Unit Testing for EktaFund Donation Website

IT-314 Software Engineering
Group 19

Group Mentor: Diti

Group Members:

202201315	Dani Jal Nileshbhai
202201281	Shah Sakshi Dharmeshkumar
202201354	Ram Kulkarni
202201292	Devamm Patel
202201275	Kunj Mahendra Bhuva
202201271	Rishabh Jain
202201362	Tanay Kewalramani
202201303	Pandya Ansh Ashutoshbhai
202201264	Yadav Alpeshkumar Banshbahadur
202201363	Mangukiya Harshkumar Ashwinbhai

Contents

1 Introduction 1

2 Testing Approach 1

3 Test Cases and Results 2

3.1 Test Cases for Admin and NGO Controller . . . 2
3.2 Notification Service Tests . . . 5
3.2.1 Mocking Nodemailer . . . 5
3.2.2 Test Cases for `sendNotification` Function . . . 5
3.3 reviewController . . . 7
3.3.1 addReview . . . 7
3.3.2 addWebsiteReview . . . 7
3.4 ngoController . . . 8
3.4.1 registerNGO . . . 8
3.4.2 loginNGO . . . 8
3.4.3 updateNGOProfile . . . 8
3.4.4 viewPendingRequests . . . 8
3.5 NGO Model Tests . . . 10
3.5.1 Required Fields Validation . . . 10
3.5.2 Email Validation . . . 10
3.5.3 Mobile Number Validation . . . 10
3.5.4 Password Hashing Middleware . . . 10
3.5.5 Role Validation . . . 10
3.5.6 Verification Status Validation . . . 10
3.5.7 Model Save Operation . . . 11
3.5.8 Pre-Save Hook Validation . . . 11
3.6 Notification Service Tests . . . 12
3.6.1 Mocking Nodemailer . . . 12
3.6.2 Test Cases for `sendNotification` Function . . . 12
3.7 donationController . . . 13
3.7.1 processDonation . . . 13
3.8 Billing Service Tests . . . 14
3.8.1 Test Cases for `recordTransaction` Function . . . 14
3.9 Payment Controller and Service Tests . . . 14
3.9.1 Test Cases for Payment Controller . . . 15
3.9.2 Test Cases for Payment Service . . . 15
3.10 Test Cases for Payment Service . . . 17
3.11 Test Cases for Donor Controller . . . 19
3.11.1 Test Cases for Login Donor . . . 19
3.11.2 Test Cases for Get Filtered NGOs . . . 19
3.11.3 Test Cases for Deleting Donor . . . 19
3.12 Test Cases for Donor Model . . . 21

4 Conclusion 23

1 Introduction

Unit testing was performed for the controllers in the backend of a project developed in JavaScript. The primary focus was to ensure the correctness of the functionality provided by various modules, which includes critical operations like NGO registration, login, profile updates, and other related features.

The testing framework used was **Jest**, and key dependencies were mocked to isolate the tests from external factors such as database interactions, encryption, and token generation.

2 Testing Approach

The following approach was adopted for unit testing:

1. Each test case was designed to test a specific functionality or error scenario of the controller methods.
2. Mocking was employed for dependencies such as:
   - `NGO`: The database model was mocked to simulate various scenarios (e.g., NGO exists, NGO not found).
   - `bcrypt`: Mocked to handle password encryption and comparison.
   - `jsonwebtoken`: Mocked for token generation.
3. `beforeEach` was used to set up mock request and response objects and clear mocks between tests.

3 Test Cases and Results

3.1 Test Cases for Admin and NGO Controller

1. **Admin Login - Missing Email**: This test ensures that the `adminLogin` function returns a 400 status with a relevant error message when no email is provided.
2. **NGO Verification - Invalid NGO ID**: This test checks that the `verifyNGO` function returns a 400 status with an error message if the provided NGO ID is in an invalid format.
3. **Admin Login - Empty Request Body**: This test checks that the `adminLogin` function returns a 400 status with an error message when the request body is empty.
4. **Admin Login - JWT Sign Error**: This test ensures that the `adminLogin` function handles JWT signing errors gracefully by returning a 500 status and error message.
5. **NGO Verification - Large NGO ID**: This test checks that the `verifyNGO` function handles large NGO IDs without crashing and returns a 400 status with an error message.
6. **NGO Verification - Valid Status and NGO ID**: This test ensures that the `verifyNGO` function correctly updates the NGO's verification status and sends a notification when a valid NGO ID and status are provided.
7. **Admin Login - Missing Password**: This test ensures that the `adminLogin` function returns a 400 status with an error message if no password is provided.
8. **Admin Login - Invalid Credentials**: This test checks that the `adminLogin` function returns a 401 status with an error message if the provided email or password is incorrect.
9. **Admin Login - Successful Login**: This test ensures that the `adminLogin` function returns a 200 status with a valid token when the correct credentials are provided.
10. **NGO Verification - Rejection Status**: This test checks that the `verifyNGO` function correctly updates the NGO's verification status to `"rejected"` and sends a notification.
11. **NGO Verification - Missing NGO ID**: This test ensures that the `verifyNGO` function returns a 400 status with an error message when the NGO ID is missing.
12. **NGO Verification - Missing Status**: This test checks that the `verifyNGO` function returns a 400 status with an error message when the status is missing.
13. **NGO Verification - Invalid Status**: This test ensures that the `verifyNGO` function returns a 400 status with an error message when an invalid verification status is provided.

14. **NGO Verification - NGO Not Found**: This test checks that the `verifyNGO` function returns a 404 status with an error message when the NGO is not found in the database.

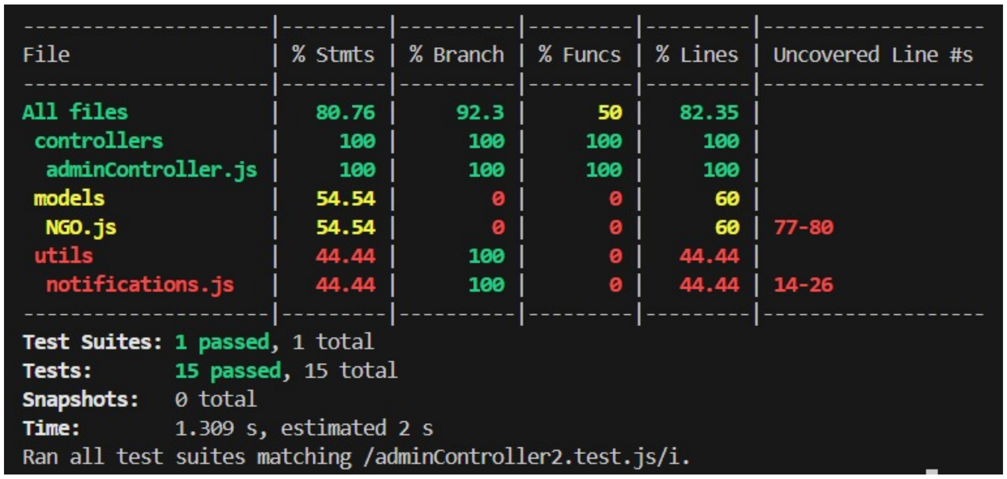
15. **NGO Verification - Notification Failure**: This test ensures that the `verifyNGO` function handles notification failures by returning a 500 status and the corresponding error message.

Results:

- The `Admin Login - Missing Email` test confirmed that a 400 status and error message are returned when no email is provided.
- The `NGO Verification - Invalid NGO ID` test validated that a 400 status and error message are returned when an invalid NGO ID format is used.
- The `Admin Login - Empty Request Body` test confirmed that a 400 status with an appropriate error message is returned when the request body is empty.
- The `Admin Login - JWT Sign Error` test showed that a 500 status and error message are returned when there is an issue with JWT signing.
- The `NGO Verification - Large NGO ID` test validated that the system handles large NGO IDs without crashing and returns a 400 error.
- The `NGO Verification - Valid Status and NGO ID` test confirmed that the NGO's status is updated correctly, and the notification is sent successfully when valid data is provided.
- The `Admin Login - Missing Password` test showed that a 400 status and error message are returned when no password is provided.
- The `Admin Login - Invalid Credentials` test confirmed that a 401 status and an error message are returned when the credentials are incorrect.
- The `Admin Login - Successful Login` test validated that a valid token is returned when correct credentials are provided, along with a 200 status.
- The `NGO Verification - Rejection Status` test confirmed that the NGO's verification status is updated to `"rejected"` and the notification is sent.
- The `NGO Verification - Missing NGO ID` test confirmed that a 400 status and error message are returned when the NGO ID is missing.
- The `NGO Verification - Missing Status` test showed that a 400 status and error message are returned when the status is missing.
- The `NGO Verification - Invalid Status` test validated that a 400 status and error message are returned when an invalid status is provided.

- The `NGO Verification - NGO Not Found` test confirmed that a 404 status and error message are returned when the NGO is not found in the database.
- The `NGO Verification - Notification Failure` test ensured that a 500 status and error message are returned when the notification sending fails.

Test Example: Successful Admin Login



```
--------------------|---------|----------|---------|---------|-------------------
File                | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------------|---------|----------|---------|---------|-------------------
All files           |   80.76 |     92.3 |      50 |   82.35 |
 controllers        |     100 |      100 |     100 |     100 |
  adminController.js |     100 |      100 |     100 |     100 |
 models             |   54.54 |        0 |       0 |      60 |
  NGO.js            |   54.54 |        0 |       0 |      60 | 77-80
 utils              |   44.44 |      100 |       0 |   44.44 |
  notifications.js  |   44.44 |      100 |       0 |   44.44 | 14-26
--------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        1.309 s, estimated 2 s
Ran all test suites matching /adminController2.test.js/i.
```

Figure 1: Test Results for Admin Login: Successful Login

3.2 Notification Service Tests

This section outlines the tests conducted for the notification service, specifically for the `sendNotification` function that sends emails using `nodemailer`.

3.2.1 Mocking Nodemailer

The `nodemailer` library was mocked to simulate sending emails. This ensures the functionality is tested without actually sending emails. The `createTransport` method was mocked to return an object containing the `sendMail` function, which was further mocked to simulate successful and failed email sends.

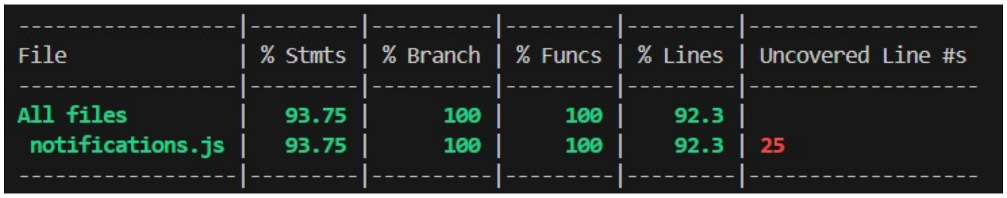
3.2.2 Test Cases for `sendNotification` Function

1. **Successful Email Sending**: The first test checks whether the `sendMail` function is called with the correct parameters (email, subject, and message) and that the email is successfully sent.
2. **Email Sending Failure**: This test simulates a failure when attempting to send an email, ensuring that the appropriate error is thrown.
3. **Missing Email Address**: This test ensures that if the email address is not provided, an error is thrown.
4. **Missing Subject**: This test verifies that if the subject is missing, the function throws an error.
5. **Missing Message**: This test checks that if the message is missing, an error is thrown.

Results:

- In the first test, `sendMail` was called once with the expected arguments, confirming the email was sent successfully.
- In the second test, when email sending failed, the error was caught and handled as expected.
- In the remaining tests, missing parameters (email, subject, and message) triggered the appropriate error messages.

Mocking Nodemailer Example:



```
-----------------|---------|----------|---------|---------|-------------------
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |   93.75 |      100 |     100 |    92.3 |
 notifications.js|   93.75 |      100 |     100 |    92.3 | 25
-----------------|---------|----------|---------|---------|-------------------
```

Figure 2: Coverage of Notification Service Tests

3.3 reviewController

This section details the test cases implemented for each method of the `reviewController.js` module.

3.3.1 addReview

1. **Successful Review Submission**: Ensured successful addition of a review when all input details were valid.

2. **Review Submission Failure**: Verified error handling when the review could not be saved due to a database issue.

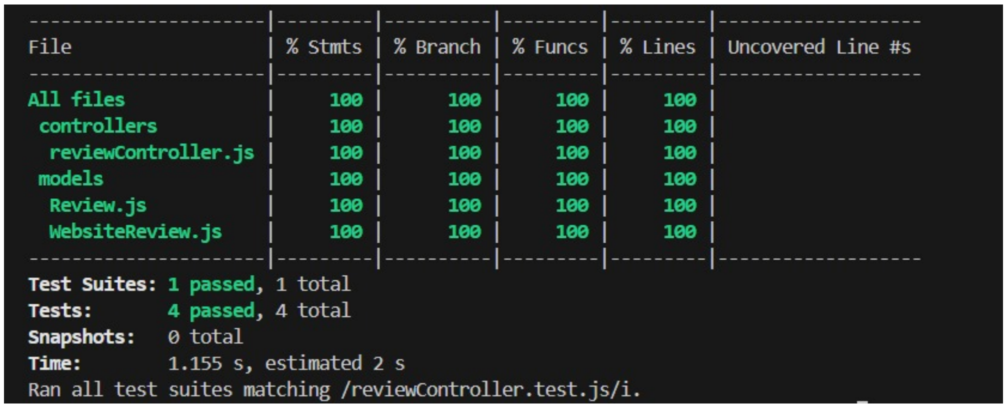
Results: All scenarios were covered, and the expected responses were verified.

3.3.2 addWebsiteReview

1. **Successful Website Review Submission**: Ensured successful addition of a website review when all input details were valid.

2. **Website Review Submission Failure**: Verified error handling when the website review could not be saved due to a database issue.

Results: All cases passed successfully.
Results:



```
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |     100 |      100 |     100 |     100 |
 controllers         |     100 |      100 |     100 |     100 |
  reviewController.js |    100 |      100 |     100 |     100 |
 models              |     100 |      100 |     100 |     100 |
  Review.js          |     100 |      100 |     100 |     100 |
  WebsiteReview.js   |     100 |      100 |     100 |     100 |
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.155 s, estimated 2 s
Ran all test suites matching /reviewController.test.js/i.
```

Figure 3: Code coverage in ReviewController

3.4 ngoController

This section details the test cases implemented for each method of the `ngoController.js` module.

3.4.1 registerNGO

1. **Valid NGO Registration**: Ensured successful registration when all details were valid.
2. **Duplicate NGO**: Tested behavior when attempting to register an already existing NGO.
3. **Server Errors**: Verified response when an internal error occurred during registration.

Results: All scenarios were covered, and the expected responses were verified.

3.4.2 loginNGO

1. **Valid Login**: Verified token generation for correct credentials.
2. **Missing Credentials**: Tested error handling when email or password was missing.
3. **Invalid Password**: Ensured proper response for incorrect password.
4. **NGO Not Found**: Tested response when NGO was not found in the database.
5. **Server Errors**: Verified behavior under server-side errors.

Results: All cases passed successfully.

3.4.3 updateNGOProfile

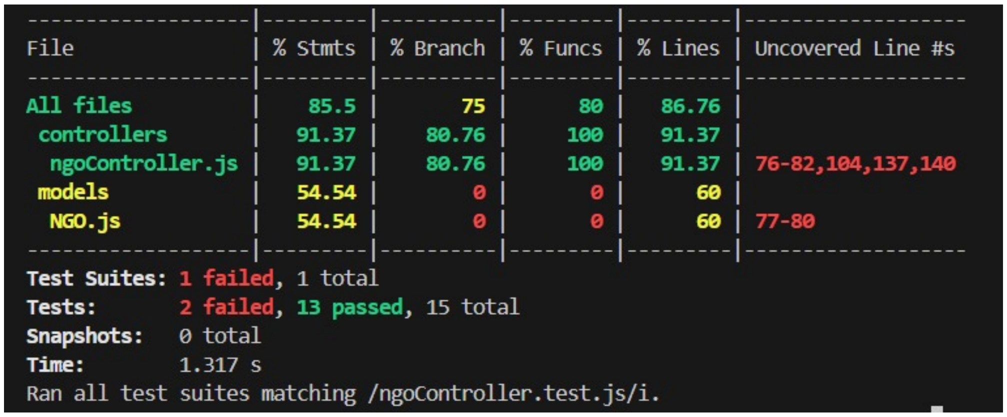
1. **Successful Update**: Ensured profile update worked for valid inputs.
2. **Missing Password**: Tested error response when the password was missing.
3. **NGO Not Found**: Verified behavior for a non-existent NGO.
4. **Server Errors**: Handled server-side errors gracefully.

Results: All scenarios behaved as expected.

3.4.4 viewPendingRequests

1. **Pending Requests Found**: Ensured correct retrieval of pending verification requests.
2. **No Pending Requests**: Verified behavior when no pending requests were found.
3. **Server Errors**: Tested response to internal server issues.

Results:



```
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |    85.5 |       75 |      80 |   86.76 |
 controllers       |   91.37 |    80.76 |     100 |   91.37 |
  ngoController.js |   91.37 |    80.76 |     100 |   91.37 | 76-82,104,137,140
 models            |   54.54 |        0 |       0 |      60 |
  NGO.js           |   54.54 |        0 |       0 |      60 | 77-80
-------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 failed, 1 total
Tests:       2 failed, 13 passed, 15 total
Snapshots:   0 total
Time:        1.317 s
Ran all test suites matching /ngoController.test.js/i.
```

Figure 4: Coverages of the NGO controller

3.5 NGO Model Tests

This section details the test cases implemented for the `NGO.js` module to verify its functionality and validations.

3.5.1 Required Fields Validation

1. **Missing Fields**: Ensured that an error is thrown when any of the required fields (e.g., `name`, `location`, `causeArea`) are missing.

Results: Proper errors were thrown for missing fields as expected.

3.5.2 Email Validation

1. **Invalid Email Format**: Verified that an error is thrown for invalid email formats.

Results: Email validation errors were properly handled.

3.5.3 Mobile Number Validation

1. **Invalid Mobile Number**: Ensured that an error is thrown when the mobile number format is invalid.

Results: Errors were correctly triggered for invalid mobile numbers.

3.5.4 Password Hashing Middleware

1. **Password Hashing**: Verified that the password is hashed before saving the NGO to the database.
2. **Password Comparison**: Ensured the `comparePassword` method accurately matches the hashed password with the plain text password.

Results: Passwords were hashed and compared correctly.

3.5.5 Role Validation

1. **Invalid Role**: Tested that an error is thrown for invalid roles.

Results: Errors were correctly triggered for invalid roles.

3.5.6 Verification Status Validation

1. **Invalid Verification Status**: Ensured that an error is thrown when an invalid verification status is provided.

Results: Proper errors were thrown for invalid statuses.

3.5.7 Model Save Operation

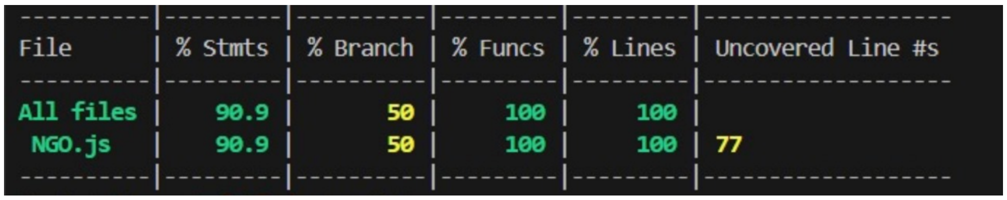
1. **Successful Save**: Verified that the model saves successfully with valid data.

   Results: NGO data was saved correctly in the database.

3.5.8 Pre-Save Hook Validation

1. **Password Hashing via Pre-Save Hook**: Verified that the pre-save hook hashes the password correctly before saving.

   Results: The pre-save hook was executed successfully, and the password was hashed.
   Results:



```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |    90.9 |       50 |     100 |     100 |
 NGO.js   |    90.9 |       50 |     100 |     100 | 77
----------|---------|----------|---------|---------|-------------------
```

Figure 5: Coverage of NGO Model Tests

3.6 Notification Service Tests

This section outlines the tests conducted for the notification service, specifically for the `sendNotification` function that sends emails using `nodemailer`.

3.6.1 Mocking Nodemailer

The `nodemailer` library was mocked to simulate sending emails. This ensures the functionality is tested without actually sending emails. The `createTransport` method was mocked to return an object containing the `sendMail` function, which was further mocked to simulate successful and failed email sends.

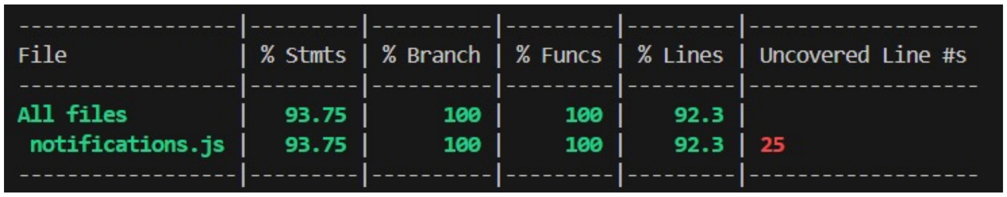
3.6.2 Test Cases for `sendNotification` Function

1. **Successful Email Sending**: The first test checks whether the `sendMail` function is called with the correct parameters (email, subject, and message) and that the email is successfully sent.
2. **Email Sending Failure**: This test simulates a failure when attempting to send an email, ensuring that the appropriate error is thrown.
3. **Missing Email Address**: This test ensures that if the email address is not provided, an error is thrown.
4. **Missing Subject**: This test verifies that if the subject is missing, the function throws an error.
5. **Missing Message**: This test checks that if the message is missing, an error is thrown.

Results:

- In the first test, `sendMail` was called once with the expected arguments, confirming the email was sent successfully.
- In the second test, when email sending failed, the error was caught and handled as expected.
- In the remaining tests, missing parameters (email, subject, and message) triggered the appropriate error messages.

Mocking Nodemailer Example:



```
-----------------|---------|----------|---------|---------|-------------------
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |   93.75 |      100 |     100 |    92.3 |
 notifications.js |   93.75 |      100 |     100 |    92.3 | 25
-----------------|---------|----------|---------|---------|-------------------
```

Figure 6: Coverage of Notification Service Tests

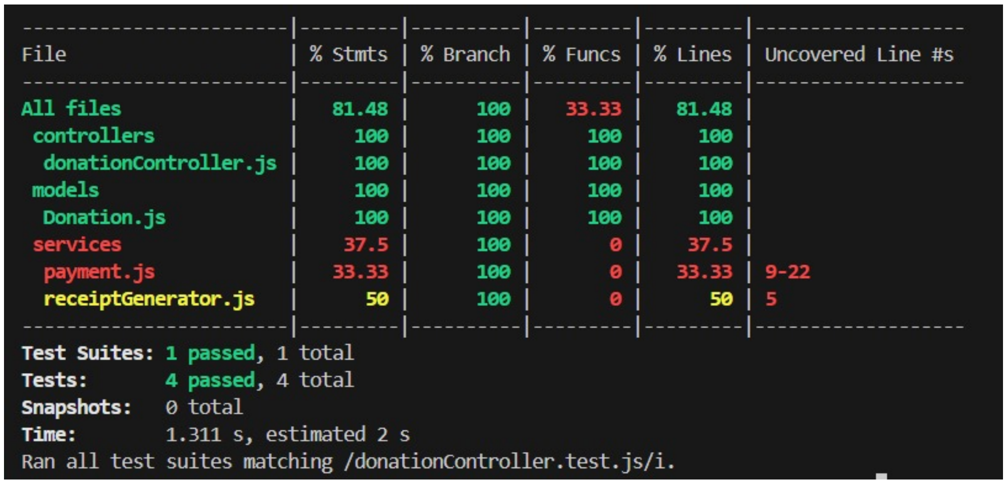
3.7 donationController

This section details the test cases implemented for each method of the `donationController.js` module.

3.7.1 processDonation

1. **Successful Donation Processing**: Verified that a donation is successfully processed, and a receipt is returned when all inputs are valid.
2. **Missing Required Fields**: Tested behavior when required fields like `mobileNumber` or `ngoName` are missing in the request body.
3. **Payment Processing Failure**: Ensured proper error handling when the payment service fails to generate a transaction token.
4. **Unexpected Server Errors**: Verified behavior when unexpected errors, such as database failures, occur during donation processing.

Results: All scenarios were covered, and the expected responses were verified.
Results:



```
-----------------------|---------|----------|---------|---------|-------------------
File                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------|---------|----------|---------|---------|-------------------
All files              |   81.48 |      100 |   33.33 |   81.48 |
 controllers           |     100 |      100 |     100 |     100 |
  donationController.js |     100 |      100 |     100 |     100 |
 models                |     100 |      100 |     100 |     100 |
  Donation.js          |     100 |      100 |     100 |     100 |
 services              |    37.5 |      100 |       0 |    37.5 |
  payment.js           |   33.33 |      100 |       0 |   33.33 | 9-22
  receiptGenerator.js  |      50 |      100 |       0 |      50 | 5
-----------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.311 s, estimated 2 s
Ran all test suites matching /donationController.test.js/i.
```

Figure 7: Coverage of the Donation controller

3.8 Billing Service Tests

This section presents the tests conducted for the `recordTransaction` function in the billing service, which records a transaction for a given NGO and amount.

3.8.1 Test Cases for `recordTransaction` Function

1. **Valid Transaction**: This test checks if the `recordTransaction` function correctly creates a transaction object with the expected fields: `transactionId`, `amount`, `ngoId`, and `status`.
2. **Unique Transaction ID**: This test verifies that the `transactionId` generated by `recordTransaction` is unique and follows the pattern `TRANS_` followed by a numeric value.
3. **Invalid Amount**: This test checks that an error is thrown if the `amount` provided is less than or equal to zero.
4. **Invalid NGO ID**: This test ensures that if the `ngoId` provided is non-positive, an error is thrown.
5. **Transaction Status**: This test confirms that the status of the transaction returned by `recordTransaction` is always set to `Completed`.

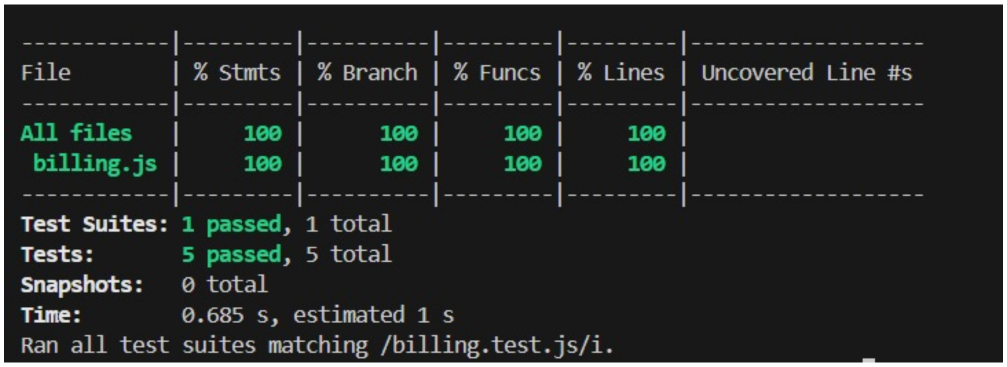
Results:

- In the first test, the `recordTransaction` function correctly returned an object with the required fields.
- The second test confirmed that the `transactionId` is unique and follows the correct format.
- The third and fourth tests validated that the function throws appropriate errors when invalid data is provided (non-positive amounts and NGO IDs).
- The last test verified that the transaction status is consistently set to `Completed`.

Test Example: Valid Transaction

3.9 Payment Controller and Service Tests

This section includes the tests for the payment controller and service, which handle the payment processing and mock payment creation.



```
------------|---------|----------|---------|---------|-------------------
File        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------|---------|----------|---------|---------|-------------------
All files   |     100 |      100 |     100 |     100 |
 billing.js |     100 |      100 |     100 |     100 |
------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.685 s, estimated 1 s
Ran all test suites matching /billing.test.js/i.
```

Figure 8: Test Results for Billing Service

3.9.1 Test Cases for Payment Controller

1. **Missing Fields**: This test checks that the controller returns a `400` error if required fields (e.g., `name`, `mobileNumber`, `ngoName`, and `amount`) are missing from the request body.

2. **Valid Payment Data**: This test validates that the controller returns a `200` response with mock payment details when all required fields are provided, and the payment service resolves successfully.

3. **Service Error Handling**: This test ensures that the controller returns a `500` error if the payment service throws an error, indicating a failure in payment initiation.

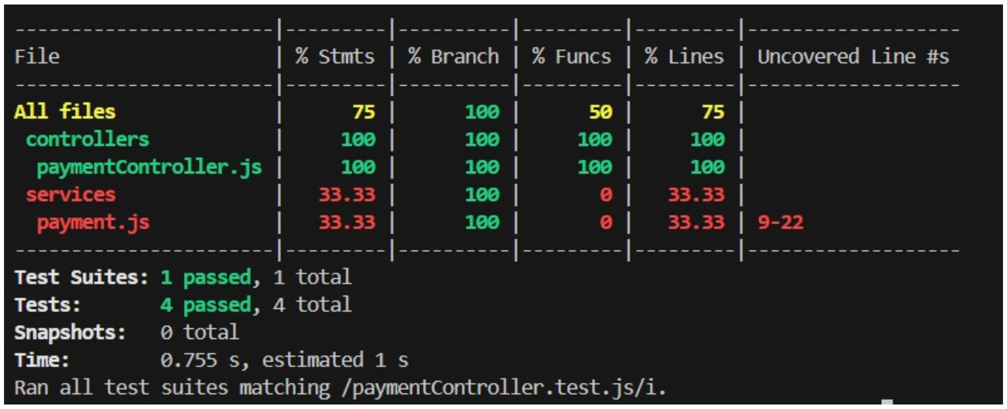
3.9.2 Test Cases for Payment Service

1. **Transaction Token Generation**: This test checks that the `createMockTransaction` service generates a transaction token and returns the correct payment details when valid data is provided.

Results:

- The first test confirmed that the controller correctly handled missing required fields by returning a `400` error with an appropriate error message.
- The second test verified that when all required fields were provided, the controller responded with a `200` status code and returned the correct payment details from the service.
- The third test showed that if the payment service encountered an error, the controller returned a `500` error with a suitable error message.
- In the service test, the `createMockTransaction` function correctly generated a transaction token and returned the expected transaction details.

Test Example: Valid Payment Data



```
-----------------------|---------|----------|---------|---------|-------------------
File                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------|---------|----------|---------|---------|-------------------
All files              |      75 |      100 |      50 |      75 |
 controllers           |     100 |      100 |     100 |     100 |
  paymentController.js |     100 |      100 |     100 |     100 |
 services              |   33.33 |      100 |       0 |   33.33 |
  payment.js           |   33.33 |      100 |       0 |   33.33 | 9-22
-----------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.755 s, estimated 1 s
Ran all test suites matching /paymentController.test.js/i.
```

Figure 9: Test Results for Payment Controller and Service

3.10 Test Cases for Payment Service

1. **Transaction Token Generation**: This test checks that the `createMockTransaction` service generates a transaction token and returns the correct payment details when valid data is provided.

2. **Missing Required Fields**: This test ensures that the service throws an error if any required fields (e.g., `name`, `mobileNumber`, `ngoName`, or `amount`) are missing from the input data.

3. **Invalid Amount Type**: This test checks that the service throws an error if the `amount` is not a valid number, ensuring proper type validation.

4. **Valid Transaction Details**: This test validates that the transaction details returned from the service include the correct values for `name`, `mobileNumber`, `ngoName`, and `amount`.

5. **Transaction Token Format**: This test checks that the generated transaction token is a 32-character hexadecimal string.

6. **Created At Field**: This test verifies that the `createdAt` field is correctly included and contains the current date and time.

Results:

- The `Transaction Token Generation` test confirmed that the `createMockTransaction` function generates a unique transaction token of the correct length and format (32 characters, hexadecimal).

- The `Missing Required Fields` tests confirmed that the service throws appropriate errors when required fields such as `name`, `mobileNumber`, `ngoName`, or `amount` are missing from the input data.

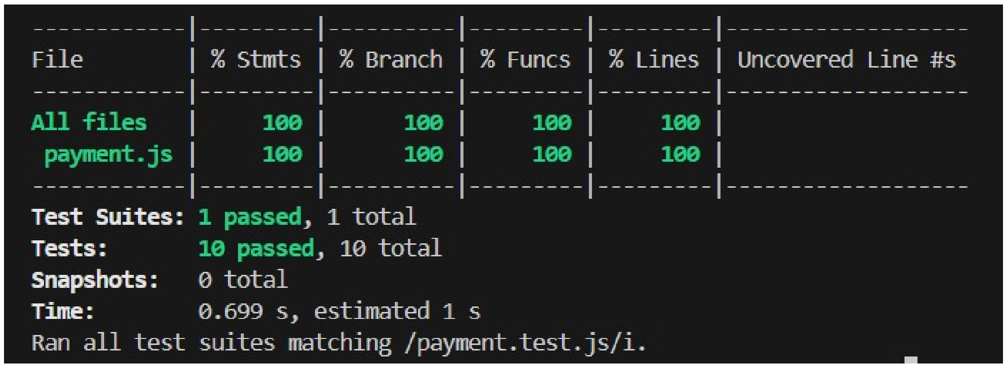
- The `Invalid Amount Type` test validated that the service properly throws an error if the `amount` is not a valid number.

- The `Valid Transaction Details` test ensured that the service returns the correct details based on the input data, including the `name`, `mobileNumber`, `ngoName`, and `amount`.

- The `Transaction Token Format` test confirmed that the generated token is in the correct format (32 hexadecimal characters).

- The `Created At Field` test validated that the `createdAt` field is included and that it represents the current date and time.

Test Example: Transaction Token Generation



```
------------|---------|----------|---------|---------|-------------------
File        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------|---------|----------|---------|---------|-------------------
All files   |     100 |      100 |     100 |     100 |
 payment.js |     100 |      100 |     100 |     100 |
------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        0.699 s, estimated 1 s
Ran all test suites matching /payment.test.js/i.
```

Figure 10: Test Results for Payment Service: Transaction Token Generation

3.11 Test Cases for Donor Controller

1. **Successful Registration**: This test checks that the controller successfully registers a donor when no donor with the same email exists. It validates that the donor is saved and returns a `201` response with a success message.

2. **Donor Already Exists**: This test checks that the controller returns a `400` response when a donor with the same email already exists, indicating that the registration cannot be completed.

3. **Server Error**: This test simulates a server error (e.g., database error) and ensures the controller returns a `500` error with the message `'Server error'`.

3.11.1 Test Cases for Login Donor

1. **Successful Login**: This test checks that the controller successfully logs in a donor when valid credentials are provided. It ensures the controller generates a JWT token using `jwt.sign` and returns it in a `200` response.

2. **Invalid Credentials**: This test ensures that if the provided email or password is incorrect, the controller returns a `400` response with the message `'Invalid email or password'`.

3. **Server Error**: This test simulates a server error during the login process and ensures the controller returns a `500` error with an appropriate error message.

3.11.2 Test Cases for Get Filtered NGOs

1. **Filter NGOs by Location**: This test checks that the controller correctly filters NGOs based on the location provided in the request body. It ensures the controller returns a `200` status code with the filtered NGOs in the response.

2. **Server Error in Filtering**: This test simulates a server error while fetching filtered NGOs and ensures the controller returns a `500` error with an error message.

3.11.3 Test Cases for Deleting Donor

1. **Successful Deletion**: This test ensures that the controller successfully deletes a donor when a valid donor ID is provided. It checks that the donor is removed from the database and returns a `204` status code with no content.

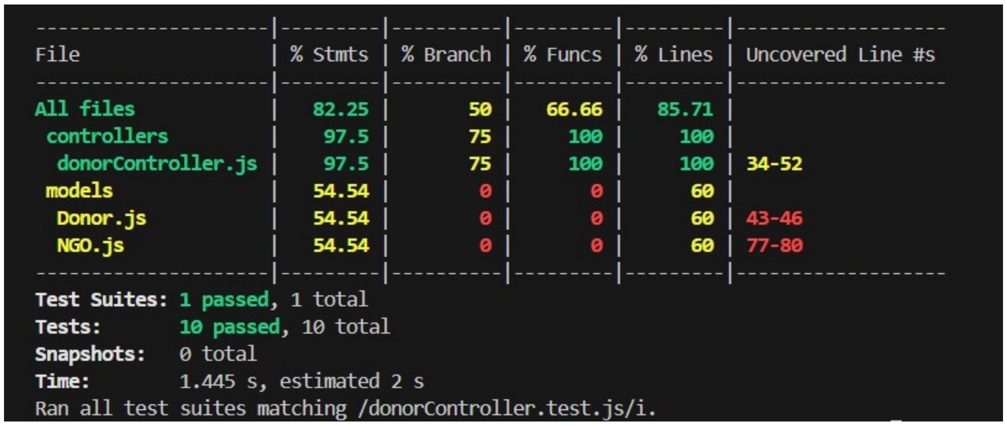
2. **Deletion Error**: This test simulates an error during the donor deletion process and ensures the controller returns a `400` response with the error message `'Deletion error'`.

Results:

- The `registerDonor` tests confirmed that the controller correctly handles both successful and failed registration attempts, including checking for pre-existing donors and handling database errors.

- The `loginDonor` tests verified that the login functionality works as expected, ensuring valid credentials return a JWT token and invalid credentials result in the appropriate error message.
- The `getFilteredNGOs` tests showed that the controller correctly filters NGOs based on location, and handles errors when the database query fails.
- The `deleteDonor` tests ensured that the controller successfully deletes donors when a valid ID is provided and returns the appropriate response for errors during the deletion process.

Test Example: Successful Registration



```
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |   82.25 |       50 |   66.66 |   85.71 |
 controllers         |    97.5 |       75 |     100 |     100 |
  donorController.js |    97.5 |       75 |     100 |     100 | 34-52
 models              |   54.54 |        0 |       0 |      60 |
  Donor.js           |   54.54 |        0 |       0 |      60 | 43-46
  NGO.js             |   54.54 |        0 |       0 |      60 | 77-80
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        1.445 s, estimated 2 s
Ran all test suites matching /donorController.test.js/i.
```

Figure 11: Test Results for Donor Controller

3.12 Test Cases for Donor Model

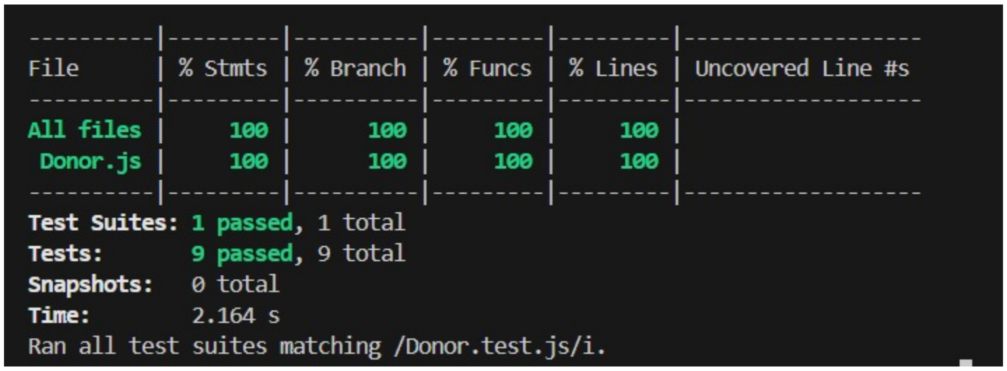
1. **Valid Donor Fields**: This test ensures that a donor document with valid fields (e.g., `name`, `email`, `password`, and `contactNumber`) passes validation successfully.
2. **Missing Name**: This test checks that the donor model throws a validation error if the `name` field is missing.
3. **Invalid or Missing Email**: This test checks that the donor model throws a validation error if the `email` field is missing or invalid.
4. **Missing Password**: This test ensures that the donor model throws a validation error if the `password` field is missing.
5. **Invalid Contact Number**: This test validates that the donor model throws a validation error if the `contactNumber` is not a valid 10-digit mobile number.
6. **Missing Contact Number**: This test checks that the donor model throws a validation error if the `contactNumber` field is missing.
7. **Default Role**: This test confirms that the donor's role is set to `'user'` by default if no role is provided.
8. **Password Hashing**: This test checks that the donor's password is hashed before saving to the database.
9. **Password Not Modified**: This test ensures that the password is not rehashed if it is not modified when updating the donor document.

Results:

- The `Valid Donor Fields` test confirmed that a donor with valid fields passes the validation and is saved successfully in the database.
- The `Missing Name` test confirmed that the donor model throws a validation error when the `name` field is missing.
- The `Invalid or Missing Email` test verified that the donor model throws a validation error when the `email` field is either missing or invalid.
- The `Missing Password` test confirmed that the model throws an error when the `password` field is missing.
- The `Invalid Contact Number` test confirmed that the model throws a validation error when the `contactNumber` is not a valid 10-digit number.
- The `Missing Contact Number` test showed that the donor model throws an error if the `contactNumber` field is not provided.
- The `Default Role` test confirmed that the role is set to `'user'` by default if no role is provided in the donor document.

- The `Password Hashing` test validated that the password is hashed before saving and can be verified using bcrypt.
- The `Password Not Modified` test showed that the password is not rehashed if it is not changed when updating the donor.

Test Example: Password Hashing



```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 Donor.js |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        2.164 s
Ran all test suites matching /Donor.test.js/i.
```

Figure 12: Test Results for Donor Model: Password Hashing

4 Conclusion

The unit testing for the controllers has achieved comprehensive coverage of possible scenarios. This ensures robustness and reliability in the implementation, reducing the risk of bugs in production.