

Raport Teste AI

1) Functia `post /review`, teste functionale:

- Am inclus in prompt functia si cerinta de a genera teste functionale pentru aceasta:

Functia de mai jos posteaza un review intr-o aplicatie web despre carti. Scribe te rog teste functionale pentru aceasta functie.

```
import express from 'express';
import Database from 'better-sqlite3';
import { verifyToken } from './authentication.js';
import { db } from './config.js';
```

```
const router = express.Router();
router.post('/', verifyToken, (req, res) => {
  const { book_id, rating } = req.body;
  const user_id = req.user.id;
```

... (restul functiei)

- Raspuns AI:

Pentru această funcție de tip POST care adaugă un review într-o aplicație web despre cărți, putem scrie **teste funcționale** folosind un framework precum **Jest** împreună cu **Supertest** pentru a simula cereri HTTP. Vom acoperi următoarele cazuri:

✅ Cazuri de test recomandate:

1. Review adăugat cu succes
2. Lipsă câmpuri obligatorii (`book_id` sau `rating`)
3. Rating invalid (non-numeric, sub 1, peste 5)
4. Review duplicat
5. Eroare internă la inserare în DB



6. Token lipsă sau invalid (autentificare)

 Exemplu testare funcțională (review.test.js):

```
js Copy Edit  
  
import request from 'supertest';  
import express from 'express';  
import reviewRouter from './calea_catre_router.js';  
import { db } from '../config.js';  
import { verifyToken } from './authentication.js';  
  
// Mock database și autentificare
```

... (restul codului)

- După rularea codului generat și corectarea erorilor legate de rute, am obținut:

```
PASS functional_tests/reviewsAI.test.js
```

Codul generat:

```
import request from 'supertest';  
import express from 'express';  
import reviewRouter from '../routes/reviews.js';  
import { db } from '../config.js';  
import { verifyToken } from '../routes/authentication.js';  
  
// Mock database și autentificare  
jest.mock('../config.js', () => ({  
  db: {  
    prepare: jest.fn(),  
  },  
}));  
  
jest.mock('../routes/authentication.js', () => ({  
  verifyToken: (req, res, next) => {  
    req.user = { id: 1 }; // mock user  
    next();  
  },  
}));  
  
const app = express();
```

```
app.use(express.json());
app.use('/reviews', reviewRouter);

describe('POST /reviews', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  test('✅ should add review successfully', async () => {
    db.prepare.mockImplementation((sql) => {
      if (sql.includes('SELECT')) {
        return { get: () => null }; // no existing review
      }
      if (sql.includes('INSERT')) {
        return { run: jest.fn() };
      }
    });

    const response = await request(app)
      .post('/reviews')
      .send({ book_id: 10, rating: 4 });

    expect(response.status).toBe(201);
    expect(response.text).toBe('Review added successfully!');
  });

  test('❌ should return 400 if fields are missing', async () => {
    const response = await request(app).post('/reviews').send({
      rating: 4 });
    expect(response.status).toBe(400);
    expect(response.text).toBe('All fields are required!');
  });

  test('❌ should return 400 if rating is invalid', async () => {
    const response = await request(app).post('/reviews').send({
      book_id: 5, rating: 7 });
    expect(response.status).toBe(400);
    expect(response.text).toBe('Rating must be a number between 1
and 5.');
```

```

    test('❌ should return 400 if review already exists', async ()
=> {
      db.prepare.mockImplementation((sql) => {
        if (sql.includes('SELECT')) {
          return { get: () => ({ id: 1 }) }; // review already
exists
        }
      });

      const response = await request(app)
        .post('/reviews')
        .send({ book_id: 10, rating: 4 });

      expect(response.status).toBe(400);
      expect(response.text).toBe('You have already added a review
for this book.');
```

```

    });

    test('❌ should return 500 if database error occurs', async ()
=> {
      db.prepare.mockImplementation(() => {
        throw new Error('DB crash');
      });

      const response = await request(app)
        .post('/reviews')
        .send({ book_id: 1, rating: 3 });

      expect(response.status).toBe(500);
      expect(response.text).toBe('Error adding review!');
```

```

    });
  });

```

- Generare teste structurale:

```
Scrie te rog si teste structurale pentru aceeasi functie:
import express from 'express';
import Database from 'better-sqlite3';
import { verifyToken } from './authentication.js';
import { db } from './config.js';

const router = express.Router();
router.post('/', verifyToken, (req, res) => {
  const { book_id, rating } = req.body;
  const user_id = req.user.id;

  if (!book_id || !rating) {
    return res.status(400).send("All fields are required!");
  }
  ↓
```

... (restul funcției)

- Raspuns AI:

Pentru a scrie **teste structurale (unitare)** pentru funcția de mai sus **fără a restructura codul complet**, vom simula cererile Express folosind mock-uri pentru `req`, `res` și `db.prepare`. De asemenea, vom **extrage logica handlerului** într-o funcție separată pentru a putea fi testată independent, fără router sau middleware.

✅ Pas 1: Extrage logica într-un handler pur

Creează un fișier nou:

handlers/addReview.js

js

Copy

Edit

```
export const addReviewHandler = (db) ↓ (req, res) => {
```

✓ Pas 2: Scrie testele structurale

tests/addReviewHandler.test.js

```
js Copy Edit

import { addReviewHandler } from '../handlers/addReview';

describe('addReviewHandler', () => {
  const mockPrepare = jest.fn();
  const mockDb = { prepare: mockPrepare };

  const mockRes = {
    status: jest.fn().mockReturnThis(),
  }
}
```

A sugerat extragerea logicii handlerului intr-o functie separata, in alt fisier.

In urma testarii codului, am obtinut teste corecte, care au trecut.

2) Funcția `post/comment`, teste functionale:

- Am inclus în prompt funcția și cerința de a genera teste funcționale pentru aceasta

Am o funcție în javascript și îmi doresc să scriu teste funcționale pentru ea
Crezi că ai putea să îmi scrii aceste teste?

```
router.post('/', verifyToken, (req, res) => {
  const { book_id, content } = req.body;
  const user_id = req.user.id;

  if (!book_id || !content) {
    return res.status(400).send('All fields are required!');
  }

  if (typeof book_id !== 'number' || book_id <= 0){
    return res.status(400).send('Invalid book ID!');
  }

  if(content.length < 5) {
    return res.status(400).send('Content is too short!');
  }

  if(content.length > 500) {
    return res.status(400).send('Content is too long!');
  }
}
```

- Răspunsul pe care l-am primit este următorul:

Desigur! Pentru a testa funcțional această rută (`POST /`) care adaugă un comentariu, putem folosi **Jest** împreună cu **Supertest**, presupunând că folosești **Express** și ai o aplicație exportată (`app.js` sau similar).

🔧 **Presupuneri:**

- Folosești `Express`.
- Aplicația este exportată ca `app`.
- Middleware-ul `verifyToken` atașează un obiect `user` în `req` (ex: `req.user = { id: 1 }`).
- Baza de date este fie *mock-uită*, fie este o instanță de test.

- Și următorul cod:

```
const request = require('supertest');
const app = require('../app'); // înlocuiește cu calea ta reală

jest.mock('../middleware/verifyToken', () => (req, res, next) => {
  req.user = { id: 1 }; // mock user
  next();
});

describe('POST /comments', () => {
  it('should return 400 if book_id or content is missing', async
() => {
    const res = await request(app)
      .post('/')
      .send({ content: 'Valid comment' });

    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('All fields are required!');
  });

  it('should return 400 for invalid book_id', async () => {
    const res = await request(app)
      .post('/')
      .send({ book_id: -1, content: 'Valid content' });

    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('Invalid book ID!');
  });
});
```

```

it('should return 400 if content is too short', async () => {
  const res = await request(app)
    .post('/')
    .send({ book_id: 1, content: '1234' });

  expect(res.statusCode).toBe(400);
  expect(res.text).toBe('Content is too short!');
});

it('should return 400 if content is too long', async () => {
  const longContent = 'a'.repeat(501);
  const res = await request(app)
    .post('/')
    .send({ book_id: 1, content: longContent });

  expect(res.statusCode).toBe(400);
  expect(res.text).toBe('Content is too long!');
});

it('should return 201 for a valid comment', async () => {
  // mock pentru baza de date
  const db = require('../db'); // presupunem că ai acest
  fișier

  const runMock = jest.fn();
  db.prepare = jest.fn().mockReturnValue({ run: runMock });

  const res = await request(app)
    .post('/')
    .send({ book_id: 1, content: 'This is a valid comment'
  });

  expect(res.statusCode).toBe(201);
  expect(res.text).toBe('Comment added successfully!');
  expect(runMock).toHaveBeenCalledWith(1, 1, 'This is a
  valid comment');
});

it('should return 500 if db throws an error', async () => {
  const db = require('../db');
  db.prepare = jest.fn().mockReturnValue({
    run: () => { throw new Error('DB error'); }
  });
});

```



```

    const res = await request(app)
      .post('/')
      .send({ book_id: 1, content: 'Another valid comment'
    });

    expect(res.statusCode).toBe(500);
    expect(res.text).toBe('Error adding comment!');
  });
});

```

- I-am cerut apoi să îmi scrie sintaxa cu import-uri, iar codul pe care mi l-a dat este următorul:

eu folosesc sintaxa cu import uri

Perfect! Dacă folosești sintaxa modernă cu `import` (ESModules), atunci vom adapta fișierul de test în consecință. Iată o versiune completă folosind `import`, cu `Jest` + `Supertest`.

✓ Presupuneri:

- Ai un fișier `app.js` sau `app.mjs` care exportă aplicația Express (`export default app`).
- Middleware-ul `verifyToken` adaugă `req.user = { id: 1 }`.
- Folosești o bază de date definită într-un modul (ex. `db.js` sau `db.mjs`).
- Rulezi testele cu Jest configurat pentru ESM (ex: `"type": "module"` în `package.json`).

```

import request from 'supertest';
import app from '../app.js'; // actualizează calea în funcție de
structura ta
import { jest } from '@jest/globals';

// Mock pentru middleware-ul verifyToken
jest.unstable_mockModule('../middleware/verifyToken.js', () => ({
  default: (req, res, next) => {
    req.user = { id: 1 }; // user mock
    next();
  }
}));

```

```

    }
  }));

// Mock pentru baza de date
import * as dbModule from '../db.js';

describe('POST /comments', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('should return 400 if book_id or content is missing', async () => {
    const res = await request(app).post('/').send({ content:
'valid content' });
    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('All fields are required!');
  });

  it('should return 400 for invalid book_id', async () => {
    const res = await request(app).post('/').send({ book_id: -1,
content: 'valid content' });
    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('Invalid book ID!');
  });

  it('should return 400 if content is too short', async () => {
    const res = await request(app).post('/').send({ book_id: 1,
content: '1234' });
    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('Content is too short!');
  });

  it('should return 400 if content is too long', async () => {
    const longContent = 'a'.repeat(501);
    const res = await request(app).post('/').send({ book_id: 1,
content: longContent });
    expect(res.statusCode).toBe(400);
    expect(res.text).toBe('Content is too long!');
  });

  it('should return 201 for a valid comment', async () => {

```

```

const runMock = jest.fn();
jest.spyOn(dbModule, 'default').mockReturnValue({
  prepare: () => ({
    run: runMock
  })
});

const res = await request(app).post('/').send({ book_id: 1,
content: 'A valid comment' });
expect(res.statusCode).toBe(201);
expect(res.text).toBe('Comment added successfully!');
expect(runMock).toHaveBeenCalledWith(1, 1, 'A valid comment');
});

it('should return 500 if db throws an error', async () => {
  jest.spyOn(dbModule, 'default').mockReturnValue({
    prepare: () => ({
      run: () => {
        throw new Error('DB error');
      }
    })
  });

  const res = await request(app).post('/').send({ book_id: 1,
content: 'Valid again' });
  expect(res.statusCode).toBe(500);
  expect(res.text).toBe('Error adding comment!');
});
});

```

```

Tests:      59 passed, 59 total
Snapshots: 0 total
Time:     4.272 s, estimated 17 s

```

Testele au trecut fără probleme și fără să fie nevoie să modific codul.

- Testele structurale:

imi poti face si teste structurale pentru aceeaasi functie?

Absolut! Prin **teste structurale** (cunoscute și ca *unit tests*), ne concentrăm pe testarea **comportamentului intern** al funcției/rutei, izolat de contextul extern (ca Express, HTTP sau baza de date). Asta înseamnă că:

- Extragem logica într-o funcție pură, ușor de testat.
- Mock-uim tot ce ține de Express (`req` , `res`).
- Nu facem requesturi HTTP reale (spre deosebire de testele funcționale).

Ce vom face:

1. Extragem handler-ul într-o funcție separată (`handleAddComment.js`).
2. Scriem teste pentru acea funcție, fără server, folosind `jest` .

```
export default function handleAddComment(req, res, db) {
  const { book_id, content } = req.body;
  const user_id = req.user.id;

  if (!book_id || !content) {
    return res.status(400).send('All fields are required!');
  }

  if (typeof book_id !== 'number' || book_id <= 0) {
    return res.status(400).send('Invalid book ID!');
  }

  if (content.length < 5) {
    return res.status(400).send('Content is too short!');
  }

  if (content.length > 500) {
    return res.status(400).send('Content is too long!');
  }

  const q = 'INSERT INTO comments (user_id, book_id, content) VALUES (?, ?, ?)';
  try {
    db.prepare(q).run(user_id, book_id, content);
    return res.status(201).send('Comment added successfully!');
  } catch (error) {
    console.error('Error adding comment: ', error.message);
    return res.status(500).send('Error adding comment!');
  }
}
```

```
}  
}
```

```
import handleAddComment from '../handlers/handleAddComment.js';  
  
describe('handleAddComment (unit tests)', () => {  
  let req, res, db, runMock;  
  
  beforeEach(() => {  
    runMock = jest.fn();  
    req = {  
      body: {},  
      user: { id: 1 }  
    };  
    res = {  
      status: jest.fn().mockReturnThis(),  
      send: jest.fn()  
    };  
    db = {  
      prepare: jest.fn().mockReturnValue({ run: runMock })  
    };  
  });  
  
  it('should return 400 if required fields are missing', () => {  
    handleAddComment(req, res, db);  
    expect(res.status).toHaveBeenCalledWith(400);  
    expect(res.send).toHaveBeenCalledWith('All fields are  
required!');  
  });  
  
  it('should return 400 if book_id is invalid', () => {  
    req.body = { book_id: -2, content: 'Valid content' };  
    handleAddComment(req, res, db);  
    expect(res.status).toHaveBeenCalledWith(400);  
    expect(res.send).toHaveBeenCalledWith('Invalid book ID!');  
  });  
  
  it('should return 400 if content is too short', () => {  
    req.body = { book_id: 1, content: '1234' };  
    handleAddComment(req, res, db);  
    expect(res.status).toHaveBeenCalledWith(400);  
  });  
});
```

```

    expect(res.send).toHaveBeenCalledWith('Content is too short!');
  });

  it('should return 400 if content is too long', () => {
    req.body = { book_id: 1, content: 'a'.repeat(501) };
    handleAddComment(req, res, db);
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.send).toHaveBeenCalledWith('Content is too long!');
  });

  it('should insert comment and return 201 on success', () => {
    req.body = { book_id: 1, content: 'Valid comment' };
    handleAddComment(req, res, db);
    expect(db.prepare).toHaveBeenCalled();
    expect(runMock).toHaveBeenCalledWith(1, 1, 'Valid comment');
    expect(res.status).toHaveBeenCalledWith(201);
    expect(res.send).toHaveBeenCalledWith('Comment added
successfully!');
  });

  it('should return 500 if DB throws an error', () => {
    req.body = { book_id: 1, content: 'Valid comment' };
    db.prepare = () => ({
      run: () => { throw new Error('DB error'); }
    });

    handleAddComment(req, res, db);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.send).toHaveBeenCalledWith('Error adding comment!');
  });
});

```

3) Funcția de **returnare a cărții**

3.1) Teste funcționale

Am inclus în prompt funcția și i-am cerut să îmi scrie teste funcționale pentru aceasta:

am o functie in javascript care returneaza cartile unui utilizator intr-o aplicatie web. Imi poti scrie teste functionale pentru ea utilizand jest cu mock-uri. Aceasta este functia:

am o functie in javascript care returneaza cartile unui utilizator intr-o aplicatie web. Imi poti scrie teste functionale pentru ea utilizand jest cu mock-uri. Aceasta este functia:

```
import express from "express";
import Database from "better-sqlite3";
import { db } from "../config.js";

const router = express.Router();

router.get("/:user_id/:book_id", (req, res) => {
  const { user_id, book_id } = req.params;
  const q = "SELECT * FROM library WHERE user_id = ? AND book_id = ?";

  try {
    if(!user_id.trim() || !book_id.trim() || isNaN(user_id) ||
    isNaN(book_id)){
      console.error("Invalid user id or book id provided.");
      return res.status(400).send("Invalid user id or book id
provided.");
    }
    const book = db.prepare(q).get(user_id, book_id);

    if (book) {
      return res.status(200).json(book);
    } else {
      const userExists = db.prepare("SELECT * FROM library WHERE
user_id = ?").get(user_id);
      if (userExists) {
        return res.status(404).send("The book is not in your library.");
      }
      else{
        return res.status(404).send("The user does not exist.");
      }
    }
  } catch (error) {
    console.error("There is an error in userid+bookid : ",
error.message);
    res.status(500).send("There is an error processing your request.");
  }
});
```

Răspunsul primit a fost:

Pentru a testa funcția ta cu Jest și mock-uri, trebuie să:

1. Mock-uim modulul `better-sqlite3` sau obiectul `db` importat.
2. Folosim `supertest` pentru a face request-uri către ruta Express.
3. Testăm diverse cazuri: input invalid, carte găsită, carte inexistentă dar user existent, user inexistent, eroare internă.

Setup de bază:

Assumând că `db` este exportat dintr-un fișier `.config.js` ca:

```
js
// .config.js
import Database from 'better-sqlite3';
export const db = new Database('my-db.db');
```

Codul de test `libraryRoute.test.js`:

```
js
import request from 'supertest';
import express from 'express';
import { db } from '../.config.js'; // acesta va fi mock-uit
import router from '../calea/spre/routerulTau.js'; // înlocuiește cu path-ul real
```

Asigură-te că:

1. Routerul tău este exportat corect (`export default router;`)
2. Jest este configurat în proiect (`npm install --save-dev jest supertest`)
3. Rulezi testele cu `jest` sau `npx jest`.

Dacă vrei, pot să-ți generezi și un fișier complet de configurare `package.json` cu scriptul de test, spune-mi.

Codul pe care mi l-a dat după ce am făcut modificările indicate este:

```
import request from 'supertest';
import express from 'express';
import { db } from '../.config.js'; // acesta va fi mock-uit
import router from '../routes//library.js'; // înlocuiește cu
path-ul real

jest.mock('../.config.js', () => {
  const mockPrepare = jest.fn();
  return {
```



```

        db: {
            prepare: mockPrepare
        }
    };
});

describe('GET /:user_id/:book_id', () => {
    const app = express();
    app.use('/', router);

    afterEach(() => {
        jest.clearAllMocks();
    });

    it('should return 400 for invalid input', async () => {
        const res = await request(app).get('/ /xyz');
        expect(res.status).toBe(400);
        expect(res.text).toBe("Invalid user id or book id
provided.");
    });

    it('should return 200 and book data if book exists', async () =>
    {
        const bookMock = { book_id: 1, user_id: 2, title: "Mock
Book" };

        // Simulează apeluri consecutive prepare().get()
        db.prepare
            .mockReturnValueOnce({ get: () => bookMock }); // prima
interogare - cartea există

        const res = await request(app).get('/2/1');
        expect(res.status).toBe(200);
        expect(res.body).toEqual(bookMock);
    });
});

```

```

    it('should return 404 if book not in user library but user
exists', async () => {
      db.prepare
        .mockReturnValueOnce({ get: () => undefined }) // prima
interogare - cartea nu există
        .mockReturnValueOnce({ get: () => ({ user_id: 2 }) });
// a doua - userul există

      const res = await request(app).get('/2/999');
      expect(res.status).toBe(404);
      expect(res.text).toBe("The book is not in your library.");
    });

    it('should return 404 if user does not exist', async () => {
      db.prepare
        .mockReturnValueOnce({ get: () => undefined }) // cartea
nu există
        .mockReturnValueOnce({ get: () => undefined }); //
userul nu există

      const res = await request(app).get('/999/1');
      expect(res.status).toBe(404);
      expect(res.text).toBe("The user does not exist.");
    });

    it('should return 500 on internal error', async () => {
      db.prepare.mockImplementation(() => {
        throw new Error("Database failure");
      });

      const res = await request(app).get('/2/1');
      expect(res.status).toBe(500);
      expect(res.text).toBe("There is an error processing your
request.");
    });
  });
});

```

Rulând testele cu `jest --coverage`, am obținut:

```
PASS ai_tests/functional_tests_library.test.js
  GET /:user_id/:book_id
    ✓ should return 400 for invalid input (64 ms)
    ✓ should return 200 and book data if book exists (6 ms)
    ✓ should return 404 if book not in user library but user exists (3 ms)
    ✓ should return 404 if user does not exist (5 ms)
    ✓ should return 500 on internal error (11 ms)

-----|-----|-----|-----|-----|-----
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100   |    100   |    100   |    100   |
library.js |    100   |    100   |    100   |    100   |
-----|-----|-----|-----|-----|-----
Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.928 s, estimated 2 s
Ran all test suites.
```

3.2) Teste structurale

I-am cerut să-mi scrie și testele structurale pentru aceeași funcție:

imi poti oferi si testele structurale pentru aceeasi functie?

Desigur! **Testele structurale** (cunoscute și ca teste unitare) se concentrează pe **comportamentul intern al funcției**, nu pe integrarea cu HTTP sau Express. Asta înseamnă că testăm direct **logica funcției** care returnează cartea unui utilizator, fără a porni un server Express sau a face requesturi HTTP.

Pentru a face acest lucru:

1. Este necesar să **extragem logica rutei** într-o funcție separată, care poate fi testată individual.
2. Apoi vom scrie teste unitare care mock-uiesc interacțiunile cu baza de date și verifică direct rezultatele funcției.

✓ 1. Refactorizare: extrage logica într-o funcție

Codul de refactorizare oferit este:

```
// libraryService.js
export function getUserBook(db, user_id, book_id) {
```

```

    const q = "SELECT * FROM library WHERE user_id = ? AND book_id = ?";

    if (!user_id.trim() || !book_id.trim() || isNaN(user_id) || isNaN(book_id)) {
        throw { status: 400, message: "Invalid user id or book id provided." };
    }

    const book = db.prepare(q).get(user_id, book_id);

    if (book) {
        return { status: 200, data: book };
    } else {
        const userExists = db.prepare("SELECT * FROM library WHERE user_id = ?").get(user_id);
        if (userExists) {
            throw { status: 404, message: "The book is not in your library." };
        } else {
            throw { status: 404, message: "The user does not exist." };
        }
    }
}

```

Apoi în router rămâne:

```

import { getUserBook } from "../libraryService.js";

router.get("/:user_id/:book_id", (req, res) => {
    const { user_id, book_id } = req.params;
    try {
        const result = getUserBook(db, user_id, book_id);
        res.status(result.status).json(result.data);
    } catch (err) {

```

```
        res.status(err.status || 500).send(err.message || "Unknown  
error");  
    }  
});
```

Codul oferit pentru teste:

```
import { getUserBook } from '../routes/libraryService.js'; //  
ajustează path-ul real  
  
describe('getUserBook', () => {  
    let mockDb;  
  
    beforeEach(() => {  
        mockDb = {  
            prepare: jest.fn()  
        };  
    });  
  
    it('throws 400 if input is invalid', () => {  
        expect(() => getUserBook(mockDb, ' ',  
'abc')).toThrow(expect.objectContaining({  
            status: 400,  
            message: "Invalid user id or book id provided."  
        }));  
    });  
  
    it('returns 200 and book data if book is found', () => {  
        const mockBook = { book_id: 1, user_id: 2, title: "Book 1"  
    };  
  
        mockDb.prepare.mockReturnValueOnce({  
            get: () => mockBook  
        });  
  
        const result = getUserBook(mockDb, '2', '1');  
        expect(result).toEqual({ status: 200, data: mockBook });  
    });  
});
```

```

    it('throws 404 if book not found but user exists', () => {
      mockDb.prepare
        .mockReturnValueOnce({ get: () => undefined }) // prima:
cartea
        .mockReturnValueOnce({ get: () => ({ user_id: 2 }) });
// a doua: userul

      expect(() => getUserBook(mockDb, '2',
'99')).toThrow(expect.objectContaining({
        status: 404,
        message: "The book is not in your library."
      }));
    });

    it('throws 404 if user does not exist', () => {
      mockDb.prepare
        .mockReturnValueOnce({ get: () => undefined }) // cartea
        .mockReturnValueOnce({ get: () => undefined }); //
userul

      expect(() => getUserBook(mockDb, '999',
'1')).toThrow(expect.objectContaining({
        status: 404,
        message: "The user does not exist."
      }));
    });
  });
});

```

Rezumat

Tip test	Ce verifică	Tool-uri
Funcțional (integrare)	Request HTTP, răspuns corect	Jest + Supertest
Structural (unitar)	Logica internă fără Express	Jest (cu mock-uri)

Toate testele au rulat cu succes:

```
PASS ai_tests/structural_tests_library.test.js
getUserBook
  ✓ throws 400 if input is invalid (3 ms)
  ✓ returns 200 and book data if book is found (1 ms)
  ✓ throws 404 if book not found but user exists (1 ms)
  ✓ throws 404 if user does not exist (1 ms)

-----|-----|-----|-----|-----|-----
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100   |    100   |    100   |    100   |
libraryService.js |    100   |    100   |    100   |    100   |
-----|-----|-----|-----|-----|-----
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.792 s, estimated 1 s
Ran all test suites.
```

Concluzie: Cu ajutorul AI-ului, tool-ul folosind de noi fiind ChatGPT, am putut genera rapid teste suficiente pentru a obține o acoperire de 100%. Acestea au reușit să verifice corect comportamentul general al aplicației pentru cazuri obișnuite, cum ar fi validarea datelor sau gestionarea erorilor.

Totuși, testele pe care le-am scris manual au fost mai detaliate și au acoperit mai multe tipuri de situații, inclusiv cazuri limită și greșeli mai subtile. În plus, am reușit să verificăm și cum se comportă fiecare ramură și condiție din cod, ceea ce duce la o testare mai completă și de încredere.