IT314 – Software Engineering

G27 Flight Booking System

Unit Testing Report

Document Purpose: This document covers the unit testing done on our code and summarizes on some of the important test cases. We have achieved 100% code coverage of all our backend files and around 86% coverage of some of the frontend files.

Technologies used

jest: ^29.7.0

mongodb-memory-server: ^10.1.2

Backend Unit Testing

We have made 10 test suites to cover all the files in the backend. A total of 68 testcases are there but as a summary, only few testcases from each file have been included in this document to provide a summary.

```
beforeAll(async () => {
    // Start the in-memory MongoDB server
    mongoServer = await MongoMemoryServer.create(
        { binary: {
            version: '5.0.0', // Match the version downloaded
            downloadDir: './mongodb-binaries' // Path to cached binar
        }}
    );
    const uri = mongoServer.getUri();

    // Connect to the in-memory database
    await mongoose.connect(uri, {
        useNewUrlParser: true,
        useUnifiedTopology: true,
    });
});

afterEach(async () => {
    // Clean up the database after each test
if (user) {
    await UserModel.deleteOne({ _id: user });
}
// Clean up any bookings related to the user
if (user) {
    await BookingModel.deleteMany({ userId: user });
}
});

afterAll(async () => {
    // Disconnect and stop the in-memory server
    await mongoose.disconnect();
    await mongoServer.stop();
});
```

A general configuration which is included in all test suites is as below:

Before all: Start mongo-memory-server and establish initial mongoose connect.

After Each: Delete the model entries that were inserted in the test case.

After all: Close mongoose connection and stop mongoserver.

Test cases for test suite AuthController.test.js

```
describe('signup', () => {
    it('should return 409 if user already exists', async () => {
        // Create a user directly in the database
        await UserModel.create({ name:'Test Data',email: 'test@example.com', password: 'hashedPassword' });

        const req = mockRequest({name:'Test Data', email: 'test@example.com', password: 'password' });
        const res = mockResponse();

        await signup(req, res);

        expect(res.status).toHaveBeenCalledWith(409);
        expect(res.json).toHaveBeenCalledWith({
            message: 'User already exist, you can login',
            success: false,
        });
    });
```

```
it('should return 200 and a JWT if login is successful', async () => {
    // Create a user in the database
    const user = await UserModel.create({
        email: 'test@example.com',
        password: await bcrypt.hash('password', 10),
        name: 'Test User',
    });

    const req = mockRequest({ email: 'test@example.com', password: 'password' });
    const res = mockResponse();

    await login(req, res);
    const token = res.json.mock.calls[0][0].jwtToken; // Extract the token.
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith({
        message: 'Login Success',
        success: true,
        jwtToken: token, // Check JWT exists
        email: user.email,
        name: user.name,
        userId: user._id,
    });
    const decoded = jwt.verify(token, process.env.JWT_SECRET); // Replace with your JWT_SECRET

    // Validate the payload
    expect(decoded.email).toBe('test@example.com');
    expect(decoded._id).toBeDefined(); // Ensure the _id field exists
    expect(decoded).toHaveProperty('exp'); // Check the expiration field

});
```

This test suite is for testing the signup and login functionalities. User exists or not, login and signup are returning desired status codes etc. Here, in the test cases in the screen shots, we are testing if an existing user with the same email address exists or not and verifying successful login condition.

Test cases from test suite Authvalidation.test.js

```
describe("Signup Validation", () => {
  it("should return 400 if 'name' is missing", async () => {
    const response = await supertest(app).post("/auth/signup").send({
      email: "test1@example.com",
      password: "password123",
    });
    expect(response.status).toBe(400);
    expect(response.body.message).toBe("Bad request");
    expect(response.body.error.details[0].message).toContain('"name" is required');
  });

  it("should return 400 if 'email' is invalid", async () => {
    const response = await supertest(app).post("/auth/signup").send({
      name: "John Doe",
      email: "not-an-email",
      password: "password123",
    });

    expect(response.status).toBe(400);
    expect(response.body.message).toBe("Bad request");
    expect(response.body.error.details[0].message).toContain('"email" must be a valid email');
  });
```

The Authvalidation test suite tests for the fields user enters during signup and login correct and complete or not. So it tests the validity of password entered function, exception on field empty function etc. Here the test case in the screen shot is testing the condition of name missing during signup and email invalid during signup.

Test cases from test suite BookingController.test.js

```
describe('Booking (Signup)', () => {
    it('should return 409 if user already exists', async () => {
        // Create a user in the database before testing
        const existingUser = new UserModel({
            name: 'Test User',
            email: 'test@example.com',
            password: await bcrypt.hash('password', 10),
        });
        await existingUser.save();

        const req = mockRequest({
            name: 'New User',
            email: 'test@example.com',
            password: 'password',
        });
        const res = mockResponse();

        // Call the Booking function
        await Booking(req, res);

        expect(res.status).toHaveBeenCalledWith(409);
        expect(res.json).toHaveBeenCalledWith({
            message: 'User already exist, you can login',
            success: false,
        });
    });
```

This test suite tests mainly the prerequisites for the functionality of booking that is a user exists or not or if a user needs to signup after searching the flights in order to book them. The above test case is for the existing user trying to signup again before booking a flight.

Test cases from test suite BookingRouter.test.js

```
// Test Case 5: Return 500 if there's an internal server error
it('should return 500 if there is a server error', async () => {
    // Disconnect the database to simulate a server error
    await mongoose.disconnect();

    const bookingData = {
        userId,
        flightId,
        from: 'NYC',
        to: 'LAX',
        date: '2023-06-23',
        class: 'economy',
        passengers: [{ name: 'John Doe' }],
        status: 'Confirmed',
        price: 6013,
        paymentMethod: 'card',
        addons: [],
    };

    const response = await request(app)
        .post('/bookings/create')
        .send(bookingData);

    expect(response.status).toBe(500);
    expect(response.body.success).toBe(false);
    expect(response.body.message).toBe('Error creating booking');

    // Reconnect to continue further tests
```

This test suite tests all the booking functionalities that does the flight exist or if the passengers entered are more than seats available etc. The above test case tests for a server error occurring while booking and tries to test the behaviour by disconnecting the database and the below one tests if enough seats are available or not.

```
// Test Case 4: Return 400 if not enough seats are available
it('should return 400 if there are not enough seats available', async () => {
    const user = await UserModel.create({
        name: 'John Doe',
        email: 'johndoe@example.com',
        password: 'hashedpassword123',
    });
    userId = user._id.toString();
    const flight = await FlightModel.create({
        'flight date': "26-06-2023",
        airline : "SpiceJet",
        flight_num: "SG-9001",
        class : "economy",
        from : "NYC",
        dep_time : "18:55",
        to : "LAX",
        arr_time : "21:05",
        duration : "04h 10m",
        price : "6,013",
        stops : "non-stop",
        availableseats: 100
    });
    flightId = flight._id.toString();

    const bookingData = {
        userId ,
        flightId,
        from: 'NYC',
        to: 'LAX',
        date: '2023-06-26',
        class: 'economy',
        passengers: new Array(101).fill({ name: 'John Doe'}), // 101 passengers, b
        status: 'Confirmed',
        price: 6013,
        paymentMethod: 'card',
        addons: [],
    };
    const response = await request(app)
        .post('/bookings/create')
        .send(bookingData);

    expect(response.status).toBe(400);
    expect(response.body.success).toBe(false);
    expect(response.body.message).toBe('Not enough seats available');
});
```

Test cases from test suite Connection.test.js

```
describe('connectToDatabase', () => {
    it('should throw an error when mongodb connection fails', async () => {
        // Mock an invalid URI to simulate connection failure
        process.env.ATLAS_URI = 'invalid-uri';

        // Mock the MongoClient to throw an error
        const mockConnect = jest.spyOn(MongoClient.prototype, 'connect').mockRejectedValueOnce(new Error('Connection failed'));

        // Spy on console.error
        const consoleSpy = jest.spyOn(console, 'error').mockImplementation(() => {});

        // Expect the function to throw an error
        await expect(connectToDatabase()).rejects.toThrow('Connection failed');

        // Verify that console.error was called with the specific message
        expect(consoleSpy).toHaveBeenCalledWith('Error connecting to MongoDB');

        // Restore spies
        mockConnect.mockRestore();
        consoleSpy.mockRestore();
    });
});
```

This test suite is to test the connection of mongoDB database to the system and if it fails to connect then the errors are thrown correctly or not. The above test case is to check if mongoDB throws an error when trying to connect to an incorrect url.

Test cases from test suite index.test.js

```
describe('Routes Testing', () => {
    describe('Auth Routes (/auth)', () => {
        it('should handle POST /auth/signup', async () => {
            const res = await request(app)
                .post('/auth/signup')
                .send({ name: 'Test User', email: 'test@example.com', password: 'password' });
            expect([200, 400, 409]).toContain(res.status);
        });

        it('should handle POST /auth/login', async () => {
            // Seed a user for login
            await UserModel.create({
                name: 'Test User',
                email: 'test@example.com',
                password: 'hashedpassword123',
            });

            const res = await request(app)
                .post('/auth/login')
                .send({ email: 'test@example.com', password: 'password' });
            expect([200, 400, 401,403]).toContain(res.status);
        });
    });
```

```
describe('Environment Variables', () => {
    it('should load environment variables', () => {
        expect(process.env.PORT).toBeDefined();
        expect(process.env.PORT).toBe('5050');
    });
});
describe('Undefined Routes', () => {
    it('should return 404 for undefined routes', async () => {
        const res = await request(app).get('/undefined-route');
        expect(res.status).toBe(404);
        expect(res.body).toHaveProperty('message', 'Not Found');
    });
});
```

This test suite tests the main file from where almost all the routes initiate. Hence here we test if the initial routes are working or not. Also here we are testing for environment variables as well as shown in the test case above.

Test cases from test suite SearchRouter.test.js

```
it("should return 200 and matching flights when flights exist", async () => {
  // Add test data
  const flight1 = await FlightModel.create({
      'flight date': "2023-06-26",
      airline : "SpiceJet",
      flight_num: "SG-9001",
      class : "economy",
      from : "NYC",
      dep_time : "18:55",
      to : "LAX",
      arr_time : "21:05",
      duration : "04h 10m",
      price : "6,013",
      stops : "non-stop",
      availableseats: 100
  });
  const flight2 = await FlightModel.create({
      'flight date': "2023-06-26",
      airline : "SpiceJet",
      flight_num: "SG-9002",
      class : "economy",
      from : "NYC",
      dep_time : "18:55",
      to : "LAX",
      arr_time : "21:05",
      duration : "04h 10m",
      price : "6,013",
      stops : "non-stop",
      availableseats: 100
  });
  flightId1 = flight1._id.toString();
  flightId2 = flight2._id.toString();
  const response = await supertest(app).get("/flight").query({
    from: "NYC",
    to: "LAX",
    start_date: "2023-06-26",
  });

  expect(response.status).toBe(200);
  expect(response.body.success).toBe(true);
  expect(response.body.flights).toHaveLength(2);
  expect(response.body.flights[0]).toMatchObject({
    from: "NYC",
    to: "LAX",
    "flight date": "2023-06-26",
  });
});
```

This test suite tests for the searching functionality and its functioning. The above test case is for the successful retrieval of flight data from the database.

Test cases from test suite Server.test.js

```
test("Should throw an error if PORT is not defined", () => {
    delete process.env.PORT; // Simulate missing PORT
    expect(() => startServer()).toThrow("Invalid port");
});

test("Should start server successfully on a valid port", (done) => {
    process.env.PORT = "3000"; // Simulate a valid port

    // Start the server manually
    startServer();

    // We need to set a timeout because it takes some time for the server to start
    setTimeout(() => {
        expect(console.log).toHaveBeenCalledWith("Connected to Backend on port 3000");
        done(); // End the test
    }, 100); // Wait for a short time to allow the server to start
});
```

This test suite tests the file where the server is started by checking whether the exceptions when incorrect port number is entered are thrown or not. The above test case is for the same.

Test cases from test suite Subscribe.test.js

```
describe('POST /subscribe', () => {
  it('should return 400 if email is missing', async () => {
    const response = await request(app).post('/api/subscribe').send({});
    expect(response.status).toBe(400);
    expect(response.body).toEqual({
      success: false,
      message: 'Email is required',
    });
  });

  it('should return 400 if email is already subscribed', async () => {
    const email = 'test@example.com';

    // Create a subscriber
    await SubscriberModel.create({ email });

    const response = await request(app).post('/api/subscribe').send({ email });
    expect(response.status).toBe(400);
    expect(response.body).toEqual({
      success: false,
      message: 'Email already subscribed',
    });
  });
```

This test suite tests the subscribe functionality and has tests for whether the functionality is working as expected when the user has already subscribed or if email is missing. The above test case is for the missing email and user already subscribed case.

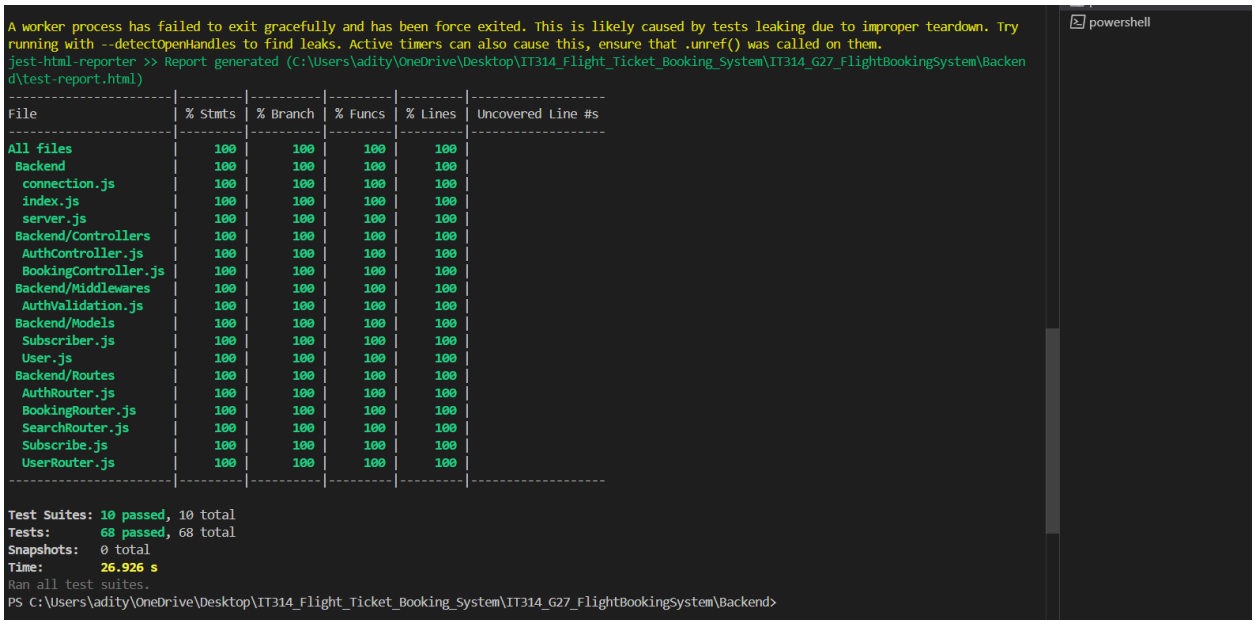
Test cases from test suite UserRouter.test.js

```
describe('PUT /user/:userId/passengers', () => {

  it('should return 200 when passengers are successfully updated', async () => {
    const user = await UserModel.create({
        name: 'Test User',
        email: 'jane@example.com',
        password: 'hashedpassword123',
        passengers: [],
      });
      userId = user._id.toString();
      const passengers = [
        {
            designation: 'Mr',
            firstName: 'John',
            lastName: 'Doe',
            dob: new Date('1990-01-01')
        },
        {
            designation: 'Mrs',
            firstName: 'Jane',
            lastName: 'Doe',
            dob: new Date('1992-01-01')
        }
    ];

    const res = await request(app)
      .put(`/user/${userId}/passengers`)
      .send({ passengers });

    expect(res.status).toBe(200);
    expect(res.body.passengers).toHaveLength(2);
    expect(res.body.passengers[0].firstName).toBe('John');
    expect(res.body.passengers[0].lastName).toBe('Doe');
    expect(res.body.passengers[0].designation).toBe('Mr');
  });
```

```
  it('should return 404 when the user is not found', async () => {
    const nonExistentUserId = new mongoose.Types.ObjectId().toString();

    const res = await request(app)
      .put(`/user/${nonExistentUserId}/passengers`)
      .send({ passengers: [] });

    expect(res.status).toBe(404);
    expect(res.body.error).toBe('User not found');
  });

  it('should return 500 when a database error occurs', async () => {
    jest.spyOn(UserModel, 'findByIdAndUpdate').mockImplementationOnce(() => {
      throw new Error('Database error');
    });

    const res = await request(app)
      .put(`/user/${userId}/passengers`)
      .send({ passengers: [] });

    expect(res.status).toBe(500);
    expect(res.body.error).toBe('Internal Server Error');
  });
});
```

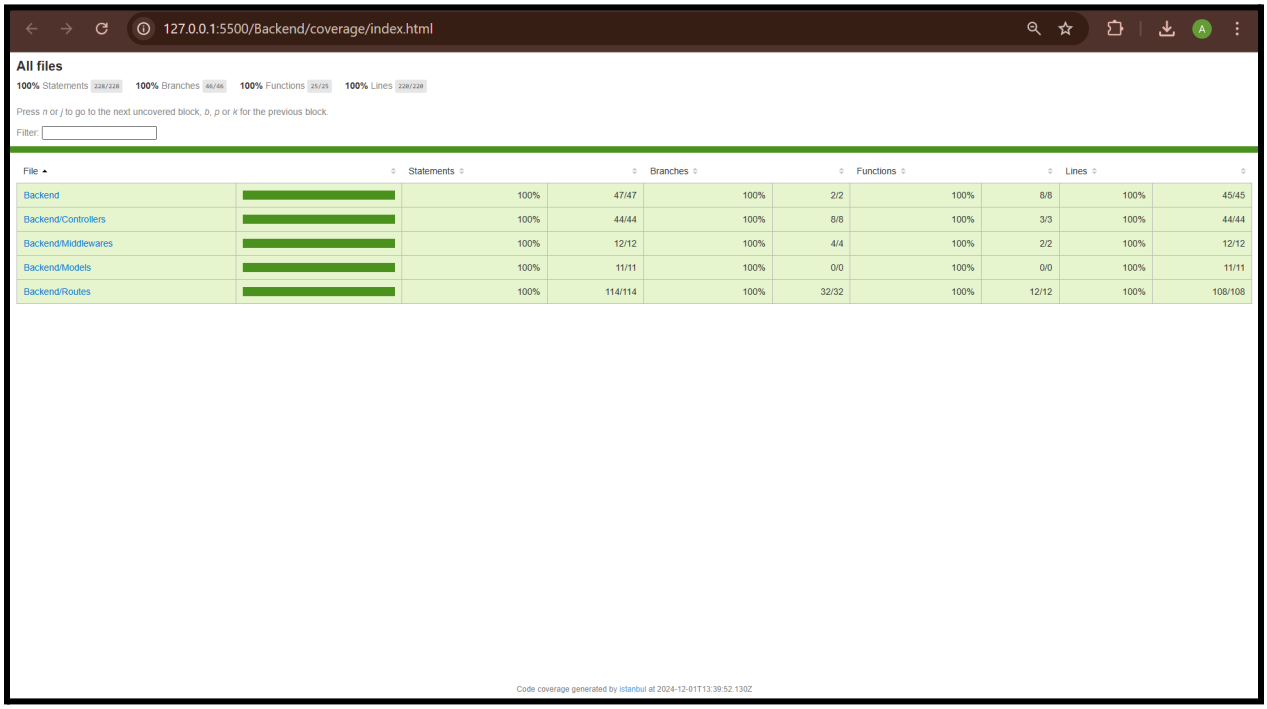
This test suite tests the user functionalities that is if a user exists, if a passenger added by user is added or edited or deleted from user database. The above test cases test the same.

Below is the **coverage report** of all backend files returned in terminal.

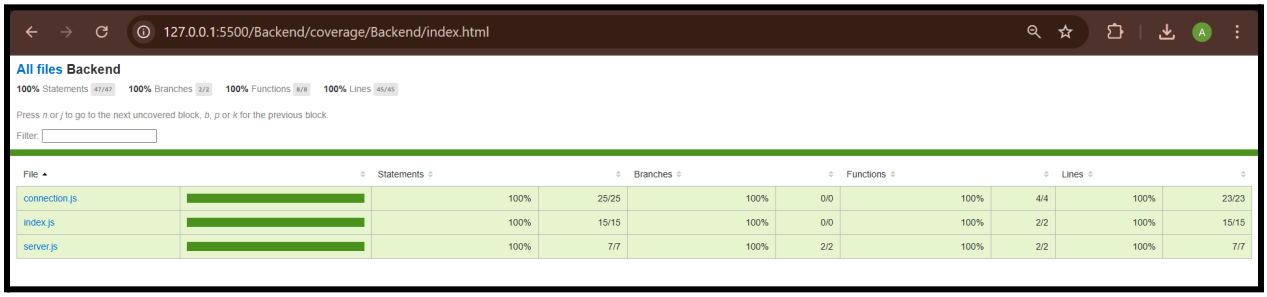


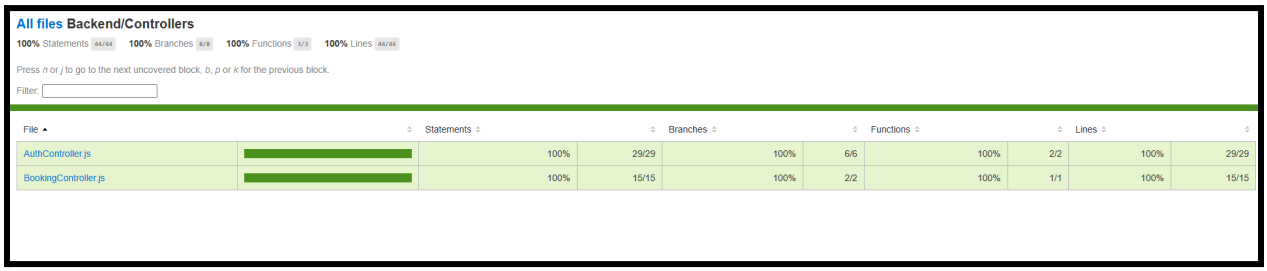
```
A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests leaking due to improper teardown. Try
running with --detectOpenHandles to find leaks. Active timers can also cause this, ensure that .unref() was called on them.
jest-html-reporter >> Report generated (C:\Users\adity\OneDrive\Desktop\IT314_Flight_Ticket_Booking_System\IT314_G27_FlightBookingSystem\Backen
d\test-report.html)
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |     100 |      100 |     100 |     100 |
 Backend              |     100 |      100 |     100 |     100 |
  connection.js       |     100 |      100 |     100 |     100 |
  index.js            |     100 |      100 |     100 |     100 |
  server.js           |     100 |      100 |     100 |     100 |
 Backend/Controllers  |     100 |      100 |     100 |     100 |
  AuthController.js   |     100 |      100 |     100 |     100 |
  BookingController.js |     100 |      100 |     100 |     100 |
 Backend/Middlewares  |     100 |      100 |     100 |     100 |
  AuthValidation.js   |     100 |      100 |     100 |     100 |
 Backend/Models       |     100 |      100 |     100 |     100 |
  Subscriber.js       |     100 |      100 |     100 |     100 |
  User.js             |     100 |      100 |     100 |     100 |
 Backend/Routes       |     100 |      100 |     100 |     100 |
  AuthRouter.js       |     100 |      100 |     100 |     100 |
  BookingRouter.js    |     100 |      100 |     100 |     100 |
  SearchRouter.js     |     100 |      100 |     100 |     100 |
  Subscribe.js        |     100 |      100 |     100 |     100 |
  UserRouter.js       |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|-------------------

Test Suites: 10 passed, 10 total
Tests:       68 passed, 68 total
Snapshots:   0 total
Time:        26.926 s
Ran all test suites.
PS C:\Users\adity\OneDrive\Desktop\IT314_Flight_Ticket_Booking_System\IT314_G27_FlightBookingSystem\Backend>
```

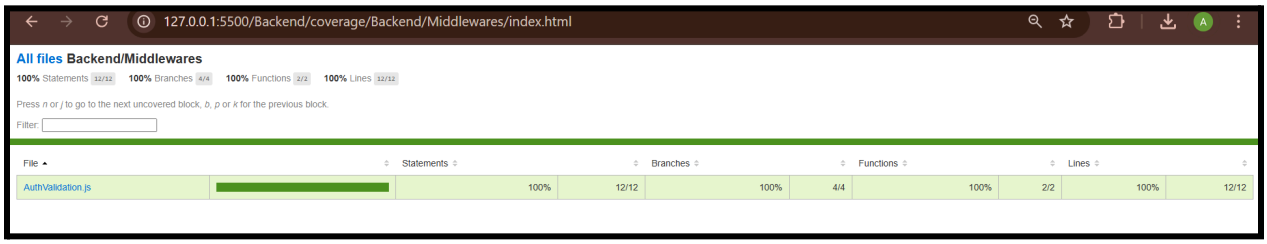
The coverage of all backend files returned in HTML File is provided below.

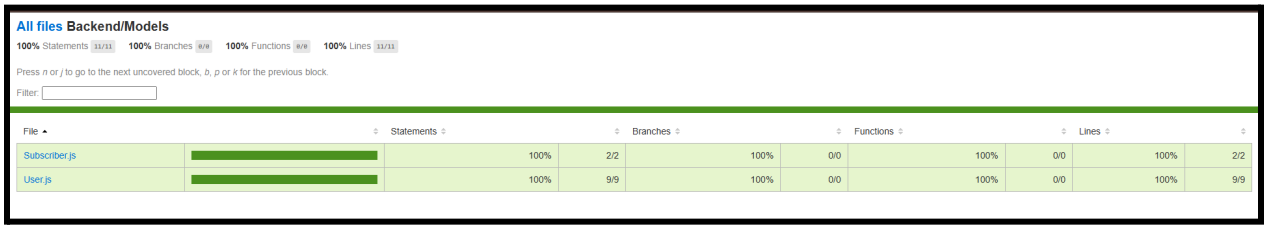


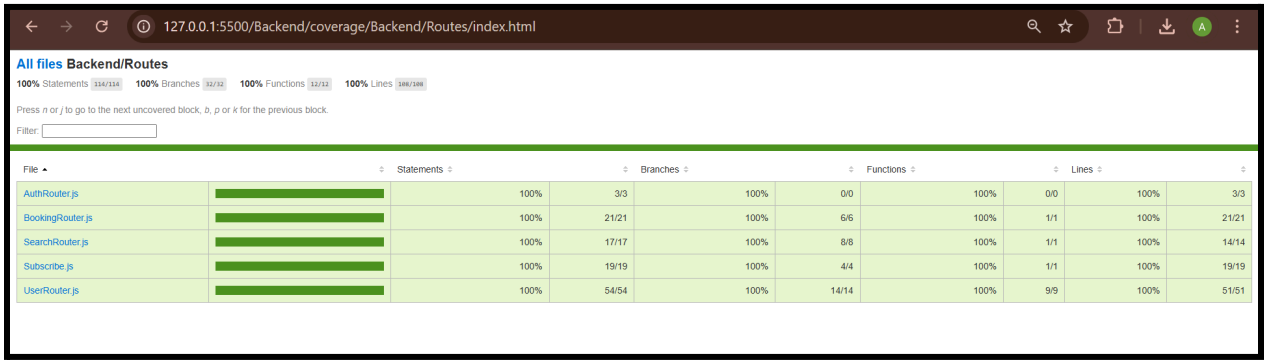
The coverage of the files in individual folders and their details is provided below.





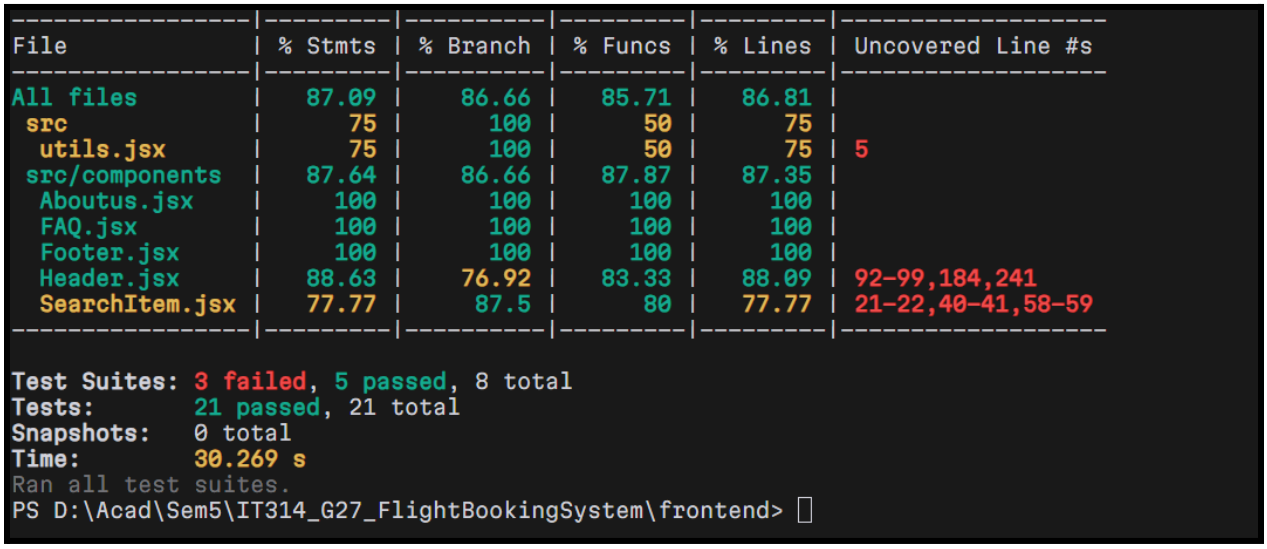




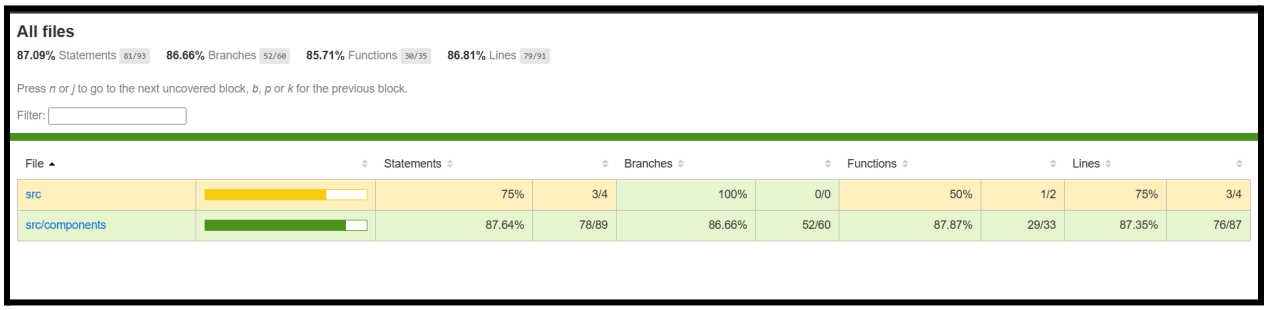


Frontend Unit Testing:

Coverage Report:



```
-----------------|---------|----------|---------|---------|-------------------
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |   87.09 |    86.66 |   85.71 |   86.81 |
 src             |      75 |      100 |      50 |      75 |
  utils.jsx      |      75 |      100 |      50 |      75 | 5
 src/components  |   87.64 |    86.66 |   87.87 |   87.35 |
  Aboutus.jsx    |     100 |      100 |     100 |     100 |
  FAQ.jsx        |     100 |      100 |     100 |     100 |
  Footer.jsx     |     100 |      100 |     100 |     100 |
  Header.jsx     |   88.63 |    76.92 |   83.33 |   88.09 | 92-99,184,241
  SearchItem.jsx |   77.77 |     87.5 |      80 |   77.77 | 21-22,40-41,58-59
-----------------|---------|----------|---------|---------|-------------------

Test Suites: 3 failed, 5 passed, 8 total
Tests:       21 passed, 21 total
Snapshots:   0 total
Time:        30.269 s
Ran all test suites.
PS D:\Acad\Sem5\IT314_G27_FlightBookingSystem\frontend>
```



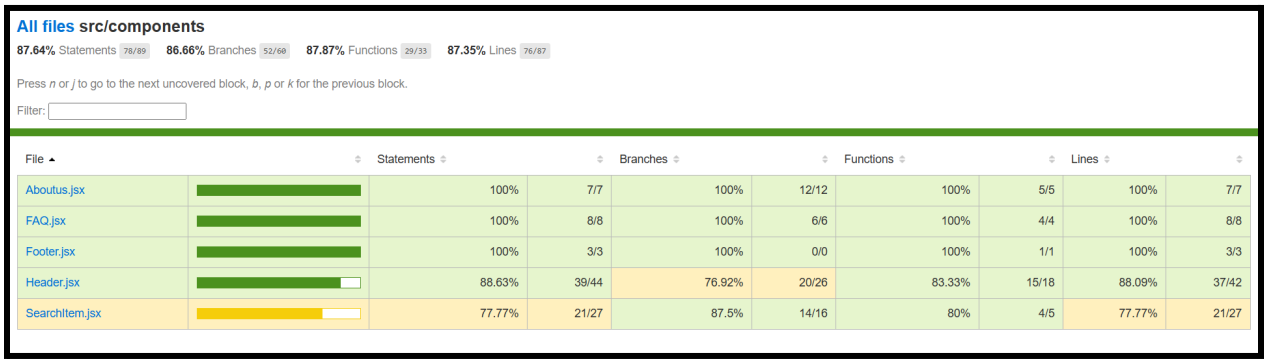
All files

87.09% Statements 81/93 **86.66%** Branches 52/60 **85.71%** Functions 30/35 **86.81%** Lines 79/91

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
src	75%	3/4	100%	0/0	50%	1/2	75%	3/4
src/components	87.64%	78/89	86.66%	52/60	87.87%	29/33	87.35%	76/87



All files src/components

87.64% Statements 78/89 **86.66%** Branches 52/60 **87.87%** Functions 29/33 **87.35%** Lines 76/87

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
Aboutus.jsx	100%	7/7	100%	12/12	100%	5/5	100%	7/7
FAQ.jsx	100%	8/8	100%	6/6	100%	4/4	100%	8/8
Footer.jsx	100%	3/3	100%	0/0	100%	1/1	100%	3/3
Header.jsx	88.63%	39/44	76.92%	20/26	83.33%	15/18	88.09%	37/42
SearchItem.jsx	77.77%	21/27	87.5%	14/16	80%	4/5	77.77%	21/27

Some examples of test cases from each suite:

(All test files are present in the source code)

```
// Test 2: Search Form Functionality
test('handles search form inputs and submission', async () => {
  render(
    <BrowserRouter>
      <Header type="home" />
    </BrowserRouter>
  );

  // Test From dropdown
  const fromInput = screen.getByTestId('from-input');
  fireEvent.change(fromInput, { target: { value: 'Delhi' } });
  expect(fromInput.value).toBe('Delhi');

  // Test To dropdown
  const toInput = screen.getByTestId('to-input');
  fireEvent.change(toInput, { target: { value: 'Mumbai' } });
  expect(toInput.value).toBe('Mumbai');

  // Test search button
  const searchButton = screen.getByRole('button', { name: /search/i });
  fireEvent.click(searchButton);

  expect(mockNavigate).toHaveBeenCalledWith('/flights', {
    state: expect.objectContaining({
      from: 'Delhi',
      to: 'Mumbai'
    })
  });
});
```

Suite: Header.test.jsx

This test verifies that the search form in the `Header` component correctly updates dropdown values for "From" and "To" fields and navigates to the `/flights` route with the expected state when the search button is clicked. It ensures proper user input handling and navigation behavior.

```jsx
test('quick links navigate to correct routes', () => {
  render(
    <BrowserRouter>
      <Footer />
    </BrowserRouter>
  );

  // Check for Quick Links section
  expect(screen.getByText('Quick Links')).toBeInTheDocument();

  // Test Home link
  const homeLink = screen.getByRole('link', { name: /home/i });
  expect(homeLink.getAttribute('href')).toBe('/');

  // Test About Us link
  const aboutLink = screen.getByRole('link', { name: /about us/i });
  expect(aboutLink.getAttribute('href')).toBe('/aboutus');

  // Test FAQ link
  const faqLink = screen.getByRole('link', { name: /faq/i });
  expect(faqLink.getAttribute('href')).toBe('/faq');
});
```

Suite: Footer.test.jsx

This test ensures that the "Quick Links" section in the `Footer` component displays correctly and that each link navigates to its intended destination (`/`, `/aboutus`, `/faq`). It validates the presence and functionality of navigation links.

```
test('toggles FAQ answers when questions are clicked', () => {
  render(
    <BrowserRouter>
      <FAQ {...mockProps} />
    </BrowserRouter>
  );

  // Find the first FAQ question
  const firstQuestion = screen.getByText(/What is SkyLynx and how does it work\?/);

  // Initially, answer should not be visible
  expect(screen.queryByText(/SkyLynx is your ultimate travel companion, designed to simpli

  // Click the question
  fireEvent.click(firstQuestion);

  // Answer should now be visible
  expect(screen.getByText(/SkyLynx is your ultimate travel companion, designed to simplify

  // Click again to close
  fireEvent.click(firstQuestion);

  // Answer should be hidden again
  expect(screen.queryByText(/SkyLynx is your ultimate travel companion, designed to simpli
});
```

Suite: Faq.test.jsx

Verifies that clicking on an FAQ question toggles the visibility of the corresponding answer, ensuring the answer is hidden initially, becomes visible after the first click, and hides again after a second click.

```
test('handles select button click for non-logged in user', () => {
  render(
    <BrowserRouter>
      <SearchItem
        flight={mockFlight}
        loggedInUser={null}
        setLoggedInUser={() => {}}
      />
    </BrowserRouter>
  );

  const selectButton = screen.getByRole('button', { name: /select/i });
  fireEvent.click(selectButton);

  expect(mockNavigate).toHaveBeenCalledWith('/login');
});
```

Suite: searchitem.test.jsx

Ensures that clicking the "Select" button when the user is not logged in redirects to the login page (/login).

```
test('switches between about and team sections', () => {
  render(
    <BrowserRouter>
      <AboutUs {...mockProps} />
    </BrowserRouter>
  );

  // Initially About section should be visible
  expect(screen.getByText('About Us')).toBeInTheDocument();

  // Click team button
  const teamButton = screen.getByRole('button', { name: /our team/i });
  fireEvent.click(teamButton);

  // Team section should be visible
  expect(screen.getByText('Meet Our Team')).toBeInTheDocument();

  // Click about button
  const aboutButton = screen.getByRole('button', { name: /About/i });
  fireEvent.click(aboutButton);

  // About section should be visible again
  expect(screen.getByText('About Us')).toBeInTheDocument();
});
```

Suite: AboutUs.test.jsx
This test checks the toggle functionality between the "About Us" and "Meet Our Team" sections in the AboutUs component. It verifies that clicking the respective buttons switches the visible content and maintains the toggle behavior as expected.

Summary for frontend unit testing:

Overall Coverage:

- **Statements**: 87.09%
- **Branches**: 86.66%
- **Functions**: 85.71%
- **Lines**: 86.81%

Key Findings:

- Most components, such as Aboutus.jsx, FAQ.jsx, and Footer.jsx, have achieved 100% coverage across all metrics.
- Header.jsx shows slight gaps, particularly in branches (76.92%) and functions (83.33%).
- SearchItem.jsx has the lowest coverage, with statements at 77.77%, branches at 87.5%, and lines at 77.77%.

Issues:

- Some test suites failed to run due to an **unexpected token error**, which limited testing of certain files.
- 3 out of 8 test suites failed, though 21 tests passed successfully.