Lungu Laura-Vanesa - Grupa 341
Negoiță-Crețu Raluca-Marina - Grupa 341

Testarea Sistemelor Software - Testarea unei aplicații mobile

1. Introducere în Testarea Sistemelor Software

Testarea sistemelor software este procesul de verificare și validare a funcționării corecte a aplicațiilor, înainte de lansare. Scopul este de a identifica eventuale defecte și de a asigura calitatea finală a produsului. În cadrul testării software, putem clasifica tipurile de testare astfel:

a. După accesul la cod:

- **White-Box Testing**: Testarea logicii interne a aplicației, bazată pe cunoașterea codului sursă (ex: metrici de acoperire, teste pe ramuri și căi de execuție).
- **Black-Box Testing**: Testarea funcționalității aplicației fără a cunoaște implementarea internă (ex: testarea interfețelor, a serviciilor API).
- **Grey-Box Testing**: Combină ambele abordări — testerul cunoaște parțial structura internă.

b. După tipul de testare:

- **Testare Funcțională**: Verifică dacă sistemul face ceea ce ar trebui (ex: login, înregistrare, cumpărare produs).
- **Testare Non-Funcțională**: Evaluează caracteristici precum performanța, securitatea sau compatibilitatea aplicației.

Exemple de tipuri de testare funcțională:

- **Unit Testing**: Testează cele mai mici părți independente ale codului, cum ar fi funcții sau componente.
- **Integration Testing**: Verifică dacă mai multe module ale aplicației lucrează împreună corect.

- **System Testing**: Testează întregul sistem ca un întreg, inclusiv toate modulele integrate.
- **End-to-End Testing (E2E)**: Simulează comportamentul unui utilizator real parcurgând întregul flow de aplicație, de la început până la sfârșit.

Exemple de testare non-funcțională:

- **Performance Testing**: Verifică rapiditatea și stabilitatea sistemului sub diverse condiții de sarcină (load testing, stress testing).
- **Usability Testing**: Măsoară cât de ușor și intuitiv este pentru utilizatori să folosească aplicația.
- **Compatibility Testing**: Testează funcționarea aplicației pe diverse dispozitive, sisteme de operare și browsere. [1]

2. Descrierea arhitecturii aplicației testate

MovieMingle este o aplicație mobilă care recomandă filme utilizatorilor pe baza preferințelor lor. Aplicația îmbunătățește experiența de vizionare prin predicții realizate cu ajutorul tehnicilor de învățare automată. Cu o interfață prietenoasă și un design modern și atrăgător, MovieMingle devine partenerul ideal pentru pasionații de film care caută titluri noi, potrivite gusturilor lor. Indiferent dacă preferi drame, comedii sau thrillere, aplicația va oferi recomandări care se potrivesc așteptărilor tale.

Adaptare backend din proiect web existent

Pentru a eficientiza procesul de dezvoltare și a reutiliza o bază solidă de cod, backend-ul aplicației MovieMingle a fost preluat dintr-un proiect de-al nostru anterior cu versiune web. Acesta este construit în **Java Spring Boot**, conține deja **logica de recomandare și endpoint-urile necesare**, și a fost adaptat pentru a răspunde nevoilor unei aplicații mobile. Link-ul către repo-ul aplicației web se poate accesa aici.

Aplicația **Movie Mingle Mobile** are o arhitectură **client-server**:

- **Frontend (Mobile App):** Realizat cu **React Native**, responsabil pentru experiența utilizatorului (UI/UX), interacțiunea cu utilizatorul, trimiterea de cereri HTTP către backend.
- **Backend:** Dezvoltat cu **Spring Boot** (Java), gestionează logica de business, autentificarea utilizatorilor, recomandările de filme și interacțiunea cu baza de date **MySQL**.
- **Securitate:** Implementată cu **JWT (JSON Web Tokens)** pentru autentificare sigură și protecție CSRF.
- **Integrare externă:** Folosește **TMDB API** pentru a furniza date actualizate despre filme, inclusiv descrieri, actori și ratinguri.

Configuratie hardware

- **CPU**: Intel Core Ultra 9 185H, Intel Core i7-9750H
- **GPU**: NVIDIA GeForce RTX 4060, NVIDIA GeForce GTX 1650
- **RAM**: 32 GB, 8 GB
- **Stocare**: SSD 512GB, SSD 477 GB capacitate totală configurat RAID

Configuratie software

- **SO**: Microsoft Windows 11 Pro, Microsoft Windows 11 Pro
- **IDE**: JetBrains IntelliJ IDEA 2024.3.1.1, Visual Studio Code
- **Runtime Java:** OpenJDK 21 (pentru Spring Boot)

Mediul de lucru utilizat nu implică rularea pe o mașină virtuală.

3. Framework-uri de testare

Testarea în proiectul nostru s-a concentrat pe compararea framework-urilor specifice celor două componente arhitecturale ale aplicației:

Frontend

Framework	Scop principal	Tehnologie
Jest + React Native Testing Library	Testare unitară și de integrare pentru componente și funcționalitate de bază (frontend)	JavaScript
Maestro	Testare end-to-end a flow-urilor reale de utilizator (frontend)	YAML scripts + Android Emulator

Backend

Framework	Scop principal	Tehnologie
JUnit 5 + Mockito	Testare unitară a serviciilor backend prin simularea dependențelor externe	Java + Spring Boot
JUnit5 (SpringBootTest)	Testare de integrare între componente reale (servicii, repo-uri, DB in-memory)	Java + Spring Boot + H2 Database
TestNG	Testare funcțională (unitară și de integrare) și organizare avansată a testelor cu scenarii reale	Java + Spring Boot + TestNG

JaCoCo	Măsurarea gradului de acoperire a codului de către testele unitare/integration	Java + Gradle / Maven

Această abordare multi-framework a permis acoperirea completă a testării, de la validarea componentelor izolate până la verificarea fluxurilor reale de utilizator, în ambele capete ale aplicației.

4. Versiuni ale tool-urilor utilizate in testare

Java - 21.0.5
Maven - 3.9.9
Spring Boot - 3.4.0
Mockito - 5.8.0
TestNG - 7.9.0
JaCoCo - 0.8.10
React Native - 0.76.9
Jest - ^29.7.0
React Native Testing Library - ^13.2.0
Expo - ~52.0.43
Maestro - 1.40.0
Node.js - 22.13.1
npm - 11.3

5. Testare Frontend folosind Jest și React Native Testing Library

5.1. Ce este Jest?

Jest [2] este un framework de testare dezvoltat de Facebook, extrem de popular în ecosistemul JavaScript, folosit în special pentru aplicații React și React Native. El permite rularea de **teste unitare, teste de integrare și teste de componente** într-un mod rapid, predictibil și ușor de configurat. Caracteristici principale:

- Rulează testele foarte rapid, local, fără a fi nevoie de emulator sau device real.
- Oferă suport pentru *snapshot testing* (compararea automată a interfeței).
- Include automat un motor de mock-uri pentru funcții externe (ex: API-uri, module).
- Integrare excelentă cu biblioteci suplimentare precum React Native Testing Library.

5.2. Ce este React Native Testing Library (RNTL)?

React Native Testing Library (RNTL) [3] extinde Jest, oferind un set de utilitare specializate pentru a testa componentele React Native într-un mod asemănător cu modul în care utilizatorii le-ar folosi. Scopul său principal este să simuleze interacțiunile reale cu aplicația, fără să depindă de detalii interne de implementare. Ce aduce în plus:

- Simulare de acțiuni reale: fireEvent.press, fireEvent.changeText, etc.
- Căutare de elemente după text, rol, accesibilitate (getByText, getByTestId).
- Focus pe *behavior*, nu pe *implementare*.
- Îmbunătățește claritatea și robustețea testelor.

5.3. De ce am ales Jest + RNTL pentru testare?

- Ideal pentru testarea rapidă a logicii aplicației și a interacțiunilor simple.
- Ne permite să scriem teste unitare și de integrare ușor, cu feedback aproape instant.
- Integrarea nativă în React Native cu Expo ceea ce scade timpul de configurare și complexitatea.
- Ne ajută să detectăm erori timpurii fără să rulăm emulatorul sau dispozitive reale.
- Avem acces complet la cod și putem simula (mocks) orice dependență (API-uri externe, context de autentificare, etc.).

5.4. Configurarea Jest și RNTL cu Expo

Aplicația MovieMingle a fost creată folosind Expo [4] pentru a simplifica procesul de dezvoltare: oferă hot-reload rapid, integrare ușoară cu funcționalități native și reduce semnificativ timpul de setup. Pentru testare folosim Jest cu presetul *jest-expo* [5]. Setup-ul de testare include și un fișier suplimentar (*jestSetup.js*) unde sunt definite mock-uri pentru diverse librării native. **Mock**-urile [6] sunt versiuni controlate ale unor funcționalități externe sau complexe (ex: gesturi, animații, stocare locală). În testare, ele ne ajută să:

- Izolăm componentele de dependențele externe.
- Evităm erorile din librării care nu sunt compatibile cu mediul de test.
- Testăm doar logica aplicației noastre, nu comportamentul platformei native.

5.5. Descrierea testelor Jest + React Native Testing Library (RNTL)

În cadrul proiectului am realizat 34 de teste de mai multe tipuri, după cum urmează:

5.5.1. Teste Unitare

Nr	Test	Ce testează
1.	Buton Login	Testează dacă butonul de login apare pe ecran cu textul corect
2.	Formular Login	Verifică dacă Dacă câmpurile de e-mail și parolă acceptă input de la utilizator și dacă valorile introduse sunt stocate corect în componente
3.	Validare câmpuri goale	Verifică validarea câmpurilor goale și afișarea mesajelor de eroare.
4.	Formular Register	Verifică că input-ul din formular își schimbă corect valoarea când utilizatorul tastează.

5.5.2. Teste Funcționale

Nr.	Test	Ce testează
1.	Afișare Loading Spinner la încărcarea unui film	Verifică dacă apare spinner-ul de încărcare cât timp datele filmului sunt încărcate.
2.	Tratare erori API TMDB	Verifică dacă aplicația scapă corect de spinner și nu crapă dacă API-ul întoarce o eroare.
3.	Afișare Mesaje Empty State	Testează afișarea mesajelor pentru liste goale de filme (watched, watchlist, favorite, rated).
4.	Afișarea loading spinner la încărcarea unor liste de filme	Testează afișarea spinner-ului de încărcare în timpul fetch-ului de filme.
5.	Afișare informații utilizator în Profile Screen	Verifică dacă datele personale ale utilizatorului sunt afișate corect după încărcare.
6.	Afișare grafice dashboard	Verifică dacă secțiunile de grafice (stats, activitate lunară, genuri) se afișează după încărcarea datelor.
7.	Deschidere modal de schimbare avatar în Profile Screen	Verifică dacă la apăsarea butonului "Change Avatar" se deschide modalul de alegere avatar.
8.	Randare câmpuri formular Register	Testează dacă Registration Screen afișează corect toate câmpurile de introducere (First Name, Last Name, Email, etc.).
9.	Afișare erori inputuri invalide	Testează dacă mesajele de eroare sunt afișate pentru inputuri greșite sau incomplete.
10.	Tratare erori server	Testează tratarea erorilor neașteptate de la server și afișarea mesajului general de eroare.
11.	Randare titlu, subtitlu și search bar	Testează dacă se afișează corect titlul aplicației, subtitlul și bara de căutare pe ecranul de căutare.
12.	Afișare mesaj empty state la căutare fără rezultate	Testează dacă se afișează mesajul „No movies found” atunci când căutarea nu returnează niciun film.

5.5.3. Teste de integrare

Nr.	Test	Ce testează
1.	Eroare Autentificare (Credențiale Invalide)	Testează flow-ul complet de autentificare și tratarea erorilor primite de la API în interfață.

2.	Autentificare cu Succes	Simulează login reușit și verifică apelarea funcției authenticate din contextul de autentificare.
3.	Navigare la Register Screen	Simulează apăsarea pe linkul "Register!" și verifică navigarea către ecranul de înregistrare.
4.	Afișare carusele cu liste de filme	Verifică dacă apar caruselele pentru filmele existente în lista userului și returnate de API.
5.	Navigare către Movie Details Screen	Testează dacă la apăsarea pe un film, navigarea către MovieDetail Screen se face corect, trimițând ID-ul filmului.
6.	Căutare filme după tastare și submit	Testează dacă textul introdus în search bar declanșează fetch-ul corect al filmelor și afișează rezultatele.
7.	Filtrare filme după gen	Testează dacă selectarea unui gen apelează corect API-ul și actualizează lista de filme afișate.

5.5.4. Teste de Performanță

Nr.	**Test**	**Ce testează**
1.	Performanță randare inițială HomeScreen	Verifică dacă randarea inițială a UI-ului HomeScreen durează sub 200ms.
2.	Performanță încărcare conținut HomeScreen	Verifică dacă toate datele (secțiunile) se încarcă complet sub 1000ms.
3.	Performanță procesare dataset mare	Verifică dacă aplicația încarcă un set mare de date (500 filme) în sub 1500ms.

5.5.5. Test de Snapshot

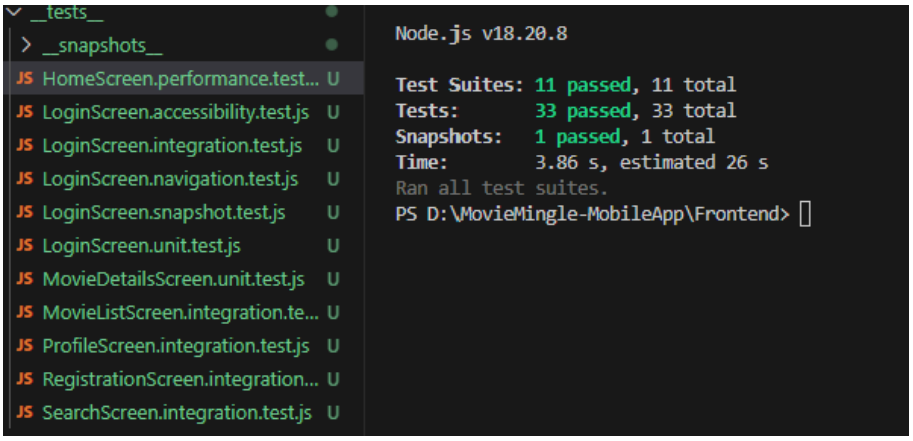
Nr.	**Test**	**Ce testează**
1.	Snapshot Login Screen	Capturează structura vizuală a ecranului de login și verifică dacă aceasta nu s-a modificat neintenționat.

5.5.6. Teste de Accesibilitate

Nr.	**Test**	**Ce testează**
1.	Accesibilitate Input-uri Formulare	Verifică existența labelurilor de accesibilitate pentru câmpurile Email și Parolă.

2.	Accesibilitate butoane	Verifică dacă butonul de Login și link-ul „Forgot Password?" sunt etichetate corect.
3.	Accesibilitate Link Register	Verifică dacă textul "Have an account?" este vizibil și accesibil.

5.6. **Code snippets:** Toate cele 34 de teste au trecut.



5.6.1. Test Unitar – Introducere email și parolă în LoginScreen

```
it("should allow user to enter email and password", async () => {
  render(
    <AuthContext.Provider value={mockAuthContext}>
      <NavigationContainer>
        <LoginScreen />
      </NavigationContainer>
    </AuthContext.Provider>
  );

  const emailInput = screen.getByTestId("emailInput");
  const passwordInput = screen.getByTestId("passwordInput");

  fireEvent.changeText(emailInput, "test@email.com");
  fireEvent.changeText(passwordInput, "password123");

  expect(emailInput.props.value).toBe("test@email.com");
  expect(passwordInput.props.value).toBe("password123");
});
```

Acest test verifică dacă input-urile pentru email și parolă din ecranul de autentificare (Login Screen) preiau și stochează corect datele introduse de utilizator.
Ce validează concret:

- Câmpul de email se actualizează cu textul introdus.
- Câmpul de parolă se actualizează cu textul introdus.

5.6.2. Teste Funcționale - Afișare informații utilizator

```
it("should render loading indicator initially", async () => {
  const { findByTestId } = render(<ProfileScreen />);
  expect(await findByTestId("loading-indicator")).toBeTruthy();

  await waitFor(() => {
    expect(fetchDashboardData).toHaveBeenCalled();
  });
});

it("should render user info after loading", async () => {
  const { findByText } = render(<ProfileScreen />);

  expect(await findByText("First Name:")).toBeTruthy();
  expect(await findByText("John")).toBeTruthy();
  expect(await findByText("Last Name:")).toBeTruthy();
  expect(await findByText("Doe")).toBeTruthy();
  expect(await findByText("Email:")).toBeTruthy();
  expect(await findByText("john@example.com")).toBeTruthy();
});

it("should render dashboard charts", async () => {
  const { findByText } = render(<ProfileScreen />);

  expect(await findByText("Dashboard 📊")).toBeTruthy();
  expect(await findByText("User Stats 📈")).toBeTruthy();
  expect(await findByText("Monthly Activity 🗓️")).toBeTruthy();
  expect(await findByText("Genre Distribution 🎭")).toBeTruthy();
});
```

Testul 1 verifică dacă, atunci când ecranul *ProfileScreen* se încarcă, apare un indicator de încărcare (loading-indicator) și dacă apelul API *fetchDashboardData* este declanșat corect.
Testul 2 validează că, după finalizarea încărcării datelor, informațiile utilizatorului (prenume, nume, email) sunt afișate corect pe ecranul de profil (*ProfileScreen*).
Testul 3 verifică dacă, după încărcarea datelor de dashboard, sunt afișate corect toate secțiunile grafice: Dashboard, User Stats, Monthly Activity și Genre Distribution.

5.6.3. Test de Integrare – Căutare filme în Search Screen

```
it("searches movies when typing in search bar and pressing submit", async () => {
  fetchMoviesBySearch.mockResolvedValueOnce({
    data: [
      {
        id: 1,
        title: "Test Movie",
        poster_path: "/test.jpg",
        vote_average: 7.5,
      },
    ],
  });

  const { getByPlaceholderText, findByText } = render(<SearchScreen />);
  const searchInput = getByPlaceholderText("Search for movies...");

  fireEvent.changeText(searchInput, "Avengers");
  fireEvent(searchInput, "submitEditing");

  await waitFor(() => {
    expect(fetchMoviesBySearch).toHaveBeenCalledWith("Avengers", 1);
    expect(findByText("Test Movie")).toBeTruthy();
  });
});
```

Testul simulează introducerea unui text (“Avengers”) în bara de căutare de pe *SearchScreen*, apasă submit și verifică dacă API-ul *fetchMoviesBySearch* este apelat corect și dacă rezultatul căutării este afișat pe ecran.

5.6.4. Test de Performanță - Încărcare Dataset mare

```
// Test 3: Large dataset performance
it("handles large dataset under 1500ms", async () => {
  // Override with large dataset
  require("../api/movieApi").fetchPopularMovies.mockImplementation(() =>
    Promise.resolve(
      Array(500)
        .fill()
        .map((_, i) => ({ id: i, title: `Movie ${i}` }))
    )
  );

  const start = performance.now();
  render(<MockHomeScreen />);

  await waitFor(
    () => {
      expect(screen.queryByText("Loading movies...")).toBeNull();
    },
    { timeout: 3000 }
  );

  const duration = performance.now() - start;
  console.log(`[PERF] Large dataset load: ${duration}ms`);
  expect(duration).toBeLessThan(1500);
});
```

Testul măsoară timpul necesar pentru încărcarea unui set mare de date (500 de filme) pe HomeScreen și validează că procesarea se finalizează sub 1500 ms.

5.6.5. Teste de Accesibilitate

```
describe("LoginScreen Accessibility", () => {
  beforeEach(() => {
    render(
      <NavigationContainer>
        <LoginScreen />
      </NavigationContainer>
    );
  });

  test("has accessible form inputs", () => {
    expect(screen.getByLabelText("Email input")).toBeTruthy();
    expect(screen.getByLabelText("Password input")).toBeTruthy();
  });

  test("has accessible buttons", () => {
    expect(screen.getByLabelText("Login button")).toBeTruthy();
    expect(screen.getByText("Forgot Password?")).toBeTruthy();
  });

  test("has accessible registration link", () => {
    expect(screen.getByText(/have an account/i)).toBeTruthy();
  });
});
```

Testul 1 verifică dacă input-urile de email și parolă din *LoginScreen* sunt corect etichetate pentru screen readers.
Testul 2 verifică dacă butonul de login și link-ul „Forgot Password?” sunt accesibile și pot fi corect identificate de către utilizatori.
Testul 3 validează că link-ul pentru crearea unui cont nou ("Have an account?") este vizibil și accesibil utilizatorilor.

5.6.6. Snapshot Test

```
test("renders login screen correctly", () => {
  const tree = render(
    <NavigationContainer>
      <LoginScreen />
    </NavigationContainer>
  ).toJSON();

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

6. Testare Frontend folosind Maestro

6.1. Ce este Maestro?

Maestro [7] este un framework de **testare end-to-end** [8] destinat aplicațiilor mobile, bazat pe un limbaj declarativ scris în **YAML**. Este conceput pentru a simplifica procesul de testare pentru aplicațiile mobile dezvoltate pe platformele iOS și Android. Maestro este optimizat în special pentru aplicațiile realizate cu **Expo** și **Flutter**. Comparativ cu alte framework-uri precum ***Detox*** [9], care necesită cunoștințe avansate de JavaScript și configurații mai complexe, Maestro se remarcă prin ușurința de utilizare și configurare, permițând dezvoltatorilor să scrie rapid teste de calitate.

6.2. Utilizarea Maestro împreună cu Expo

Expo este o platformă de dezvoltare pentru aplicații React Native. Maestro poate automatiza testele aplicațiilor Expo prin:

- Rularea aplicației cu Expo Go pe un dispozitiv mobil sau emulator.
- Automatizarea interacțiunilor folosind scripturi YAML definite în Maestro.

Integrarea Maestro cu aplicații Expo:

- Transformarea aplicației Expo într-o aplicație nativă prin comanda *npx expo prebuild*.
- Rularea aplicației native folosind *npx expo run:android* sau *npx expo run:ios*.
- Configurarea adresei IP corecte în fișierul config.yaml.
- Executarea testelor folosind comenzile Maestro, interacționând direct cu aplicația construită. [10]

6.3. Instalarea Maestro

Pentru instalarea Maestro sunt necesari următorii pași:

- Instalarea Node.js și npm.
- Instalarea Maestro folosind comanda: *npm install -g maestro*
- Verificarea instalării cu comanda: *maestro –version*
- Alternativ, zip-ul cu instalarea Maestro poate fi descărcat de pe site-ul oficial.

6.4. Configurarea Mediului pentru rularea testelor Maestro (Android)

Pașii necesari pentru a putea rula testele:

1. Pornirea backend-ului (Spring Boot) folosind *mvn spring-boot:run.*
2. Rularea aplicației frontend folosind *npx expo run:android.*
3. Rularea testelor Maestro din directorul Frontend/maestro folosind comanda: *maestro test maestro/test-login.yaml*

Notă: Pentru rularea pe un telefon real, este necesară configurarea suplimentară a dispozitivului.

6.5. Configurarea telefonului pentru testare (Android)

- Activarea opțiunilor de dezvoltator: Accesare „Setări” → „Despre telefon” → apăsare de 10 ori pe „Număr versiune”.
- Activarea USB debugging: „Setări” → „Opțiuni dezvoltator” → activare „USB debugging”.
- Conectarea telefonului la PC și acceptarea permisiunilor.

6.6. Testele Maestro realizate

Au fost realizate **4 teste end-to-end** pentru aplicație:

1. Test de autentificare (test-login.yaml)

În acest fișier este testată interacțiunea utilizatorului cu Screen-ul care apare la launch-ul aplicației și are loc acțiunea de log-in cu credențialele unui cont deja existent pentru a se testa faptul că logarea se desfășoară conform așteptărilor.

```
appId: com.ncraluca.Frontend
---
- launchApp
- tapOn: "Log In"
- tapOn: "Enter your email"
- inputText: "ralucanegoita13@gmail.com"
- tapOn: "Enter your password"
- inputText: "1213.Ralu"
- tapOn: "Log in"
- tapOn: "Log in"
- assertVisible: "You're now logged in!"
- tapOn: "OK"
```

2. Test Profil Utilizator (test-profile.yaml)

În acest fișier este testată interacțiunea utilizatorului dintre un Screen oarecare din aplicație și Screen-ul de profile astfel:

- este accesat Screen-ul de profile
- utilizatorul apasă pe "Change Avatar" pentru a selecta un alt avatar
- deoarece Maestro funcționează majoritar pe text și nu interacționează bine cu pozele, am ales ca interacțiunea să se finalizeze prin apăsarea butonului "Cancel"

```
appId: com.ncraluca.Frontend
---
- tapOn: "Your Profile"
```

```
- assertVisible: "Your Profile"
- tapOn: "Change Avatar"
- assertVisible: "Choose Your Avatar"
- tapOn: "Cancel"
```

3. Test Interacțiune HomeScreen (test-home.yaml)

În acest fișier este testată interacțiunea utilizatorului dintre un Screen oarecare din aplicație și Screen-ul de home astfel:

- se accesează screen-ul de Home
- se apasă pe butonul de "See More" pentru primul film care apare pe ecran
- se așteaptă încărcarea detaliilor filmului
- se adaugă la favorite filmul
- se adaugă la watched filmul
- se scoate filmul de la favorite
- se scoate filmul din watched

```
appId: com.ncraluca.Frontend
---
- tapOn: "Home"
- assertVisible: "Home"
- tapOn: "▸ See More"
- waitForAnimationToEnd
- assertVisible: "Add to Favorite"
- tapOn: "Add to Favorite"
- assertVisible: "Mark as Watched"
- tapOn: "Mark as Watched"
- assertVisible: "Added to Favorites"
- tapOn: "Added to Favorites"
- assertVisible: "Watched"
- tapOn: "Watched"
```

4. Test de căutare filme (test-search.yaml)

În acest fișier este testată interacțiunea utilizatorului dintre un Screen oarecare din aplicație și Screen-ul de search astfel:

- se accesează screen-ul de Search
- se caută un film după keyword-ul "Bullet"
- se caută filme după genul "Action"
- se caută filme și după genul "Animation"
- se deselectează "Action"
- se deselectează "Animation"

```
appId: com.ncraluca.Frontend
```

```
---
- tapOn: "Search"
- assertVisible: "🎬 Movie Explorer"
- tapOn: "Search for movies..."
- inputText: "Bullet"
- tapOn: "Action"
- tapOn: "Action"
- assertVisible: "Action"
- tapOn: "Animation"
- assertVisible: "Animation"
- tapOn: "Action"
- tapOn: "Animation"
```

6.7. Comparație între Maestro și Detox

Testarea aplicațiilor mobile a devenit o componentă crucială în dezvoltarea de software, iar *pentru dezvoltatorii de aplicații React Native,* ***Detox*** *și* ***Maestro*** *sunt două dintre cele mai populare framework-uri de* ***testare end-to-end.*** Ambele oferă soluții eficiente pentru testarea aplicațiilor pe dispozitive fizice și simulatoare, dar fiecare are caracteristicile și avantajele sale. În această comparație detaliată, vom explora fiecare framework în parte, subliniind punctele forte ale fiecăruia, diferențele între ele și la ce tipuri de proiecte se potrivesc cel mai bine.

6.7.1. Ce este Detox?

Detox [9] este un framework de testare end-to-end pentru aplicațiile React Native. Acesta a fost dezvoltat de Wix și este unul dintre cele mai populare framework-uri de testare pentru React Native, fiind adesea utilizat pentru a testa aplicații în mod real, pe dispozitive sau simulatoare iOS și Android. Detox folosește **JavaScript** pentru a scrie teste și oferă un set de API-uri care permit dezvoltatorilor să controleze aplicațiile, să simuleze interacțiuni și să verifice comportamentele aplicațiilor. Detox se integrează bine cu Jest, oferind o soluție robustă pentru testarea automată a aplicațiilor.

6.7.2. Avantaje Detox

- **Suport complet pentru React Native:** Detox este construit special pentru aplicațiile React Native, oferind un control detaliat asupra comportamentului aplicațiilor, cu acces la API-urile native.
- **Testare pe dispozitive reale:** Detox permite testarea pe dispozitive reale, ceea ce este esențial pentru verificarea comportamentului aplicației în condiții reale.
- **Flexibilitate și control:** Detox oferă un control detaliat asupra aplicațiilor, incluzând suport pentru gesturi, sincronizare cu aplicațiile și verificarea unor stări foarte specifice ale aplicației.
- **Comunitate activă:** Detox are o comunitate mare și activă, iar documentația este foarte bine structurată, ceea ce face mai ușor să înveți și să folosești framework-ul.

6.7.3. Dezavantaje Detox

- **Configurare complexă:** Configurarea Detox pe Android și iOS poate fi o provocare, mai ales pentru dezvoltatorii care nu sunt familiarizați cu configurările avansate ale mediului de dezvoltare.
- **Limbaj JavaScript:** Detox folosește JavaScript, ceea ce poate să nu fie ideal pentru toți dezvoltatorii. De asemenea, pentru dezvoltatorii care preferă limbaje mai declarative, această abordare poate părea un pic mai complicată.

6.7.4. Avantaje Maestro

- **Configurare simplă și rapidă:** Maestro are o *configurare mult mai simplă* decât Detox. Testele sunt scrise în YAML, iar configurarea este minimă, făcându-l o alegere excelentă pentru echipele care doresc să înceapă rapid cu testele end-to-end.
- **Suport pentru Expo și Flutter:** Maestro este ideal pentru aplicațiile care folosesc Expo sau Flutter. Dacă dezvoltatorul folosește Expo Go pentru a construi aplicațiile sale, Maestro poate fi integrat rapid pentru a efectua testele end-to-end, fără necesitatea unei configurații avansate.
- **Ușor de învățat:** Deoarece folosirea YAML este mult mai simplă și mai declarativă decât programarea în JavaScript, dezvoltatorii care nu sunt familiarizați cu limbaje de programare complexe pot învăța rapid cum să scrie teste cu Maestro.
- **Suport cross-platform:** Maestro poate fi folosit pe Android și iOS, făcându-l o alegere bună pentru aplicațiile care trebuie să ruleze pe ambele platforme.

6.7.5. Dezavantaje Maestro

- **Mai puțin flexibil:** În comparație cu Detox, Maestro oferă mai puține opțiuni de personalizare și mai puțin control asupra detaliilor aplicației. De exemplu, dezvoltatorii nu pot simula la fel de multe interacțiuni avansate sau comportamente native ale aplicației.
- **Nou pe piață:** Deși este în creștere, Maestro nu are încă o comunitate la fel de mare ca Detox, iar documentația nu este la fel de detaliată în comparație cu Detox.

6.7.6. Comparație detaliată între Detox și Maestro

Caracteristică	Maestro	Detox
Limbaj de programare	YAML (declarativ)	JavaScript (Jest)
Configurare	Simplă, rapidă (ideal pentru Expo)	Complexă, necesită configurări avansate
Suport Expo	Complet	Limitat
Suport Flutter	Da	Nu

Flexibilitate	Limitată	Mare (control detaliat asupra aplicației)
Ușurință de învățare	Foarte ridicată	Medie
Suport CI/CD	Da	Da
Comunitate	În creștere	Mare și activă

7. Studiu comparativ între Jest + RNTL și Maestro

Caracteristică	Jest + React Native Testing Library (RNTL)	Maestro
Tip testare	Testare unitară și de integrare (Unit/Integration Testing)	Testare end-to-end (E2E)
Nivel de testare	Componentă, funcționalitate internă	Flow-uri reale de utilizator
Limbaj	JavaScript	YAML (declarativ)
Necesită emulator/dispozitiv?	Nu obligatoriu (teste locale rapide)	Da (emulator sau dispozitiv fizic)
Configurare	Integrare ușoară cu Expo și Jest	Necesită prebuild aplicație și setup emulator
Simulare interacțiuni	Simulează acțiuni asupra componentelor (ex: fireEvent.press)	Simulează interacțiuni reale (tap, inputText)
Control asupra codului	Complet (mocks, spies, jest.fn(), etc.)	Limitat la interfața vizuală
Viteză de execuție	Foarte rapidă (teste locale, fără device)	Mai lentă (teste E2E emulator)
Complexitate cod test	Necesită cunoștințe de JavaScript	Foarte simplu, declarativ, doar YAML
Scalabilitate în proiect mare	Mare (teste fine, unitare și precise)	Bună pentru testarea fluxurilor principale

Detectare erori	Detectează rapid bug-uri de implementare logică	Detectează probleme de integrare și UX real
Utilizare CI/CD	Integrabil cu Jest în pipelines CI/CD	Integrabil ușor în CI/CD cu scripturi simple
Cazuri de utilizare ideale	Testarea logicii interne, componentelor UI/UX	Validarea fluxurilor complete de utilizator

8. Testare Backend folosind JUnit + Mockito

8.1. Ce este JUnit?

JUnit 5 [11] este una dintre cele mai utilizate biblioteci de testare unitare în Java. Este standardul pentru scrierea de teste automate, oferind un cadru simplu și expresiv pentru validarea logicii aplicației. Oferă adnotări moderne (@Test, @BeforeEach, @AfterEach), suport pentru testare parametrizată și integrare facilă cu IDE-uri și tool-uri de build (ex: Maven, Gradle).

8.2. Ce este Mockito?

Mockito [12] este o bibliotecă de *mocking* pentru Java, folosită pentru a simula comportamentul obiectelor dependente (ex: repository-uri, servicii externe). Permite izolarea unității testate, oferind control complet asupra valorilor returnate și verificarea apelurilor realizate asupra obiectelor simulate.

8.3. De ce sunt folosite împreună?

JUnit 5 și Mockito se completează perfect:

- JUnit oferă structura testului (setup, execuție, aserții).
- Mockito permite mock-uirea dependențelor, astfel încât testul să fie focusat exclusiv pe componenta testată.

Această combinație:

- reduce semnificativ complexitatea testelor;
- evită nevoia de baze de date reale sau conexiuni externe;
- accelerează timpul de rulare al testelor;
- crește precizia testelor unitare prin izolare totală a logicii. [13]

8.4. Descrierea testelor JUnit + Mockito

În cadrul proiectului am realizat peste 40 de teste de mai multe tipuri după cum urmează:

8.4.1. Teste Unitare

Nr.	Test	Ce testează
1.	Test unitar validare înregistrare	Verifică scenariul în care utilizatorul este deja înregistrat (se aruncă UserAlreadyExistsException)
2.	Test unitar creare cont nou	Simulează salvarea unui nou utilizator și trimiterea unui email de confirmare
3.	Test unitar codificare parolă	Verifică dacă parola utilizatorului este codificată corect
4.	Test unitar existență utilizator	Verifică dacă un utilizator există sau nu în baza de date
5.	Test unitar verificare film în favorite	Verifică dacă un film se află deja în lista de favorite
6.	Test unitar obținere lista filme favorite	Verifică dacă lista de filme favorite a utilizatorului este returnată corect
7.	Test unitar de creare a unui secure token	Generează un token și setează data de expirare
8.	Test unitar de salvare a unui secure token	Salvează un token existent în repository
9.	Test unitar de căutare a unui token	Caută un token după valoarea sa
10.	Test unitar de căutare a unui token care nu există în baza de date	Returnează null dacă token-ul nu este găsit
11.	Test unitar de stergere a unui token	Șterge un token existent
12.	Test unitar de verificare a unui token	Returnează timpul de valabilitate al tokenului
13.	Test unitar de calculare al rating-ului	Calculează media ratingurilor existente
14.	Test unitar de afișare a unui rating inexistent	Returnează null dacă ratingul nu există
15.	Test unitar de afișare a unui rating existent	Returnează ratingul dat de utilizator
16.	Test unitar care afișează lista goală de filme la care un user a oferit rating	Returnează listă goală dacă nu există ratinguri
17.	Test unitar care afișează lista de filme la care un user a oferit rating	Returnează lista de filme notate
18.	Test unitar de creare a matricei de rating pentru recomandare	Testează getterele și accesul la ratinguri în matrice

8.4.2. Teste de Integrare

Nr.	Test	Ce testează
1.	testBuildMatrix	Construiește matricea de ratinguri pornind de la useri și filme
2.	testTrainModel	Antrenează modelul SVD folosind matricea
3.	testIsMovieWatched	Verifică dacă un film adăugat în lista de vizionate este marcat corect ca fiind vizionat.
4.	testAddMovieToWatched	Testează adăugarea cu succes a unui film în lista de vizionate.
5.	testAddMovieToWatched_AlreadyWatched	Verifică dacă aplicația gestionează corect cazurile în care filmul este deja marcat ca vizionat.
6.	testLoadUserByUsername_UserExists	Verifică încărcarea corectă a detaliilor unui utilizator existent.
7.	testLoadUserByUsername_UserDoesNotExist	Verifică aruncarea excepției pentru un utilizator inexistent.
8.	testGetUserDashboardStats	Verifică generarea statisticilor dashboard-ului pentru un utilizator cu filme vizionate și evaluate.
9.	testForgottenPassword_UserDoesNotExist	Verifică comportamentul serviciului de resetare a parolei pentru un utilizator inexistent.
10.	testUpdatePassword_ValidToken	Testează actualizarea parolei utilizatorului pe baza unui token valid.
11.	testUpdatePassword_InvalidToken	Verifică gestionarea unui token invalid la resetarea parolei.
12.	testIsMovieToWatch_True	Verifică printr-un endpoint dacă un film este adăugat în watchlist.
13.	testAddMovieTotoWatch_Success	Testează adăugarea cu succes a unui film în watchlist prin endpoint API.

8.4.3. Teste Funcționale

Nr.	Test	Ce testează
1.	Test funcțional resetare parolă (succes)	Simulează trimiterea unui email de resetare parolă și salvarea tokenului
2.	Test funcțional resetare parolă (user inexistent)	Verifică aruncarea excepției dacă emailul nu corespunde niciunui utilizator
3.	Test funcțional update parolă (token valid)	Simulează actualizarea parolei folosind un token valid
4.	Test funcțional update parolă (token invalid sau expirat)	Verifică comportamentul pentru token invalid sau expirat
5.	Test funcțional ștergere film favorit	Verifică dacă un film poate fi eliminat corect din lista de favorite
6.	Test funcțional adăugare film favorit (film nou)	Simulează adăugarea unui film nou în favorite cu preluare din API extern
7.	Test funcțional de calculare a statisticilor pentru dashboard-ul unui user	Calculează și returnează statistice dashboard pentru utilizator
8.	Test funcțional de afișare a statisticilor dashboardului unui user atunci când un user nu are date	Returnează valori implicite dacă nu există date pentru utilizator
9.	Test funcțional de adăugare a unui rating de la un user inexistent	Verifică faptul că nu se permite adăugarea ratingului dacă utilizatorul nu există
10.	Test funcțional de adăugare rating pentru un film inexistent in DB	Adaugă rating și salvează film nou dacă nu există în DB
11.	Test funcțional de adăugare rating pentru un film existent în DB	Actualizează ratingul pentru un film deja existent
12.	Test funcțional de ștergere a unui rating pentru un film inexistent în DB	Returnează eroare dacă filmul nu există
13.	Test funcțional de ștergere a unui rating care nu există în DB	Returnează eroare dacă ratingul nu există
14.	Test funcțional de ștergere a unui rating existent	Șterge ratingul existent
15.	Test funcțional care verifică service-ul de recomandare de filme	Generează recomandări pentru un utilizator

16.	Test funcțional de verificare a actualizării matricei de recomandare la adăugarea unui rating.	Reconstruiește matricea și actualizează modelul.

8.5. **Code Snippets:** toate testele au trecut

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.150 s --
[INFO] Running com._errors.MovieMingle.service.DefaultSecureTokenServiceTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.254 s --
[INFO] Running com._errors.MovieMingle.service.RatingServiceTest
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.101 s -
[INFO] Running com._errors.MovieMingle.service.user.DefaultAppUserAccountServiceT
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.169 s --
[INFO] Running com._errors.MovieMingle.service.user.UserFavouritesServiceTest
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.082 s -
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 44, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.729 s
[INFO] Finished at: 2025-05-14T20:48:02+03:00
[INFO] ------------------------------------------------------------------------
PS D:\MovieMingle-MobileApp\Backend>
```

8.5.1. **Test de Integrare** - testGetUserDashboardStats()

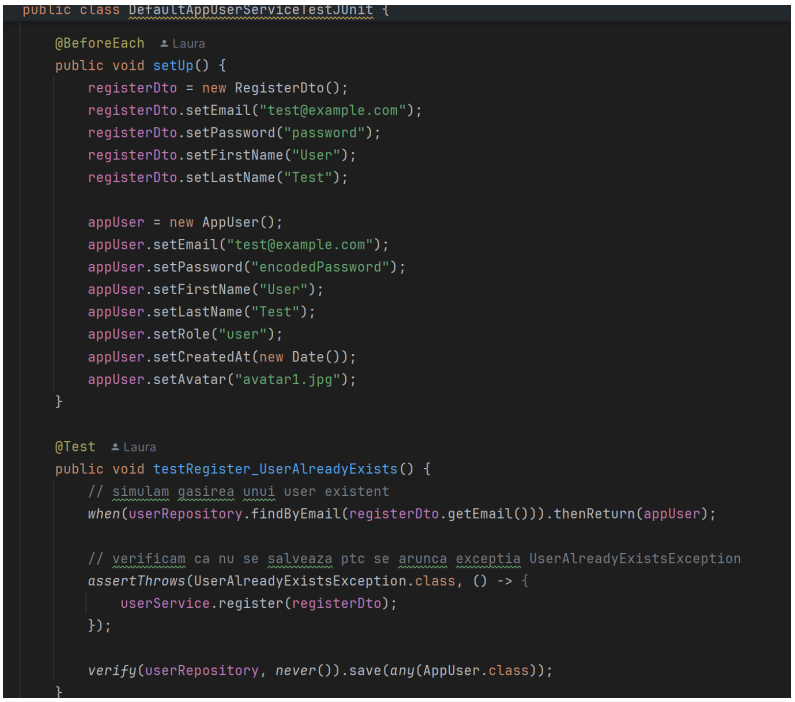
```java
@Test  new *
public void testGetUserDashboardStats() {
    // cream user si il salvam in db
    AppUser user = new AppUser();
    user.setEmail("testuser@example.com");
    user.setPassword("EncodedPassword123");
    user.setRole("user");
    user.setAccountVerified(true);
    userRepository.save(user);

    Long tmdbId = 550L;
    MovieDto movieDto = movieApiClient.getMovie(tmdbId);

    Movie movie = movieRepository.findByTmdbId(tmdbId);
    if (movie == null) {
        movie = new Movie();
        movie.setTMDBId(tmdbId);
        movie.setSeriesTitle(movieDto.getTitle());
        movie.setPosterLink(movieDto.getPosterPath());
        movie.setOverview(movieDto.getDescription());
        movie.setDirector(movieDto.getDirectorsNames() != null ? movieDto.getDirectorsNames() : "Unknown");
        movie.setRuntime(movieDto.getRuntime() != null ? movieDto.getRuntime().toString() : "0");
        movieRepository.save(movie);
    }

    // adaugam filmul la lista de filme vizionate
    UserWatchedMovie watchedMovie = new UserWatchedMovie();
    watchedMovie.setUser(user);
    watchedMovie.setMovie(movie);
    watchedMovie.setWatchedAt(LocalDateTime.now());
    userWatchedMovieRepository.save(watchedMovie);
```

```
    // adaugam un rating pentru film
    Rating rating = new Rating();
    rating.setUser(user);
    rating.setMovie(movie);
    rating.setRating(8);
    ratingRepository.save(rating);

    // request pt a obtine statistica
    UserDashboardDto stats = dashboardService.getUserDashboardStats(Long.valueOf(user.getId()));

    // verificam statistica
    assertNotNull(stats);
    assertEquals( expected: 1, stats.getTotalMoviesWatched());
    assertEquals( expected: 8.0, stats.getAverageRating());
    assertEquals( expected: "Fight Club", stats.getTopRatedMovie());
    assertEquals( expected: "Fight Club", stats.getLowestRatedMovie());
    assertEquals( expected: 2, stats.getTotalHoursWatched()); // 139 minute ≈ 2.31 ore (rotunjit la 2 ore)
    assertNotNull(stats.getMoviesPerMonth());
    assertNotNull(stats.getGenresWatched());

    // verificam daca genul filmului este inclus
    assertTrue(stats.getGenresWatched().containsKey("Drama"));
}
```

Acest test verifică integrarea dintre mai multe componente ale aplicației (bază de date, servicii, API extern) pentru generarea dashboard-ului utilizatorului. Practic, testează fluxul complet de colectare și procesare a datelor despre filmele vizionate, evaluările oferite și genurile preferate.

- Creează un utilizator nou și îl salvează în baza de date.
- Obține detalii despre un film real folosind MovieApiClient (API extern).
- Dacă filmul nu există în DB, îl inserează.
- Adaugă filmul în lista de filme vizionate (UserWatchedMovie).
- Adaugă o notă pentru film în Rating.
- Apelează metoda getUserDashboardStats() din DashboardService pentru a genera statisticile.
- Verifică:
    - numărul total de filme vizionate,
    - media notelor,
    - cel mai apreciat și cel mai slab film,
    - timpul total de vizionare (în ore),
    - filmele vizionate pe lună,
    - genurile de filme.

8.5.2. **Test Unitar** - testRegister_UserAlreadyExists()



```
public class DefaultAppUserServiceTestJUnit {

    @BeforeEach  Laura
    public void setUp() {
        registerDto = new RegisterDto();
        registerDto.setEmail("test@example.com");
        registerDto.setPassword("password");
        registerDto.setFirstName("User");
        registerDto.setLastName("Test");

        appUser = new AppUser();
        appUser.setEmail("test@example.com");
        appUser.setPassword("encodedPassword");
        appUser.setFirstName("User");
        appUser.setLastName("Test");
        appUser.setRole("user");
        appUser.setCreatedAt(new Date());
        appUser.setAvatar("avatar1.jpg");
    }

    @Test  Laura
    public void testRegister_UserAlreadyExists() {
        // simulam gasirea unui user existent
        when(userRepository.findByEmail(registerDto.getEmail())).thenReturn(appUser);

        // verificam ca nu se salveaza ptc se arunca exceptia UserAlreadyExistsException
        assertThrows(UserAlreadyExistsException.class, () -> {
            userService.register(registerDto);
        });

        verify(userRepository, never()).save(any(AppUser.class));
    }
```

Acest test unitar verifică scenariul în care un utilizator încearcă să se înregistreze folosind o adresă de email care este deja asociată unui cont existent în baza de date. Metoda register() din serviciul DefaultAppUserService aruncă o excepție de tip UserAlreadyExistsException dacă există deja un utilizator înregistrat cu adresa de email din formularul de înregistrare (RegisterDto).

Tehnici utilizate:

- **Mocking:** Se simulează comportamentul repository-ului AppUserRepository astfel încât să returneze un AppUser existent.
- **Asserții:** Se utilizează assertThrows() pentru a verifica dacă excepția este aruncată corect.
- **Verificare interacțiuni:** Se folosește verify() pentru a confirma că metoda save() NU este apelată, deoarece înregistrarea nu trebuie să continue.

8.5.3. **Test Funcțional** - testForgottenPassword_Success()

```
private AppUser appUser;  10 usages
private SecureToken secureToken;  12 usages

@BeforeEach  Laura
public void setUp() {
    appUser = new AppUser();
    appUser.setId(123);
    appUser.setEmail("test@example.com");
    appUser.setPassword("oldPassword");

    secureToken = new SecureToken();
    secureToken.setToken("valid-token");
    secureToken.setUser(appUser);

    ReflectionTestUtils.setField(userAccountService,  name: "baseURL",  value: "http://localhost:8080");
}

@Test  Laura
public void testForgottenPassword_Success() throws UnknownIdentifierException, MessagingException {
    when(userService.getUserById("test@example.com")).thenReturn(appUser);
    when(secureTokenService.createSecureToken()).thenReturn(secureToken);
    when(secureTokenRepository.save(any(SecureToken.class))).thenReturn(secureToken);
    doNothing().when(emailService).sendMail(any(ForgotPasswordEmailContext.class));

    userAccountService.forgottenPassword( userName: "test@example.com");

    verify(userService, times( wantedNumberOfInvocations: 1)).getUserById("test@example.com");
    verify(secureTokenService, times( wantedNumberOfInvocations: 1)).createSecureToken();
    verify(secureTokenRepository, times( wantedNumberOfInvocations: 1)).save(any(SecureToken.class));
    verify(emailService, times( wantedNumberOfInvocations: 1)).sendMail(any(ForgotPasswordEmailContext.class));
}
```

Acest test verifică funcționalitatea completă a procesului de recuperare a parolei pentru un utilizator existent. Scenariul testat este unul des întâlnit într-o aplicație reală: atunci când un utilizator solicită resetarea parolei, sistemul trebuie să genereze un token, să îl salveze și să trimită un e-mail cu instrucțiunile de resetare. Testul se ocupă de:

- Identificarea utilizatorului după e-mail
- Generarea unui token securizat de resetare a parolei
- Persistarea tokenului în baza de date
- Trimiterea unui e-mail de recuperare folosind contextul ForgotPasswordEmailContext

9. Testare Backend folosind TestNG

9.1. Ce este TestNG?

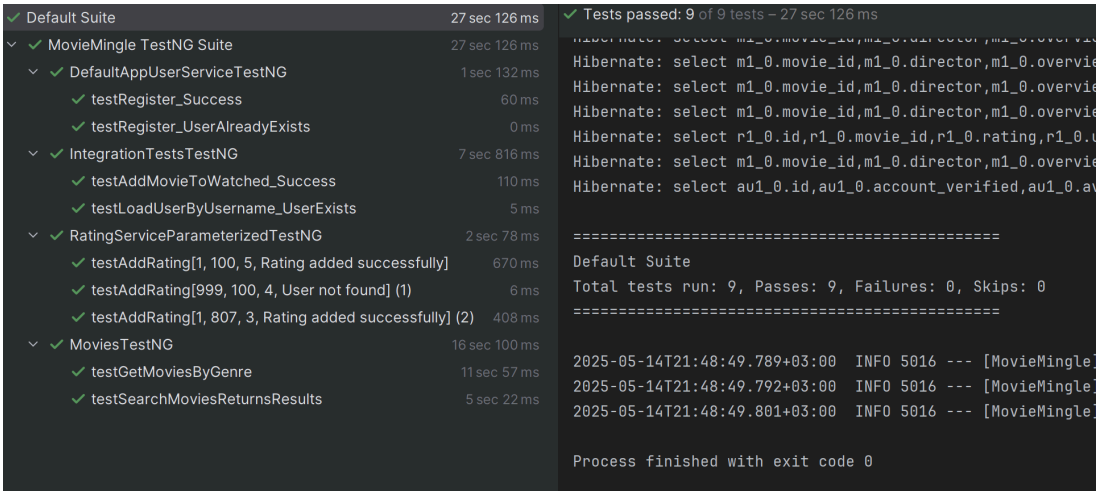
TestNG (Test *Next Generation*) [14] este un framework de testare pentru Java, inspirat din JUnit, dar cu mai multe capabilități puternice pentru testarea complexă, cum ar fi:

- Suport pentru **teste parametrizate**
- Control avansat asupra **ordinii de execuție**
- Gruparea testelor (ex: @Test(groups = "integration"))
- Configurații înainte/după clase, metode, module (@BeforeMethod, @AfterClass etc.)
- Suport nativ pentru **paralelizare** a testelor

9.2. Descrierea testelor scrise cu TestNG

Nr.	Tip	Test	Descriere
1.	**Test Unitar**	testRegister_UserAlreadyExists	Verifică dacă aplicația aruncă o excepție (UserAlreadyExistsException) atunci când se încearcă înregistrarea unui utilizator deja existent în baza de date.
2.	**Test Unitar**	testRegister_Succes	Testează cu succes înregistrarea unui utilizator nou, incluzând salvarea acestuia, generarea unui token și trimiterea e-mailului de confirmare.
3.	**Test de Integrare**	testAddMovieToWatched_Success	Testează integrarea completă între serviciul UserWatchedMovieService și baza de date. Verifică adăugarea unui film în lista de vizionări.
4.	**Test de Integrare**	testLoadUserByUsername_UserExists	Verifică dacă un utilizator real din baza de date poate fi încărcat corect prin CustomUserDetailsService.
5.	**Test Funcțional (+parametrizat)**	testAddRating	Verifică comportamentul metodei addRating pentru mai multe scenarii, inclusiv: utilizator valid + film existent, utilizator inexistent, film nou.
6.	**Test Funcțional**	testSearchMoviesReturnsResults	Verifică dacă metoda getMoviesBySearch() returnează rezultate relevante pentru un cuvânt-cheie ("Inception").
7.	**Test Funcțional**	testGetMoviesByGenre	Verifică dacă sistemul returnează corect filme pentru un anumit gen (ex: comedy = genul 35).

9.3. **Code Snippets:** toate testele au trecut



9.4. **Test Unitar -** testRegister_UserAlreadyExists()

```java
public class DefaultAppUserServiceTestNG {  new *
    private SecureTokenService secureTokenService;
    @Mock  1 usage
    private SecureTokenRepository secureTokenRepository;
    @InjectMocks  3 usages
    private DefaultAppUserService userService;
    private RegisterDto registerDto;  10 usages
    private AppUser appUser;  4 usages

    @BeforeMethod  new *
    public void setUp() {
        MockitoAnnotations.openMocks( testClass: this);

        registerDto = new RegisterDto();
        registerDto.setEmail("test@example.com");
        registerDto.setPassword("password");
        registerDto.setFirstName("User");
        registerDto.setLastName("Test");

        appUser = new AppUser();
        appUser.setEmail("test@example.com");
    }

    @Test(expectedExceptions = UserAlreadyExistsException.class)  new *
    public void testRegister_UserAlreadyExists() throws UserAlreadyExistsException {
        // Simulam existenta utilizatorului în DB
        when(userRepository.findByEmail(registerDto.getEmail())).thenReturn(appUser);
        // Apelul ar trebui să arunce excepția
        userService.register(registerDto);
        // Verificam că nu s-a încercat salvarea în DB
        verify(userRepository, never()).save(any(AppUser.class));
    }
```

Acest test verifică comportamentul metodei register() atunci când un utilizator încearcă să se înregistreze cu o adresă de e-mail deja existentă în sistem. Scop:

- Să se asigure că sistemul aruncă o excepție de tip *UserAlreadyExistsException* în cazul în care e-mailul este deja înregistrat.
- Să verifice că nu se încearcă salvarea în baza de date în acest caz.

9.5. **Test de Integrare -** testAddMovieToWatched_Success()

```java
public class IntegrationTestsTestNG  extends AbstractTestNGSpringContextTests {

    @BeforeMethod  new *
    public void setUp() {
        // Create and save a test user
        user = new AppUser();
        user.setEmail("testng@example.com");
        user.setPassword("secure");
        user.setRole("user");
        user.setAccountVerified(true);
        userRepository.save(user);

        // Create and save a test movie
        movie = new Movie();
        movie.setTMDBId(12345L);
        movie.setSeriesTitle("TestNG Movie");
        movie.setPosterLink("poster.jpg");
        movie.setRuntime("100");
        movie.setDirector("Test Director");
        movie.setOverview("Test Overview");

        movieRepository.save(movie);
    }

    @Test  new *
    public void testAddMovieToWatched_Success() {
        String result = userWatchedMovieService.addMovieToWatched((long) user.getId(), movie.getTmdbId(), movie.getSeriesTitle());
        assertEquals(result,  expected: "Movie added to watched list.");

        boolean isWatched = userWatchedMovieService.isMovieWatched((long) user.getId(), movie.getTmdbId());
        assertTrue(isWatched);
    }
```

Acest test scris cu TestNG validează integrarea completă dintre componentele aplicației pentru scenariul în care un utilizator adaugă un film în lista sa de filme vizionate.
Ce face concret testul:

- Creează și salvează un utilizator real în baza de date de test.
- Creează și salvează un film în baza de date.
- Apelează metoda addMovieToWatched() din serviciul UserWatchedMovieService.
- Verifică dacă filmul a fost marcat corect ca „vizionat” pentru acel utilizator.

9.6. Test Funcțional (parametrizat) - testAddRating()

```
@SpringBootTest  new *
public class RatingServiceParameterizedTestNG extends AbstractTestNGSpringContextTests {

    @Autowired
    private RatingService ratingService;

    @DataProvider(name = "ratingData")  new *
    public Object[][] ratingData() {
        return new Object[][] {
                { 1L, 100L, 5, "Rating added successfully" },
                { 999L, 100L, 4, "User not found" },          // user doesn't exist
                { 1L, 807L, 3, "Rating added successfully" } // new movie
        };
    }

    @Test(dataProvider = "ratingData")  new *
    public void testAddRating(Long userId, Long tmdbId, int ratingValue, String expectedMessage) {
        String result = ratingService.addRating(userId, tmdbId, ratingValue);
        assertEquals(result, expectedMessage);
    }
}
```

Acest test funcțional scris cu TestNG și adnotarea @DataProvider validează metoda addRating() din serviciul RatingService, simulând mai multe scenarii reale:

- un utilizator valid care evaluează un film existent,
- un utilizator inexistent (caz negativ),
- evaluarea unui film nou care nu există încă în baza de date.

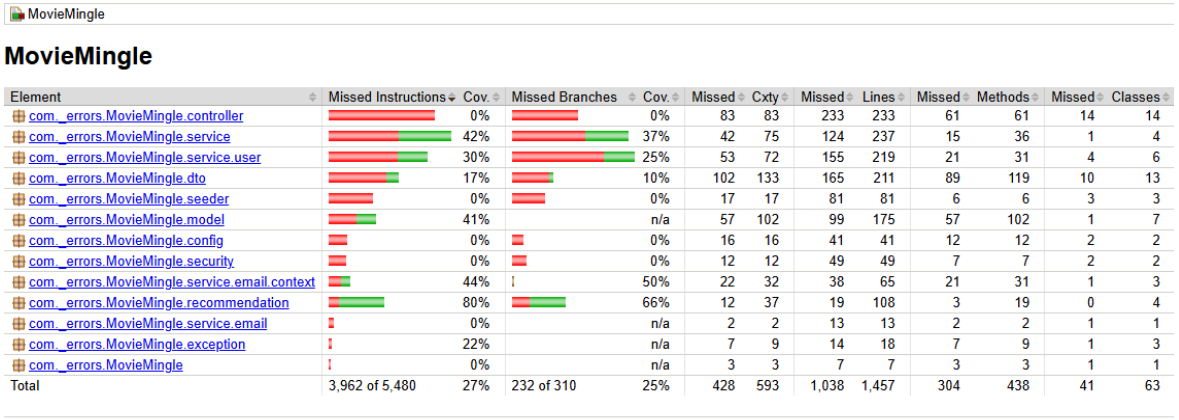
Fiecare caz este testat automat cu date diferite, iar rezultatul este verificat comparând mesajul returnat cu cel așteptat. Acest tip de test ajută la acoperirea logicii de business prin testare multiplă într-un mod concis și eficient.

10. Studiu comparativ între JUnit + Mockito și TestNG

Caracteristică	JUnit + Mockito	TestNG
Tip testare	Testare unitară, de integrare și funcțională (cu Spring Boot)	Testare unitară, de integrare, funcțională și paralelizabilă
Nivel de testare	Clasă, metodă, serviciu, integrare cu repository-uri/mock-uri	Orice nivel (clasă, funcționalitate, suite), cu suport nativ pentru grupuri și priorități
Limbaj	Java	Java
Structură și configurare	Simplu și intuitiv (anotări standard, suport Spring)	Mai configurabil, dar puțin mai detaliat în setup
Suport parametrizare	Limitat (folosind @ParameterizedTest + @CsvSource, etc.)	Puternic, nativ prin @DataProvider și @Factory
Testare paralelă	Posibilă cu extensii/configurări suplimentare	Nativă, configurabilă pe clasă, metodă sau test suite
Control asupra codului testat	Complet (Mockito pentru mock, spy, verify, etc.)	La fel ca JUnit (se poate integra ușor cu Mockito)
Viteză de execuție	Rapidă în combinație cu Mockito și Spring test	Ușor mai rapid în rulări paralele
Complexitate cod test	Simplu, clar, orientat pe metode individuale	Mai flexibil, dar cu sintaxă mai încărcată în unele cazuri
Detectare erori	Detectează bug-uri logice și comportamentale	Similar, dar oferă și control granular prin grupuri și priorități
CI/CD	Suport total (GitHub Actions, Jenkins, etc.)	Suport complet, inclusiv configurare paralelă
Scalabilitate proiect mare	Foarte bună pentru microservicii și aplicații Spring Boot	Scalabil, mai ales pentru proiecte enterprise unde paralelismul contează
Cazuri de utilizare ideale	Testarea serviciilor, logicii interne, REST APIs, cu mock-uri	Testare funcțională, paralelizată, testare data driven

11. Gradul de acoperire cu teste Backend - JaCoCo

JaCoCo (Java Code Coverage) [15] este un instrument care măsoară **gradul de acoperire cu teste** al codului sursă Java.



MovieMingle

MovieMingle

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com._errors.MovieMingle.controller		0%		0%	83	83	233	233	61	61	14	14
com._errors.MovieMingle.service		42%		37%	42	75	124	237	15	36	1	4
com._errors.MovieMingle.service.user		30%		25%	53	72	155	219	21	31	4	6
com._errors.MovieMingle.dto		17%		10%	102	133	165	211	89	119	10	13
com._errors.MovieMingle.seeder		0%		0%	17	17	81	81	6	6	3	3
com._errors.MovieMingle.model		41%		n/a	57	102	99	175	57	102	1	7
com._errors.MovieMingle.config		0%		0%	16	16	41	41	12	12	2	2
com._errors.MovieMingle.security		0%		0%	12	12	49	49	7	7	2	2
com._errors.MovieMingle.service.email.context		44%		50%	22	32	38	65	21	31	1	3
com._errors.MovieMingle.recommendation		80%		66%	12	37	19	108	3	19	0	4
com._errors.MovieMingle.service.email		0%		n/a	2	2	13	13	2	2	1	1
com._errors.MovieMingle.exception		22%		n/a	7	9	14	18	7	9	1	3
com._errors.MovieMingle		0%		n/a	3	3	7	7	3	3	1	1
Total	3,962 of 5,480	27%	232 of 310	25%	428	593	1,038	1,457	304	438	41	63

Pentru a evalua calitatea testării unitare și de integrare din cadrul backend-ului aplicației **MovieMingle**, am utilizat tool-ul de analiză **JaCoCo**, care măsoară acoperirea codului prin teste. Rezultatele generate evidențiază în mod clar pachetele bine acoperite și zonele care necesită îmbunătățiri.

Rezumat general:

- Acoperire totală a instrucțiunilor: **53%**
- Acoperire totală a ramurilor de decizie: **41%**
- Clase acoperite: **63 din 69 (91%)**
- Linii netestate: **1457 din 2119**

Zone bine acoperite:

Pachet	**Acoperire**
seeder	92%
recommendation	80%
service.email.context	71%

Zone cu acoperire moderată:

Pachet	**Acoperire**
service	65%
model	56%
dto	48%

Zone slab acoperite:

Pachet	Acoperire
controller	9%
security	18%
service.email	5%

12. Interpretarea acoperirii codului

Analiza realizată ne oferă o imagine echilibrată asupra nivelului de testare atins în acest moment. Sunt câteva zone importante unde testarea este bine implementată și contribuie clar la stabilitatea aplicației. În special modulele care țin de generarea datelor (seedere) și sistemul de recomandări au o acoperire solidă, ceea ce e important, fiind componente esențiale în funcționarea aplicației.

Pe de altă parte, există și zone cu acoperire scăzută care ridică semne de întrebare. Controller-ele, de exemplu, sunt aproape netestate, ceea ce înseamnă că interacțiunea cu API-ul nu este verificată automat. În cazul aplicațiilor de tip REST, acest lucru poate duce la erori funcționale greu de depistat manual. La fel, partea de securitate are o acoperire slabă, iar în lipsa testelor specifice, e dificil să avem siguranța că mecanismele de autentificare și autorizare funcționează corect în toate scenariile.

De asemenea, serviciul care gestionează trimiterea de e-mailuri și clasele de tip exception nu sunt suficient acoperite. În ambele cazuri, este important să existe teste care să verifice nu doar funcționarea normală, ci și comportamentul în caz de erori — pentru a evita probleme neașteptate în producție.

Faptul că zonele critice precum logica de business sunt testate consistent reprezintă un punct foarte bun de plecare. Totuși, pentru a ajunge la un nivel de încredere ridicat în stabilitatea aplicației, ar fi util ca testarea să fie extinsă în mod specific în zonele cu risc funcțional sau de securitate.

13. Concluzii

În cadrul proiectului Movie Mingle Mobile App, testarea a fost tratată în mod riguros și implementată pe multiple niveluri ale aplicației, asigurând o acoperire coerentă și relevantă atât pentru interfața utilizator, cât și pentru logica internă. Frontend-ul beneficiază de teste unitare și de integrare dezvoltate cu Jest și React Native Testing Library, acestea fiind orientate pe verificarea comportamentului componentelor și a interacțiunilor specifice. Testarea end-to-end este realizată prin framework-ul Maestro, utilizând scenarii declarative în YAML care simulează fluxuri reale de utilizator și validează funcționalități complete în

mediul mobil. La nivelul backend-ului, au fost utilizate JUnit, Mockito și TestNG pentru a valida corectitudinea metodelor, interacțiunea dintre servicii și tratarea excepțiilor. Prezența acestor forme complementare de testare reflectă o abordare bine structurată a procesului de asigurare a calității software, contribuind la creșterea fiabilității aplicației și la susținerea unei dezvoltări scalabile și predictibile.

14. Bibliografie

[1] M. Ehmer and F. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, 2012, https://doi.org/10.14569/ijacsa.2012.030603. Data ultimei accesări: 15 mai 2025

[2] Getting Started · Jest, https://jestjs.io/docs/getting-started. Data ultimei accesări: 25 aprilie 2025

[3] React Native Testing Library, https://callstack.github.io/react-native-testing-library/. Data ultimei accesări: 25 aprilie 2025

[4] Introduction to Expo, https://docs.expo.dev/. Data ultimei accesări: 27 aprilie 2025

[5] Unit testing, https://docs.expo.dev/develop/unit-testing/. Data ultimei accesări: 26 aprilie 2025

[6] Mock Functions · Jest, https://jestjs.io/docs/mock-function-api. Data ultimei accesări: 26 aprilie 2025

[7] What is Maestro?, https://docs.maestro.dev/. Data ultimei accesări: 27 aprilie 2025

[8] Our journey into mobile E2E Testing, https://medium.com/@heitorcolangelo/our-journey-into-mobile-e2e-testing-0f0236122773. Data ultimei accesări: 27 aprilie 2025

[9] Detox, https://wix.github.io/Detox/. Data ultimei accesări: 27 aprilie 2025

[10] Maestro Mobile UI Testing Framework: A Beginner's Guide (2024), https://www.testdevlab.com/blog/getting-started-with-maestro-mobile-ui-testing-framework. Data ultimei accesări: 27 aprilie 2025

[11] JUnit 5 User Guide, https://junit.org/junit5/docs/current/user-guide/. Data ultimei accesări: 15 mai 2025

[12] Mockito Framework Site, https://site.mockito.org/. Data ultimei accesări: 15 mai 2025

[13] Gurubasava, "Unit Testing Frameworks: A comparative study.", https://www.irjet.net/archives/V9/i7/IRJET-V9I7437.pdf. Data ultimei accesări: 15 mai 2025

[14] TestNG Documentation, https://testng.org/. Data ultimei accesări: 15 mai 2025

[15] JaCoCo - Maven Plug-in, https://www.eclemma.org/jacoco/trunk/doc/maven.html. Data ultimei accesări: 15 mai 2025