

**UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

**GENERARE AUTOMATĂ DE UNIT TESTE PENTRU APLICAȚII
REACT**

propusă de

Oana Juravle

Sesiunea: *iulie, 2019*

**Coordonator științific
Conf. Dr. Mădălina Răschip**

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ**

**GENERARE AUTOMATĂ DE UNIT TESTE PENTRU APLICAȚII
REACT**

Oana Juravle

Sesiunea: *iulie, 2019*

**Coordonator științific
Conf. Dr. Mădălina Răschip**

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemnatul(a) _____
domiciliul în _____
născut(ă) la data de _____, identificat prin CNP _____,
absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de
_____ specializarea _____, promoția
_____, declar pe propria răspundere, cunoscând consecințele falsului în
declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr.
1/2011 art.143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul:

_____ elaborată sub îndrumarea dl. / d-na
_____, pe care urmează să o
susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

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

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări
științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări
de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că
lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data azi, _____

Semnătură student _____

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Generare Automată de Unit Teste pentru Aplicații React”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc 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 Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent Oana Juravle

(semnătura în original)

| | |
|--|-----------|
| Introducere | 6 |
| 1. Descrierea problemei | 7 |
| 1.1. Ce este ReactJS? | 7 |
| 1.2. Unit testing | 8 |
| 1.2.1. Ce sunt unit testele? | 8 |
| 1.2.2. Care sunt avantajele unit testelor? | 9 |
| 1.2.3. Care sunt limitările unit testelor? | 9 |
| 1.2.4. Ce trebuie testat? | 10 |
| 1.3. Implementări existente | 10 |
| 1.4. Contribuții proprii | 12 |
| 2. Tehnologii folosite | 14 |
| 3. react-unit-test-generator | 17 |
| 3.1. Ideea | 17 |
| 3.2. Structura proiectului | 17 |
| 3.3. Publicarea aplicației ca pachet pe npm | 18 |
| 3.4. Descrierea soluției | 19 |
| 3.4.1. Setup | 19 |
| 3.4.2. Identificarea fișierului pentru care se vor genera testele | 20 |
| 3.4.3. Crearea fișierului destinație | 21 |
| 3.4.4. Crearea arborelui de identificatori | 21 |
| 3.4.5. Crearea template-urilor pentru teste | 24 |
| 3.4.6. Teste generate | 25 |
| a. Verificarea faptului că este încărcată componenta pe baza prop-urilor | 25 |
| b. Verificarea răspunsului componentei la acțiunile utilizatorului | 26 |
| 3.4.7. Analiza coverage a testelor generate | 32 |
| 4. Rezultate obținute | 33 |
| 4.1. Rezultate | 33 |
| 4.2. Limitări ale aplicației | 35 |
| Concluzii | 37 |
| Posibile îmbunătățiri și planuri de viitor | 37 |
| Bibliografie | 38 |

Introducere

În contextul dezvoltării web al zilelor noastre testarea joacă un rol foarte important în dezvoltarea aplicațiilor. Însă este dificil pentru un programator să scrie întreaga funcționalitate a unei aplicații fără a-i scăpa din vedere vreun detaliu și pe termen lung scrierea de unit teste ajunge să consume mult din timpul și resursele alocate unei aplicații. În timp, toate acestea fac din unit teste un task plictisitor și 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 aplicație scrisă folosind ReactJS. Pe parcursul implementării am avut în vedere cele mai comune funcționalități dintr-un proiect, pentru a avea rezultate satisfăcătoare indiferent de scopul proiectului în care este folosit acest program. În cadrul unei aplicații 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 și pe partea de client pentru a testa interacțiunea dintre utilizator și aplicație. Am ales ca acest proiect să se focuseze pe testarea care trebuie efectuată pentru a asigura buna funcționare a aplicației la interacțiunea cu utilizatorul, mai exact pe modul în care răspunde aplicația la acțiuni 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 conținutului unei pagini însă este capabil să facă diferite presupuneri asupra acțiunilor care determină aceste schimbări. Ulterior, programatorul poate revizui rezultatul obținut în urma generării și să adauge afirmațiile legate de schimbările vizuale care apar în conținut. De exemplu, dacă avem un buton care deschide un modal, aplicația poate să identifice butonul și să testeze ca funcția corespunzătoare a fost într-adevăr apelată la *onClick* (deci la nivel de cod implementarea este corectă) însă poate fi util și să verificăm dacă modalul este afișat în pagină. În acest caz, fișierul rezultat poate fi editat manual pentru a adăuga această verificare la finalul testului corespunzător.

În continuare, în primul capitol vor fi prezentate câteva noțiuni principale ale librăriei ReactJS pentru a oferi context pentru cele ce urmează. De asemenea, voi vorbi despre unit teste, împreună cu avantajele și limitările lor, dar și despre pașii care trebuie urmați în crearea unit testelor pentru a avea o suită de teste care oferă garanția că acoperă cele mai importante părți ale aplicației. Spre finalul capitolului sunt prezentate o serie de aplicații existente la momentul actual care încearcă să ușureze procesul de testare, urmate de un subcapitol care cuprinde contribuțiile proprii. Cel de-al doilea capitol al lucrării prezintă tehnologiile care au stat la baza implementării aplicației. În capitolul trei este descrisă metoda abordată pentru generarea automată a unit testelor. Capitolul patru prezintă analiza rezultatelor aplicației în practică.

1. Descrierea problemei

1.1. Ce este ReactJS?

ReactJS¹ este o librărie de Javascript utilizată pentru a crea interfețe/aplicații web care nu necesită reîncărcarea paginii web în momentul în care datele se modifică (*single-page apps*). Acest lucru face ca ReactJS să fie una dintre cele mai puternice librării la momentul actual, fiind în aceleași timp simplu de înțeles și ușor de testat. A fost publicată în 2013 de echipa celor de la Facebook.

ReactJS are la bază componente². O componentă este un modul care returnează un anumit rezultat. Conceptual, o componentă este o funcție javascript: acceptă un set de date de intrare(în acest caz numite “props”) și returnează un element React. Aceasta poate conține, la rândul său, una sau mai multe subcomponente. Componentele permit structurarea aplicației în părți independente și reutilizabile, de exemplu: butoane, formulare, tabele, etc.

În ReactJS componentele pot fi definite în două moduri:

- Componente funcționale (*Functional Components*)

După cum sugerează și numele, aceste componente sunt doar funcții care acceptă un singur obiect drept argument - props și returnează un element React. În cadrul componentelor funcționale 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 adiționale, printre care conceptul de stare și cel de “ciclu de viață” al componentei(*component lifecycle*). Acesta este compus dintr-o serie de metode speciale care oferă control asupra componentei și care se execută automat în diferite etape ale procesului - inițializare, încărcare în DOM(*mounting*), actualizare, ștergere din DOM(*unmounting*). Orice componentă trebuie să implementeze o metodă numită “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 folosiți 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 menționat despre props este că valorile acestora pot fi accesate doar pentru a fi citite, nu și modificate.

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

¹ <https://reactjs.org/>

² <https://reactjs.org/docs/components-and-props.html>

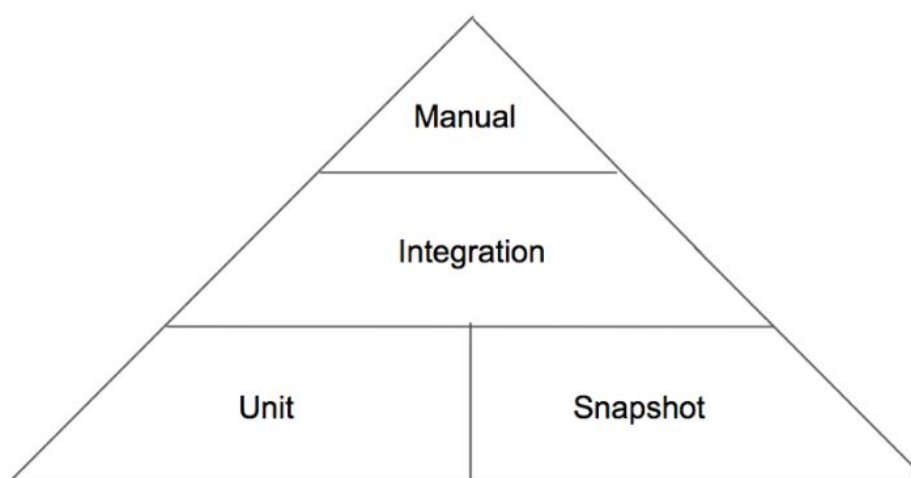
care este definit. În acest obiect sunt păstrate date care se pot schimba - de exemplu, datele dintr-un formular sau date filtrate în urma unei căutări/sortări.

1.2. Unit testing

1.2.1. Ce sunt unit testele?

Unit testele reprezintă primul nivel pentru testarea unei aplicații - în cadrul unit testelor sunt testate izolat mici funcționalități /componente specifice ale aplicației³.

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



Piramida de testare⁴

Unit testele aparțin abordării de testare numite “White Boxing” deoarece testele sunt adăugate după ce funcționalitatea 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 funcționalitate, folosind date de testare separate de cele folosite de echipa de testare.

Pentru a avea garanția ca unit testele acoperă într-adevăr părțile importante din componenta testată, în general se folosește o altă tehnică de White Boxing, și anume “Code Coverage”-ul. Analiza code coverage identifică ce funcționalități nu au fost incluse în suita de

³ <https://www.guru99.com/unit-testing-guide.html#1>

⁴ <https://hackernoon.com/using-jest-and-enzyme-for-testing-react-apps-4d3e1543bf0d>

teste corespunzătoare. Aceasta oferă informații referitoare la Statement Coverage - verifică dacă toate liniile de cod sunt parcurse de suita de teste cel puțin o dată, informații referitoare la Branch Coverage - verifică dacă sunt acoperite toate ramurile *if/else* precum și 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) și la fixarea lor imediată, cu un cost minim, fără a impacta alte părți ale proiectului. Conform studiului “TDD--The Art of Fearless Programming”⁵ publicat în IEEE Software rata de depistare a defectelor în producție poate fi scăzută cu 40%-80% pentru proiecte care au la baza o structură bine definită de unit teste.

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

Unit testele pot servi drept documentație pentru aplicația în dezvoltare. Dat fiind că o suită completă de teste trebuie să includă toate cerințele pentru o anumită parte de aplicație, oferă o mai bună înțelegere 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 funcționale sunt ușor de scris și asigură acoperire totală deoarece o componentă funcțională returnează, pentru același set de date de intrare, același rezultat și nu există efecte secundare. Așadar 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, crește și numărul de unit teste necesare, acestea devenind tot mai complexe și costisitoare atât ca și timp cât și ca și cerințe financiare. Cerințele proiectului pot include menținerea nivelului de code coverage destul de mare(80-90%), uneori ajungând și la 100%. Acest lucru înseamnă o scădere a ratei de implementare pentru funcționalități noi, dat fiind că programatorii trebuie să acorde timp suplimentar pentru scrierea unit testelor.

Un alt dezavantaj este faptul că pentru funcționalități comune însă folosite în componente diferite aceeași logică de testare este folosită în locuri multiple. Putem lua ca

⁵ <https://ieeexplore.ieee.org/document/4163024>

exemplu testarea unui formular - în general, pașii pentru validarea unui formular presupun popularea câmpurilor din formular și verificarea că aceste valori sunt salvate la *submit*.

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 la render;
- Dată fiind acțiunea 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 dispoziție de diferite pachete: în general, librăriile au deja propriile teste implementate pentru funcționalitatea pusă la dispoziție;
- Constantele: așa cum a fost menționat 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 fișier 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 funcționalitate implementată. Pentru tehnologii precum Java și Python există câteva unelte care oferă garanția faptului că testele obținute dau rezultate bune și sunt folosite chiar în cadrul diverselor companii. Acest lucru este posibil deoarece Java și Python sunt limbaje folosite în general pe backend, iar rezultatele funcțiilor implementate sunt destul de predictibile - primesc un set de date de intrare și returnează un set de date de ieșire care eventual este folosit în cadrul altor funcții. Spre deosebire de Java și Python, metodele definite în Javascript și în mod special React se reflectă în interfața utilizatorului, ceea ce duce la o multitudine de rezultate posibile. Din această cauză este dificil să se facă presupuneri legate de conținutul unei pagini web prin intermediul unor teste generate automat.

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

6

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

Randoop⁷

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

Auger⁸

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, astfel⁹: pentru fiecare funcție definită în cadrul modulului, Auger extrage valorile argumentelor și a rezultatului returnat, după care este simulat un apel al acelei funcții cu argumentele și valoarea returnată înlocuite cu valorile de test corespunzătoare; pentru funcțiile apelate din diverse librării este extrasă valoarea returnată în urma execuției astfel încât în test valoarea returnată de același apel al funcției să poată fi înlocuită corespunzător.

JS Test Gen¹⁰

JS Test Gen este un pachet care generează template-uri de unit teste pentru funcții Javascript. Se folosește de babel¹¹ pentru a identifica modulele care sunt exportate din fișierul testat. Odată identificate, generează template-urile de test formate cu ajutorul unui pachet numit prettier¹². Din păcate, această librărie generează doar template-uri generale pentru funcțiile exportate dintr-un fișier, nu face niciun fel de presupunere asupra rezultatului actual obținut în urma execuției 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 obținut folosind JS Test Gen

Snappify¹³

Snappify este o librărie creată pentru testarea componentelor scrise cu React și TypeScript. Snappify se folosește de faptul că, folosind TypeScript, există o interfață care descrie tipul prop-urilor folosite în componenta respectivă. Pe baza acestor informații sunt generate valori aleatoare corespunzătoare prop-urilor și apoi se creează testele folosind Snapshots. O idee similară este abordată și în librăria *generator-react-jest-tests*¹⁴. Deși soluția propusă în această librărie are potențial, în proiectele reale este nevoie de teste care să verifice mai riguros funcționalitatea dată, iar testarea bazată doar pe snapshot-uri construite din combinații ale prop-urilor primite de un element nu asigură neapărat funcționarea corectă a aplicației, ci doar că între rulări succesive ale testelor nu au apărut modificări în comportamentul 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 obținut folosind Snappify

¹³ <https://github.com/denisraslov/snappify>

¹⁴ <https://www.npmjs.com/package/generator-react-jest-tests>

1.4. Contribuții 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 funcționalitatea 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. Mai exact, oferă suport pentru:

- verificarea comportamentului componentei atunci când sunt folosite doar prop-urile *default* dar și pentru încărcarea componentei cu o listă de prop-uri setate de cel care a implementat componenta respectivă. În acest fel și 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 acțiunea evenimentelor de *onClick* și *redirect* pentru orice element. Prin testarea acestor evenimente, se garantează faptul că la nivel de cod totul funcționează conform implementării și că interfața aplicației răspunde corespunzător în interacțiunea cu utilizatorul;
- validarea formulelor 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 eșuat. În acest fel programatorul este notificat de faptul că este posibil ca aplicația 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

Jest¹⁵ este una dintre cele mai folosite librării de testare la momentul actual și este dezvoltată de Facebook. Am ales această librărie pentru proiectul meu din motive de performanță dar și pentru că oferă o sintaxă foarte intuitivă și vine cu toate pachetele necesare astfel încât nu e nevoie să se instaleze librării adiționale pentru a avea o suită de teste eficiente. De asemenea, are nevoie de foarte puține configurări pentru a fi integrată în cadrul proiectului și 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, și în special în această aplicație - dat fiind că testele sunt generate automat, au nevoie de o descriere cât mai clară, care să ajute programatorul să vadă care sunt părțile de logică acoperite. Acest lucru se face cu ajutorul a trei metode¹⁶ oferite de librăria Jest:

- *describe()*
describe() este o metodă opțională folosită pentru a grupa un set de teste și conține un text ce reprezintă o descriere a ceea ce se testează în suită;
- *test()* / *it()*
test() este similar cu *describe()*. Conține 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 funcție a cărei evaluare trebuie să fie mereu adevărată, în caz contrar întreaga execuție a suitei se încheie și este returnată eroarea care a cauzat oprirea rulării testelor. *toEqual()* face parte dintr-o listă mult mai mare de comparatori, printre care și *toContain()*, *toBeCalled()*, *toHaveBeenCalledWith()*, etc.

De asemenea, Jest oferă posibilitatea de a executa părți 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-ul¹⁷ reprezintă un tip de testare ce aparține librăriei Jest și care verifică, la fiecare execuție a testelor, că nu apar schimbări neașteptate în interfața utilizatorului.

Atunci când testele rulează pentru prima dată, dacă funcția *toMatchSnapshots()* este întâlnită într-unul din teste - de cele mai multe ori după o metodă *expect()* - Jest va genera automat un fișier 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 fișier va fi salvată o reprezentare a componentei încărcate. Acesta poate fi validat sau editat pentru a asigura corectitudinea datelor inițial salvate.

La următoarele rulări ale testelor, Jest va încărca snapshot-urile deja existente și le va compara cu cele returnate de testul curent. Eventualele diferențe vor fi afișate 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 diferențele găsite. Acestea sunt mai apoi evaluate și dacă într-adevăr schimbările au fost intenționate se actualizează fișierul 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

Enzyme¹⁸ 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 funcții 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 și 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 dispoziție și 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

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

Spre exemplu, dacă componenta conține un tabel sortabil de utilizatori și 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 redirecționează spre altă pagină a aplicației putem lua și valoarea noului URL pentru a testa și această funcționalitate. Așadar, o suită completă de teste pentru acest exemplu ar putea conține:

- 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 funcționalitatea butonului de adăugare a unui utilizator nou.

3.2. Structura proiectului

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

- Directorul *src*

Aici se află toată logica pentru generarea template-urilor, printre care: fișierul de intrare (*index.js*), fișierul în care se află funcția responsabilă de generarea arborelui de identificatori (*createIdentifiersMap.js*), precum și alte două subdirectoare: *templates* și *helpers*. După cum îi spune și numele, primul subdirector cuprinde toate template-urile din care este alcătuit fișierul de teste returnat. Aceste template-uri sunt organizate, la rândul lor, în alte subdirectoare, pe baza funcționalităților 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* - conține funcții ajutătoare care sunt folosite în mai multe locuri. În acest fel se respectă principiul “Don’t repeat yourself” (DRY) care spune că logica ce se repetă într-un proiect trebuie abstractizată.

- Directorul *dist*

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

- Directorul *node_modules*

Și acest director este creat și actualizat automat de fiecare dată când se adaugă un pachet nou în aplicația 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 funcție dintr-un pachet instalat, definiția ei va exista în directorul `node_modules` și 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` și prin executarea unei comenzi `“npm install”` fiecare pachet este descărcat și adăugat în acest director.

- `package.json`

Acest fișier este creat încă de la început, atunci când se inițializează aplicația printr-o comandă `“npm init”`. În urma executării acestei comenzi sunt adăugate o serie de câmpuri în fișierul `package.json`, dintre care:

- numele aplicației: `react-unit-test-generator`
- versiunea curentă a aplicației
- un câmp `“main”` corespunzător fișierului 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”` și `“devDependencies”` sunt obiecte care fac referire la pachetele instalate în proiect, sub forma `“<NumePachet>: <VersiuneInstalată>”`. Diferența dintre cele două obiecte este dată de scopul pentru care au fost instalate pachetele: dacă un pachet este folosit doar pe parcursul dezvoltării aplicației 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 aplicației ca pachet pe npm

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

Modul cel mai ușor prin care o astfel de aplicație ajunge să fie folosită într-un proiect real este prin publicarea ei ca și pachet pe *npm*¹⁹ (Node Package Manager), în acest fel devenind accesibilă pentru oricine dorește 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. Așadar, am decis să public aplicația prezentată în primul rând pentru

¹⁹ <https://www.npmjs.com/>

a-i demonstra utilitatea într-un mediu practic și în al doilea rând pentru a-i crește gradul de accesibilitate așa încât să poată fi folosită de oricine dorește să o includă în proiectele proprii.

La momentul scrierii acestei lucrări, pachetul se află la versiunea 1.9.1 și a fost descărcat de 672 ori.

The screenshot shows the npm package page for **react-unit-test-generator**. At the top, it indicates version 1.9.1, is public, and was published 4 minutes ago. Navigation tabs include Readme, Admin, 21 Dependencies, 0 Dependents, and 30 Versions. The main description states it's a helper for writing unit tests for React apps. The 'Install' section shows the command `npm install --save-dev react-unit-test-generator`. The 'Keywords' section lists: `jest snapshot react test testing component generator`. On the right, a sidebar shows the install command, 672 weekly downloads, version 1.9.1 with ISC license, 0 open issues, 0 pull requests, homepage at github.com, repository on github, and last published 4 minutes ago.

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 roboți care îl analizează și se asigură că este sigur, dar și de portaluri care descarcă pachetul pentru a-l expune pe propriul lor domeniu. Cu toate acestea, putem aproxima că, deși pachetul nu era la o versiune finală, pe parcursul implementării a atras atenția unor dezvoltatori care au considerat ca le-ar fi util un astfel de pachet în proiectele lor.

3.4. Descrierea soluției

3.4.1. Setup

Dat fiind că aplicația 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 și nu influențează comportamentul aplicației expuse utilizatorului, este recomandat ca acesta să fie instalat în

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

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

“<NumeComanda>: react-unit-test-generator”

Prin execuția pachetului ca și comandă în acest fel se crează două variabile în obiectul global *process.env* de care ne vom folosi pentru a obține informații despre aplicația care folosește acest pachet - *npm_package_name* și despre fișierul pentru care se vor genera unit testele - *npm_config_filename*.

Această comandă poate primi un argument opțional, adăugat în scriptul din fișierul *package.json*: *--config=<NumeFișierDeConfigurare>*. Fișierul de configurare trebuie să fie adăugat în rădăcina aplicației și să conțină 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 fișierul cu testele returnate: { “destination”: “./root/tests” }

Dacă acest argument nu este adăugat, se consideră ca aplicația are structura cea mai comună, în care componentele se găsesc într-un director *src* și testele trebuie adăugate într-un director *tests*, ambele în rădăcina aplicației. Dacă este adăugat, se verifică dacă acest fișier există și dacă conținutul său poate fi accesat. În caz de eroare, aceasta este afișată în terminal și programul își oprește execuția.

3.4.2. Identificarea fișierului pentru care se vor genera testele

După cum am menționat mai sus, programul își va începe execuția printr-o comandă adăugată în fișierul *package.json* și executată de la linia de comandă sub forma:

“npm run <NumeComandă> --fileName=<NumeFișier>”

Parametrul pe care îl primește acest script - *NumeFișier* - este obligatoriu și poate fi numele fișierului care conține definiția componentei sau calea(completă ori parțială) către fișierul respectiv.

La acest pas este validat fișierul. Aici, dar și 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 informații utile în corectarea ei. Așadar, se verifică dacă comanda a fost executată cu argumentul corespunzător numelui fișierului; în caz contrar, un mesaj specific va fi afișat în terminal.

După ce se validează numele fișierului de configurare (dacă este cazul) și calea către fișierul din datele de intrare, programul parcurge recursiv toate fișierele din directorul dat ca și punct de intrare în aplicație și returnează calea către acele fișiere 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 conține ‘/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 fișierului respectiv nu a returnat cel puțin un element, utilizatorul va fi informat despre acest lucru.

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

3.4.3. Crearea fișierului destinație

Următorul pas este cel de creare a fișierului destinație, care se va adăuga în directorul de teste și va avea același nume ca și componenta testată, urmat de extensia *.test.js* (dacă fișierul de test cu acest nume există deja atunci conținutul 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 și tag-uri de HTML, cât și prin intermediul unei librării. Pentru exemplificare, am folosit librăria *semantic-ui-react*. Aceasta aduce o sintaxă diferită, stiluri și nivele de elemente suplimentare. Spre exemplu, componenta “Message” din *semantic-ui-react* transformată în HTML va arăta ca în exemplul de mai jos:

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

Diferențe de sintaxă între *semantic-ui-react* și HTML²⁰

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

²⁰ <https://react.semantic-ui.com/>

Prima abordare presupunea citirea fișierului în care este definită componenta și parcurgerea lui astfel: știind ca fiecare componentă declarată ca și clasă trebuie să conțină o metodă numită *render* - responsabilă cu returnarea conținutului care va fi afișat 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 funcției; chiar dacă în general metoda *render* începe și se termină cu acolade, în interiorul ei există și alte seturi de acolade de care trebuie ținut cont. Acest lucru este valabil mai apoi pentru fiecare metodă adăugată la evenimentele de tip *onClick* sau *onChange* - cu atât mai mult cu cât definiția acestora poate fi făcută și inline. Pentru a stabili unde începe și se termină fiecare element este nevoie de o verificare similară, și apoi apare și problema elementelor definite ca și tag-uri de HTML și componente luate din librăria menționată mai sus. Luând toate aceste lucruri în calcul plus problema parcurgerii fișierului în primul rând, am decis să mă îndrept spre altă soluție.

O altă metodă - cu care am hotărât să merg mai departe în implementarea algoritmului - presupune folosirea unei pachet oferit de React, *react-test-renderer*²¹. *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 că pentru același element din pagină, în funcție de cum a fost încărcat - doar HTML sau printr-o componentă din *semantic-ui-react* - obiectul rezultat poate arăta diferit:

| | |
|---|--|
| <pre><button data-testid="test-button" onClick={this.handleClick}> Example Button </button></pre> | <pre>{ type: 'button', props: { 'data-testid': 'test-button', onClick: [Function: bound handleClick] }, children: ['Example Button'] }</pre> |
|---|--|

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

| | |
|---|---|
| <pre><Button data-testid="test-button" onClick={this.handleClick}> Example Button </Button></pre> | <pre>{ type: 'button', props: { 'data-testid': 'test-button', className: 'ui button', 'aria-pressed': undefined, disabled: undefined, onClick: [Function], role: undefined, tabIndex: undefined }, children: ['Example Button'] }</pre> |
|---|---|

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

²¹ <https://reactjs.org/docs/test-renderer.html>

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

Cu mențiunea făcută mai sus, obiectul rezultat este parcurs și se extrag elementele relevante. Pentru parcurgerea obiectului se folosește 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 cunoaște la fiecare pas lista de părinți ai nodului curent și se poate stabili categoria în care trebuie încadrat. De exemplu, un buton care aparține unui formular va fi tratat diferit față de restul butoanelor din aplicație. Similar, un buton cuprins într-un tag HTML “<a href...>” va trebui să preia de la părintele său acțiunea declanșată la click. În final, în funcție de elementele identificate în componentă, arborele de identificatori va avea o structura asemănătoare cu aceasta:

```
{ anchors: [],
  buttons:
    [ { disabled: false,
      identifier: 'test-button',
      label: 'Click!',
      type: 'button' },
      { disabled: false,
        identifier: 'back-to-users',
        label: 'Back to users list',
        type: 'button',
        redirectTo: '/users' } ],
  form:
    { fields: [ [Object], [Object], [Object], [Object], [Object] ],
      submitButton:
        { disabled: true,
          identifier: 'submit-button',
          label: 'Submit',
          type: 'button' },
      identifier: 'sui-form' },
  inputs: [] }
```

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

Fișierul de test final este compus pe parcurs din multiple template-uri. Există template-uri mai specifice, dar și template-uri generale, cum ar fi lista de librării și fișiere care trebuie importată la începutul fiecărui fișier 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, fișierul 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 noua suită, pentru a grupa testele generate folosind pentru

props o combinație de valori default și valori trimise de programator. Deoarece suita va conține mai multe teste, componenta este reîncărcată înainte de execuția fiecărui test. În acest fel ne asigurăm că rezultatul unui test anterior nu influențează execuția 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 menționat într-un capitol anterior, programul prezentat ține 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 situație, în funcție de valorile pentru props pe care le primește. Există proprietăți primite de la componenta părinte care sunt obligatorii pentru funcționarea corectă a componentei copil. Acestea sunt marcate corespunzător și 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 luăm ca exemplu o componentă care returnează un formular. Acest formular este folosit atât pentru a adăuga un utilizator nou cât și 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 și nu pe cel de editare. Așadar, aceste valori nu sunt trimise mereu de la componenta părinte, și 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 posibilități pentru generarea valorilor de props de care are nevoie componenta în teste - generarea aleatorie a unor valori, ținând 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, și în unele situații se dorește testarea unor valori exacte pentru a face un test cu adevărat util iar datele generate aleator nu oferă garanția ca acoperă aceste cazuri speciale. Acest lucru nu este neapărat o limitare a aplicației prezentate - și 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 fișierului în care se găsește și 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 funcționalitatea componentei folosind valorile default este necesar să fie trimise și cele obligatorii. Pentru a construi acest set de date am pornit de la următoarea idee: obiectul de testProps conține valori definite pentru prop-urile obligatorii, însă poate să conțină valori și pentru o parte din cele opționale. Așadar obiectul final de test va avea la baza valorile din testProps, însă pentru cheile comune din testProps și defaultProps cele din urmă vor suprascrie valorile din testProps, și 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 folosește reciprocă expresiei prezentate mai sus.

b. Verificarea răspunsului componentei la acțiunile utilizatorului

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

Pentru *onClick* se folosesc metodele propuse de Enzyme. Pașii care trebuie urmați pentru a verifica răspunsul componentei la *onClick* sunt diferiți în funcție de modul în care

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

1. Testarea metodelor trimise prin props și 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). Pașii pentru generarea testului în acest caz sunt:

- Crearea unei funcții de test care să o înlocuiască pe cea originală
Jest oferă o metoda, *jest.fn()*, care permite înlocuirea funcțiilor asupra cărora nu avem control cu obiecte care pot fi inspectate și controlate.
- Înlocuirea metodei în componentă
Acest lucru se face prin reîncărcarea componentei, de data aceasta cu valoarea de test în locul celei inițiale. Astfel, de oriunde va fi refolosită, funcția de test va fi apelată în locul celei originale.
- Identificarea elementului care declanșează acțiunea
De cele mai multe ori elementul responsabil pentru acțiune este fie un buton, fie o ancora. Pentru a simula evenimentul original ne folosim de două din metodele propuse de Enzyme, *find()* și *simulate()*: *component.find(selector).simulate("click")*;
- Verificarea faptului ca răspunsul componentei este cel așteptat
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 funcțiilor trimise prin props

În fragmentul de mai sus sunt urmați exact pașii descriși: este înlocuită funcția originală cu cea de test - pentru a evidenția faptul ca metoda folosită mai departe este diferită de cea inițială am schimbat numele funcției de test așa 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 și se adaugă rezultatele așteptate. După cum se poate observa, “returnAssertion” primește ca și parametru o valoare booleană - hasPositiveAssertion. Dacă este setată pe *true*, ne așteptăm 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).toHaveBeenCalled();
});
```

Test generat pentru testarea metodelor trimise prin props

2. Testarea metodelor definite în cadrul componentei și apelate la onClick

Pentru testarea acestui caz am ales o abordare ușor diferită. Și în această situație avem nevoie de o metodă care permite controlul asupra funcției apelate. Cu toate ca varianta prezentată mai sus, care folosește *jest.fn()* pentru a obține acest lucru, ar fi funcționat și aici, am considerat ca mai potrivita ar fi o alta metoda oferita de Jest, și anume *spyOn()*. Diferența dintre cele doua metode este că *spyOn* păstrează implementarea originală a funcției. În acest fel putem duce testul cu un pas mai departe: nu doar că verificăm dacă metoda a fost apelată ci putem face și alte presupuneri, de exemplu să ne asigurăm ca obiectul *state* al componentei a fost modificat corespunzător după execuția funcției. Acest lucru nu ar fi fost posibil cu *jest.fn()* deoarece această metodă oferă suport doar pentru suprascrierea funcției originale. Așadar, pașii de urmat în acest caz sunt:

- Crearea unui “spion” pentru metoda de pe instanța componentei
Acest spion este atașat 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 declanșează acțiunea
Aceași metodă aplicată pentru testarea funcțiilor primite prin props este folosită și aici: elementul este identificat folosind *find()* iar evenimentul este simulat prin *simulate("click")*.
- Verificarea faptului ca răspunsul componentei este cel așteptat
Din nou, se verifică dacă metoda definită la *onClick* a fost apelata. Datorita metodei de testare folosită, funcția originală chiar a fost executată în cadrul testului, așadar putem

face și alte presupuneri, legate de forma state-ului. Pentru a testa schimbările din state o posibilă soluție ar fi fost compararea valorilor din state înainte și după execuția funcției testate, sau chiar și doar a acelor valori care au fost modificate în cadrul testului. Problema cu această soluție este ca programul nu are suficient control asupra noilor valori din state pentru a face această verificare eficientă. Așa 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 și un fișier de snapshot care conține elementele din state cu valorile lor actualizate, dacă este cazul. La următoarele execuții ale testului respectiv se compară noul snapshot cu cel existent și dacă există diferențe atunci vom ști că ceva s-a schimbat în logica aplicației.

```
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 instanța componentei

Fragmentul de cod de mai sus cuprinde etapele prezentate. Funcțiile folosite sunt similare cu cele pentru testarea metodelor trimise prin props, cu mențiunea că “mockMethod” va returna spionul atașat componentei, și 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).toHaveBeenCalled();
  expect(component.find(Component).instance().state).toMatchSnapshot();
});

```

Test generat pentru testarea metodelor din instanța componentei

3. Testarea răspunsului componentei la `onChange`

Evenimentul de tip *onChange* poate să apară în diferite părți ale aplicației, atunci când utilizatorul face o schimbare a valorilor din câmpurile unui formular. Dat fiind că formularele sunt destul de întâlnite în cadrul aplicațiilor, pentru logarea sau crearea unui nou cont, pentru adăugarea unei entități 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 situațiile, reușește să completeze câmpurile unui formular și 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 și extragerea celor relevante. Pentru a decide care elemente aparțin unui formular, în timpul parcurgerii Depth First algoritmul actualizează mereu lista părinților pentru un element. Dacă se întâlnește 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 și părinte un formular și 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, și 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*. Așa cum am prezentat în capitolele anterioare, testele au nevoie de o structură diferită dacă la submit este apelată o funcție trimisă prin props sau din instanța componentei. Pentru optimizarea rezultatului, template-ul pentru încărcarea componentei împreună cu valoarea de test a funcției(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: 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 ca metoda corespunzătoare a fost apelata 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 apelata la *onSubmit* nu este util deoarece acea metodă va fi mereu apelată înaintea unei ultime validări a formularului, însă fără a se executa 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 ca 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 adaugând în test presupunerea făcută: la finalul testului se verifică dacă există în componentă cel puțin un element care să aibă clasa “*error*”.

3.4.7. Analiza coverage a testelor generate

După ce testele generate au fost adăugate în fișier, programul mai are un singur pas de executat: rularea testelor obținute, pentru a stabili rata de acoperire a codului prin analiza coverage. Desigur ca 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, așa că l-am inclus în logica internă a aplicației.

Deoarece comanda de start a programului corespunde execuției directe a unui script din pachet, rularea testelor nu a putut fi făcută printr-o comandă din terminal, așa cum se întâmplă în toate cazurile când programatorul execută testele direct din aplicația React. Pentru a face totuși acest lucru posibil, am abordat o metodă mai puțin întâlnită: am importat librăria Jest în fișierul meu și am apelat direct o funcție internă din cadrul acesteia:

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

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

Am ales să trimit doar opțiunile de configurare necesare pentru a rula strict fișierele returnate în cadrul execuției curente a programului, pentru a pune mai bine în evidență rezultatele obținute.

4. Rezultate obținute

4.1. Rezultate

Pentru a testa utilitatea pachetului în practică am creat o aplicație React care simulează o aplicație de gestionare a unei liste de utilizatori. Pentru aceasta, am adăugat o pagină principală, unde sunt listați sub formă de tabel toți utilizatorii înregistrați. Această pagină conține de asemenea un câmp de căutare și un buton pentru adăugarea unui nou utilizator. La selectarea butonului de adăugare sau a numelui unui utilizator aplicația redirecționează 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.

Aplicația este implementată folosind ultima versiune a librăriei React (16.8.6), folosește librăria *react-router-dom*²² pentru schimbarea rutelor din cadrul aplicației și librăria *semantic-ui-react* ca sursă a componentelor folosite. Așadar, deși aplicația este folosită doar pentru a demonstra utilitatea pachetului, este construită folosind unele dintre cele mai populare librării utilizate în aplicațiile React din producție. Nu am implementat și integrarea cu backend-ul deoarece acest lucru este în afara scopului prezentării: pachetul descris în această lucrare generează unit teste pentru interfața unei aplicații; orice se întâmplă în interacțiunea 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 ușor de controlat.

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

Pagina de listare:

Pagina de listare este compusă din elemente ușor de identificat de aplicația noastră, de exemplu: componente `<Link />` din librăria *semantic-ui-react* pentru lista de utilizatori din tabel și o componentă `<Button />` care are ca și părinte o componentă `<Link />`. În același timp, se testează comportamentul componentei cu prop-uri default pentru lista de utilizatori dar și 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:

²² <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)
```

- 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 afișarea formularului are o implementare puțin mai complicată decât cea responsabilă de listare, prezentată mai sus. Există evenimente de tip *onClick* (pentru butonul care redirecționează utilizatorul la pagina de listare), *onChange* (pentru modificarea valorilor din câmpuri) și *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.
```

- 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 | | | | | |

Observații rezultate în urma generării testelor pentru această componentă:

- cu toate că această componentă este folosită și pentru adăugarea și pentru editarea unui utilizator, în generarea testelor pentru validarea formularului sunt luate în considerare doar valorile default pentru props, așadar verificările se fac, în acest caz, pentru formularul de adăugare. Din această cauză, în unele situații nu sunt acoperite ambele ramuri pentru o expresie *if/else*, ceea ce duce la o scădere a coverage-ului pe coloana “%Branch”. Acesta poate fi totuși considerat un caz special. În funcție de scopul aplicației care folosește react-unit-test-generator, rata de acoperire a formularelor poate să difere.

4.2. Limitări ale aplicației

Există câteva situații în care algoritmul nu poate genera unele teste pentru elementele extrase din componentă. Pentru tratarea acestor situații am luat în calcul și 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 fișierul de teste și să le modifice pe cele incomplete. Așadar 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 execuției. Acest raport este afișat după ce sunt create și rulate restul testelor și are avantajul că nu întrerupe execuția algoritmului pe parcurs și în același timp oferă o informație 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 funcționa corect:

- Elementele care nu au un *data-testid* nu vor putea fi testate

Programul recunoaște elementele din componentă pe baza unui identificator. În acest caz am ales să folosesc *data-testid*-ul. Inițial, 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 apelată la click:

1. metoda este definită în același fișier în care se găsește componenta, și apelată la `onClick`
2. metoda este definită într-un părinte al componentei și trimisă prin props la componenta copil, unde va fi apelată la `onClick`
3. metoda este definită *inline*, la `onClick`.

În ultimul caz, funcția astfel definită va fi una anonimă. Din acest motiv, în cadrul testului nu avem nicio referință la numele funcției pentru a putea adăuga presupunerile ca aceasta a fost apelată.

Tabelul afișat în terminal este creat cu ajutorul unei librării numite *cli-table*²³. Pentru a afișa aceste detalii împreună cu restul informațiilor despre teste oferite de Jest a fost nevoie să creez un *custom reporter*, pe care mai apoi l-am inclus în opțiunile de configurare pentru Jest. Noțiunea de custom reporter este specifică librăriei Jest și reprezintă o clasă care implementează una sau mai multe din următoarele metode: *onStartRun*, *onTestRun*, *onTestResult* și *onRunComplete*, care se vor apela atunci când este declanșat evenimentul pe care îl reprezintă. În cazul nostru am folosit *onRunComplete*, pentru a afișa tabelul la finalul întregii execuții.

Raportul pentru excepțiile 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

²³ <https://www.npmjs.com/package/cli-table>

Concluzii

Consider că am reușit să ating obiectivul propus pentru acest proiect, și anume să construiesc o aplicație pe care să mă simt confortabil să o folosesc în orice proiect. După cum s-a putut observa și în capitolele prezentate anterior, pachetul react-unit-test-generator este capabil să returneze suite de teste care acoperă minim 50% din logica de implementare, facilitează procesul de *debugging* în cazul în care este întâlnită o eroare internă, prin optimizarea mesajelor de eroare așa încât programatorul să identifice cât mai ușor cauza pentru care aplicația și-a oprit execuția, este ușor de configurat (poate funcționa doar cu opțiunile de configurare implicite, dacă structura proiectului este cea clasică) și este orientat spre testarea evenimentelor declanșate la interacțiunea cu utilizatorul. Toate acestea fac din acest pachet un pachet promițător, cu atât mai mult cu cât este accesibil și pe registrul npm. Deoarece încă nu există o librărie completă care să genereze automat unit testele necesare unei aplicații React (sau chiar Javascript) și dat fiind că este nevoie uneori ca unit testele să fie foarte specifice, aplicația prezentată aduce o îmbunătățire considerabilă a acestui proces și sunt de părere că ar putea reprezenta o sursă de inspirație pentru abordările viitoare ale acestui domeniu.

Posibile îmbunătățiri și planuri de viitor

Aplicația prezentată nu este încă suficient de dezvoltată pentru a acoperi prin unit teste întreaga implementare a unei aplicații de producție. După cum a fost menționat încă de la începutul lucrării, varianta actuală a proiectului asigură testarea celor mai comune funcționalități. Pe termen scurt, se poate adăuga suport și pentru testarea altor elemente sau interacțiuni ale aplicației cu utilizatorul, de exemplu testarea câmpurilor de tip *dropdown* sau *multiselect*. Pe termen lung, logica aplicației poate fi extinsă pentru a verifica și schimbările vizuale declanșate de evenimentele de tip *onClick* și *onChange*. Desigur, este vorba doar de o chestiune de timp și priorități până când îmbunătățirile menționate mai sus vor fi implementate. Deoarece pachetul este publicat pe npm, există și posibilitatea ca acesta să ajungă în atenția altor persoane interesate de acest domeniu și care să dorească să contribuie prin adăugarea de noi funcționalități.

Bibliografie

Articole de specialitate

- [1] TDD--The Art of Fearless Programming
<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/>

Documentații 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>