (4ms)
    With custom props
      √ renders correctly (18ms)
      √ redirects to "/users/add" on "Add new user" button click (24ms)
      √ tests that redirect works correctly on "Josh Carter" click (12ms)
```

- analiza *coverage*:

```
---------------|----------|----------|----------|----------|-------------------|
File           |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
---------------|----------|----------|----------|----------|-------------------|
All files      |      100 |      100 |      100 |      100 |                   |
 UsersIndex.js |      100 |      100 |      100 |      100 |                   |
---------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        4.98s
```

Formularul de adăugare/editare

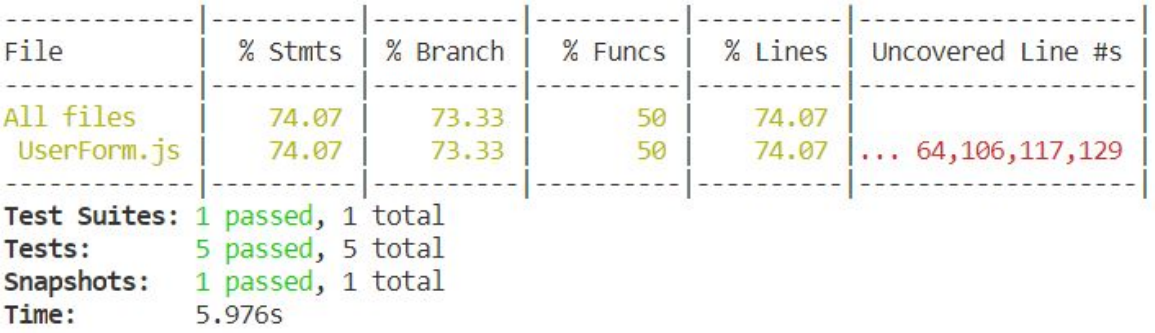
Componenta responsabilă de afişarea formularului are o implementare puţin mai complicată decât cea responsabilă de listare, prezentată mai sus. Există evenimente de tip *onClick* (pentru butonul care redirecţionează utilizatorul la pagina de listare), *onChange* (pentru modificarea valorilor din câmpuri) şi *onSubmit* (pentru trimiterea formularului).

- teste generate:

```
PASS tests/UserForm.test.js
  Automated Generated Tests
    With default props
      √ renders correctly (3ms)
    With custom props
      √ renders correctly (19ms)
      √ redirects to "/users" on "Back to users list" button click (30ms)
      Form validation
        √ tests Form Fields - success (90ms)
        √ tests Form Fields - failure (30ms)

 › 1 snapshot written.
```

- analiza *coverage*:



```
-------------|----------|----------|----------|----------|-------------------|
File         |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-------------|----------|----------|----------|----------|-------------------|
All files    |    74.07 |    73.33 |       50 |    74.07 |                   |
 UserForm.js |    74.07 |    73.33 |       50 |    74.07 |... 64,106,117,129 |
-------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        5.976s
```

Observaţii rezultate în urma generării testelor pentru această componentă:

- cu toate că această componentă este folosită şi pentru adăugarea şi pentru editarea unui utilizator, în generarea testelor pentru validarea formularului sunt luate în considerare doar valorile default pentru *props*, aşadar verificările se fac, în acest caz, pentru formularul de adăugare. Din această cauză, în unele situaţii nu sunt acoperite ambele ramuri pentru o expresie *if/else,* ceea ce duce la o scădere a ratei de acoperire pentru coloana "%Branch". Acesta poate fi totuşi considerat un caz special. În funcţie de scopul aplicaţiei care foloseşte react-unit-test-generator, rata de acoperire a formularelor poate să difere.

4.2. Limitări ale aplicaţiei

Există câteva situaţii în care algoritmul nu poate genera unele teste pentru elementele extrase din componentă. Pentru tratarea acestor situaţii am luat în calcul şi posibilitatea de a returna programatorului aceste teste împreună cu cele care au putut fi create cu succes. Problema cu această variantă este că programatorul trebuie mai apoi să parcurgă fişierul de teste şi să le modifice pe cele incomplete. Aşadar am ales să nu expun astfel erorile care pot să apară în generarea testelor, ci să le includ în rezultatul final sub forma unui raport al execuţiei. Acest raport este afişat după ce sunt create şi rulate restul testelor şi are avantajul că nu întrerupe execuţia algoritmului pe parcurs şi în acelaşi timp oferă o informaţie detaliată programatorului despre motivul pentru care testul respectiv nu a putut fi creat. Astfel, acesta are posibilitatea de a modifica codul sursă pe baza rezultatelor date de program iar la următoarea rulare testul va fi adăugat cu succes.

Pentru implementarea curentă a programului acestea sunt limitările impuse pentru a funcţiona corect:

- Elementele care nu au un *data-testid* nu vor putea fi testate
  Programul recunoaşte elementele din componentă pe baza unui identificator. În acest caz am ales să folosesc *data-testid-ul*. Iniţial, în etapa de creare a arborelui de identificatori toate elementele care sunt recunoscute de algoritm sunt adăugate în rezultatul final. În etapa de creare de teste specifice fiecăruia, se încearcă identificarea

fiecărui element în cadrul componentei pe baza *data-testid-urilor* din arborele de identificatori. În cazul în care identificatorul nu este definit, testul pentru elementul respectiv nu va fi creat.

- Butoanele care au definită *inline* metoda apelată la click nu vor putea fi testate
  În React există trei variante de a defini metoda care va fi apelata la click:
  1. metoda este definita în acelaşi fişier în care se găseşte componenta, şi apelată la *onClick*
  2. metoda este definita într-un părinte al componentei şi trimisă prin *props* la componenta copil, unde va fi apelată la *onClick*
  3. metoda este definită *inline*, la *onClick*.
     În ultimul caz, funcţia astfel definită va fi una anonimă. Din acest motiv, în cadrul testului nu avem nicio referinţă la numele funcţiei pentru a putea adăuga presupunerile ca aceasta a fost apelată.

Tabelul afişat în terminal este creat cu ajutorul unei librării numite *cli-table*[24]. Pentru a afişa aceste detalii împreună cu restul informaţiilor despre teste oferite de Jest a fost nevoie să creez un *custom reporter*, pe care mai apoi l-am inclus în opţiunile de configurare pentru Jest. Noţiunea de *custom reporter* este specifică librăriei Jest şi reprezintă o clasă care implementează una sau mai multe din următoarele metode: *onStartRun*, *onTestRun*, *onTestResult* şi *onRunComplete*, care se vor apela atunci când este declanşat evenimentul pe care îl reprezintă. În cazul nostru am folosit *onRunComplete*, pentru a afişa tabelul la finalul întregii execuţii.

Raportul pentru excepţiile returnate de programul nostru va arata astfel:

Test	Reason for skipping
Attempt to test the "Handle Click" button click	Inline onClick declarations are not supported
Attempt to test the click event for the "Back to users list" button	No identifier specified
Attempt to test the form functionallity	No identifier specified

Exemplu de raport al testelor care nu au putut fi create

[24] https://www.npmjs.com/package/cli-table

Concluzii

Consider că am reuşit să ating obiectivul propus pentru acest proiect, şi anume să construiesc o aplicaţie de generare a unit testelor care să fie independentă de tehnologiile folosite într-un proiect React. După cum s-a putut observa şi în capitolele prezentate anterior, pachetul react-unit-test-generator este capabil să returneze suite de teste care acoperă minim 50% din logica de implementare. De asemenea, facilitează procesul de *debugging în* cazul în care este întâlnită o eroare internă, prin optimizarea mesajelor de eroare aşa încât programatorul să identifice cât mai uşor cauza pentru care aplicaţia şi-a oprit execuţia. Este uşor de configurat (poate funcţiona doar cu opţiunile de configurare implicite, dacă structura proiectului este cea clasică) şi orientat spre testarea evenimentelor declanşate la interacţiunea cu utilizatorul. Toate acestea fac din acest pachet un pachet promiţător, cu atât mai mult cu cât este accesibil şi pe registrul npm. Deoarece încă nu există o librărie completă care să genereze automat unit testele necesare unei aplicaţii React (sau chiar Javascript) şi dat fiind că este nevoie uneori ca unit testele să fie foarte specifice, aplicaţia prezentată aduce o îmbunătăţire considerabilă a acestui proces şi sunt de părere că ar putea reprezenta o sursă de inspiraţie pentru abordările viitoare ale acestui domeniu.

Posibile îmbunătăţiri şi planuri de viitor

După cum a fost menţionat încă de la începutul lucrării, aplicaţia prezentată asigură testarea celor mai comune funcţionalităţi întâlnite într-un proiect React. Aceasta poate fi dezvoltată în continuare pentru a acoperi prin unit teste întreaga implementare a unei aplicaţii de producţie. Pe termen scurt, se poate adăuga suport şi pentru testarea altor elemente sau interacţiuni ale aplicaţiei cu utilizatorul, de exemplu testarea câmpurilor de tip *dropdown* sau *multiselect*. În mod similar, logica aplicaţiei poate fi extinsă pentru a verifica şi schimbările vizuale declanşate de evenimentele de tip *onClick şi onChange*. Desigur, este vorba doar de o chestiune de timp şi priorităţi până când îmbunătăţirile menţionate mai sus vor fi implementate. Deoarece pachetul este publicat pe npm, există şi posibilitatea ca acesta să ajungă în atenţia altor persoane interesate de acest domeniu şi care să dorească să contribuie prin adăugarea de noi funcţionalităţi.

Bibliografie

Articole de specialitate

[1] Ron Jeffries, Grigori Melnik. 2007. TDD--The Art of Fearless Programming. *IEEE Software*, vol. 24, no 3, pp 24 - 30
https://ieeexplore.ieee.org/document/4163024

Articole tehnice

[2] Unit Testing Tutorial: What is, Types, Tools, EXAMPLE
https://www.guru99.com/unit-testing-guide.html#1

[3] Using Jest and Enzyme for testing React Apps
https://hackernoon.com/using-jest-and-enzyme-for-testing-react-apps-4d3e1543bf0d

[4] Components testing in React: what and how to test with Jest and Enzyme.
https://www.freecodecamp.org/news/components-testing-in-react-what-and-how-to-test-with-jest-and-enzyme-7c1cace99de5/

[5] Auger - Automatic Unit Test Generation for Python
http://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html

[6] Testing in React with Jest and Enzyme: An Introduction
https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8

[7] How to use ReactJS with Webpack 4, Babel 7, and Material Design
https://www.freecodecamp.org/news/how-to-use-reactjs-with-webpack-4-babel-7-and-material-design-ff754586f618/

Documentaţii oficiale

[8] React - https://reactjs.org/

[9] React / Components and Props - https://reactjs.org/docs/components-and-props.html

[10] Randoop - https://randoop.github.io/randoop/

[11] Auger - https://github.com/laffra/auger

[12] JS-Test-Gen - https://github.com/js-test-gen

[13] BabelJS - https://babeljs.io/

[14] Prettier - https://prettier.io/

[15] Snappify - https://github.com/denisraslov/snappify

[16] Jest - https://jestjs.io/

[17] Snapshot Testing - https://jestjs.io/docs/en/snapshot-testing

[18] Enzyme - https://airbnb.io/enzyme/

[19] Semantic UI React - https://react.semantic-ui.com/

[20] Test Renderer - https://reactjs.org/docs/test-renderer.html