Detailed Breakdown of the Jest Test Suites

1.1 Back-End

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	84.14	62.58	86.2	89.35	
Factory	100	100	100	100	
DBControllerFactory.ts	100	100	100	100	
controllers	76.47	66.27	79.76	79.88	
DBController.ts	76.47	66.27	79.76	79.88	418-442,582-697
repo	76.62	6.25	95	95.16	
accountsMaster.ts	82.14	0	100	100	36-108
postsMaster.ts	80	25	100	100	34-67
propertyMaster.ts	80	0	100	100	22-52
unitMaster.ts	63.15	0	80	80	53-55
routes	97.45	70	100	99.09	
accounts.ts	100	85.71	100	100	21
login.ts	94.11	66.66	100	94.11	35
properties.ts	94.28	50	100	100	27-59
signup.ts	100	75	100	100	25
routes/nested_routes	89.74	84.61	87.5	90.62	
posts.ts	100	80	100	100	10
units.ts	81.81	87.5	77.77	83.33	46-52
tests/unit/utils	100	100	100	100	
recordExistsTest.js	100	100	100	100	
types	100	100	100	100	
DBTypes.ts	100	100	100	100	

1.1.1 PropertyMaster Testing Documentation

What We're Testing: The test suite focuses on the PropertyMaster class, specifically its interactions with a database controller. This class serves as a layer between the application logic and the database, handling operations related to property data.
Components Involved:

- PropertyMaster: The class under test. It has methods like registerNewProperty, getProperty, and getAllProperties.
- DBControllerFactory: A factory class (mocked in the tests) that presumably creates instances of a DBController.
- DBTypes: Contains TypeScript type definitions, like PropertyData, which defines the structure of property data.

Test Structure and Logic:

1. **Mocks:** The DBControllerFactory is mocked using Jest's spyOn and mockImplementation functions. This means that instead of using a real database

controller, the tests use a mock object with pre-programmed behavior. This is important for isolating the PropertyMaster and ensuring tests are deterministic.

2. **Test Data:** The expected output data for the mocked DBController methods are defined as constants, such as createNewPropertyOutput, getPropertyOutput, and getAllPropertiesOutput. This data represents what the mocked DBController should return when called and is then used for assertions.

3. **Test Cases:**

    - registerNewProperty: Tests successful property registration and error handling. It checks if the correct data is passed to the DBController and if the correct response is returned.
    - getProperty: Tests successful property retrieval by ID and error handling.
    - getAllProperties: Tests successful retrieval of all properties and error handling.

    - errorHandler: A shared function across test cases to test the error handling of each method in the class - ensures they reject with the thrown error.

How Tests Pass: Each test case uses expect assertions to verify that:

- The PropertyMaster methods return the expected values based on the mocked DBController responses.
- The correct methods on the mocked DBController are called with the expected arguments using toHaveBeenCalledWith.
- For error handling tests, the test will pass if the method rejects with the same error thrown by the mocked DBController method. This is tested using .rejects.toEqual.

1.1.2 Properties Router Testing Documentation

What We're Testing: The test suite focuses on the Properties Router, ensuring proper handling of various routes related to properties, including registering new properties and retrieving property data.
Components Involved:

- Express: Framework used for building web applications in Node.js.
- supertest: Library for testing HTTP assertions.
- router: The router file handling properties routes.
- PropertyMaster: Repository class responsible for property operations.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes afterEach hook for clearing mocked functions between tests.
2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods.
3. **Tests ensure that:**
    - The router properly handles errors and sends a 500 response with the correct error message.
    - Register New Property Route: Tests the "/register" route for registering a new property.
    - Retrieve Property Data Route: Tests the "/real-estate" route for retrieving property data.
    - Retrieve Company Assets Route: Tests the "/real-estate/company-assets" route for retrieving all company assets.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected return values from the router endpoints.
- The interaction with the PropertyMaster repository class, ensuring correct method calls and handling of responses.

```
describe("POST /real-estate", () => {
  it("retrieves property data successfully", async () => {
    const mockReq = {
      body: { property_id: "1d2b6c84-2b4c-4893-8fb6-cf76f255d990" },
    };

    const propertyData = {
      status: 202,
      data: {
        property_id: "1d2b6c84-2b4c-4893-8fb6-cf76f255d990",
        admin_id: "test-admin",
        unit_count: 20,
        locker_count: 10,
        parking_count: 50,
        address: "123 Main St",
        picture: "main_street.jpg",
      },
    };

    jest
      .spyOn(propertyPrototype, "getProperty")
      .mockResolvedValue(propertyData);

    let response = await request(app).post("/real-estate").send(mockReq.body);
    expect(response.body).toEqual(propertyData);
    expect(response.status).toEqual(202);
    expect(propertyPrototype.getProperty).toHaveBeenCalledWith(
      mockReq.body.property_id
    );
  });

  errorHandler("/real-estate", "getProperty");
});
```

1.1.3 SignUp Backend Testing Documentation

What We're Testing: The test suite focuses on the SignUp middleware, ensuring proper handling of user registration requests, including scenarios such as empty request body, successful registration, and handling of existing users.
Components Involved:

- Express: Framework used for building web applications in Node.js.
- supertest: Library for testing HTTP assertions.
- router: The router file handling signup routes.
- AccountsMaster: Repository class responsible for user account operations.

Test Structure and Logic:

1. **Setup and Teardown:** Uses beforeEach and afterEach hooks to set up and tear down the test environment, clearing mocked functions between tests.
2. **Empty Request Body Test:** Verifies that a 400 status and appropriate error message are returned if the request body is empty.
3. **Successful Registration Test:** Mocks the registerUser method of AccountsMaster to simulate successful registration.
4. **Existing User Test:** Mocks the registerUser method to simulate an existing user scenario.
5. **Error Handling Test:** Mocks the registerUser method to throw an error.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected HTTP status codes and response bodies.
- The interaction with the AccountsMaster repository class, ensuring correct method calls and handling of responses.

1.1.4 Login Backend Testing Documentation

What We're Testing: The test suite focuses on the Login middleware, ensuring proper handling of user login requests, including scenarios such as empty request body, successful login, and handling of failed login attempts.
Components Involved:

- Express: Framework used for building web applications in Node.js.
- supertest: Library for testing HTTP assertions.
- router: The router file handling login routes.
- AccountsMaster: Repository class responsible for user account operations.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach and afterEach hooks for setting up and tearing down the test environment, clearing mocked functions between tests.
2. **Empty Request Body Test:** Verifies that a 400 status and appropriate error message are returned if the request body is empty.
3. **Successful Login Test:** Mocks the getUserDetails method of AccountsMaster to simulate successful login.
4. **Failed Login Test:** Mocks the getUserDetails method to simulate a failed login attempt.
5. **Error Handling Test:** Mocks the getUserDetails method to throw an error.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected HTTP status codes and response bodies.
- The interaction with the AccountsMaster repository class, ensuring correct method calls and handling of responses.

```
// --- Test: Successful login ---
it("should send 202 and login data on successful login", async () => {
  let req = { body: { email: "michael@example.com", password: "password6" } };
  let res = {
    status: 202,
    data: {
      account_id: "6e8f4b3c-4103-44b3-b694-cb9f6e3e3fc9",
      fullname: "Michael Scott",
      email: "michael@example.com",
      account_type: "Public",
    },
  }

  jest
    .spyOn(accountPrototype, "getUserDetails")
    .mockResolvedValueOnce(res);

  const response = await request(app).post("/").send(req.body);

  expect(accountPrototype.getUserDetails).toHaveBeenCalledWith(
    "michael@example.com",
    "password6"
  );
  expect(response.status).toEqual(res.status);
  expect(response.body).toEqual({
    response: "User logged in successfully!",
    loginData: res.data,
  });
});
```

1.1.5 Posts Router Testing Documentation

What We're Testing:
The test suite focuses on the Posts Router, ensuring proper handling of various routes related to posts, including creating posts, retrieving user posts, and retrieving property channel posts.
Components Involved:

- Express: Framework used for building web applications in Node.js.
- supertest: Library for testing HTTP assertions.
- router: The router file handling posts routes.
- PostsMaster: Repository class responsible for post operations.
- DBTypes: Contains TypeScript type definitions, like PostDetails, which define the structure of post data.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach and afterEach hooks for setting up and tearing down the test environment, and clearing mocked functions between tests.
2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods.
3. **Create Post Route:** Tests the "/create" route for creating a new post.
4. **User Posts Route:** Tests the "/user-posts" route for retrieving user posts.
5. **Property Channel Posts Route:** Tests the "/property-channel-posts" route for retrieving property channel posts.

How Tests Pass: Each test case utilizes assertions to verify:

- the expected HTTP status codes and response bodies,
- the interaction with the PostsMaster repository class, ensuring correct method calls and handling of responses.

1.1.6 PostsMaster Testing Documentation

What We're Testing:
The test suite focuses on the PostsMaster class, responsible for making queries to the database related to posts. Tests ensure proper handling of various scenarios such as creating posts, retrieving user posts, and retrieving property channel posts.
Components Involved:

- PostsMaster: The class under test, responsible for database operations related to posts.
- DBControllerFactory: Factory class responsible for creating instances of the database controller.
- IDBController: Interface defining methods for interacting with the database.
- PostDetails: Type definition for post data structure.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach and afterEach hooks for setting up and tearing down the test environment, clearing mocked functions between tests.
2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods.
3. **Create Post Method:** Tests the createPost method for creating a new post.
4. **Get User Posts Method:** Tests the getUserPosts method for retrieving user posts.
5. **Get Property Posts Method:** Tests the getPropertyPosts method for retrieving property channel posts.

How Tests Pass: Each test case utilizes assertions to verify:

- the expected return values from the class methods,
- the interaction with the database controller, ensuring correct method calls and handling of responses.

1.1.7 UnitMaster Testing Documentation

What We're Testing:
The test suite focuses on the UnitMaster, ensuring proper handling of various operations related to units, including registering new units and retrieving unit data.
Components Involved:

- UnitMaster: Repository class responsible for unit operations.
- DBControllerFactory: Factory for creating instances of DB controllers.
- createUnitOutput, getUnitOutput, getAllUnitsOutput: Mocked outputs for unit-related operations.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize a new instance of UnitMaster before each test, and afterEach hook for clearing mocked functions between tests.
2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods.
3. **Register New Unit:** Tests the registerUnit method for registering a new unit.
4. **Retrieve Unit Data:** Tests the getUnit method for retrieving unit data.
5. **Retrieve All Units for a Property:** Tests the getPropertyUnits method for retrieving all units for a property.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected return values from the UnitMaster methods
- The interaction with the DB controller, ensuring correct method calls and handling of responses.

```
describe("getPropertyUnits", () => {
  it("retrieves all units for a property successfully", async () => {
    const mockData = { property_id: "1d2b6c84-2b4c-4893-8fb6-cf76f255d990" };

    let unitSpy = jest.spyOn(unitController, "getPropertyUnits");

    let result = await unitController.getPropertyUnits(mockData.property_id);
    expect(result).toEqual(getAllUnitsOutput);
    expect(unitSpy).toHaveBeenCalled();
    expect(unitSpy).toHaveBeenCalledWith(mockData.property_id);
    expect(unitController.dbController.getAllUnits).toHaveBeenCalled();
    expect(unitController.dbController.getAllUnits).toHaveBeenCalledWith(
      mockData.property_id
    );
  });

  errorHandler("getAllUnits");
});
```

1.1.8 Unit Router Testing Documentation

What We're Testing:

The test suite focuses on the Unit Router, ensuring proper handling of requests related to units, including registering new units, retrieving unit data, and retrieving property assets.

Components Involved:

- UnitMaster: Repository class responsible for unit operations.
- Express Router: Handles HTTP requests related to units.
- Mocked UnitMaster methods: registerUnit, getUnit, getPropertyUnits.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook for setup tasks before each test, and afterEach hook for clearing mocked functions between tests.
2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods.
3. **Register New Unit:** Tests the POST /register route for registering a new unit.
4. **Retrieve Unit Data:** Tests the POST /get-unit route for retrieving unit data.
5. **Retrieve Property Assets:** Tests the POST /property-assets route for retrieving property assets.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected return values from the UnitMaster methods
- The interaction with the UnitMaster, ensuring correct method calls and handling of responses.

1.1.9 AccountsMaster Testing Documentation

What We're Testing:

The test suite for AccountsMaster ensures the proper functioning of user and employee account-related operations such as user registration, retrieving user details, employee registration, retrieving employee details, and retrieving employees for a specific property.

Components Involved:

- AccountsMaster: Repository class responsible for account operations.
- Mocked bcryptjs library: Used to mock password hashing.

- DBControllerFactory: Factory class for creating database controllers.
- Mocked DBControllerFactory methods: createInstance.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook for setup tasks before each test. Utilizes afterEach hook for clearing mocked functions between tests.

2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods. Tests ensure that AccountsMaster properly handles errors and rejects promises with the correct error messages.

3. **User Details Retrieval:** Tests the getUserDetails method for retrieving user details. Mocks the getPublicUser method of DBControllerFactory to simulate successful retrieval of user details. Verifies that the method correctly retrieves the user details and returns the expected response.

4. **Employee Details Retrieval:** Tests the getEmployeeDetails method for retrieving employee details. Mocks the getEmployee method of DBControllerFactory to simulate successful retrieval of employee details. Verifies that the method correctly retrieves the employee details and returns the expected response.

5. **User Registration:** Tests the registerUser method for registering a new user. Mocks the createNewPublicUser method of DBControllerFactory to simulate successful user registration. Verifies that the method correctly registers the user and returns the expected response.

6. **Employee Registration:** Tests the registerEmployee method for registering a new employee. Mocks the createNewEmployee method of DBControllerFactory to simulate successful employee registration. Verifies that the method correctly registers the employee and returns the expected response.

7. **Retrieve Employees for a Property:** Tests the getPropertyEmployees method for retrieving all employees for a property. Mocks the getAllEmployees method of DBControllerFactory to simulate successful retrieval of employees. Verifies that the method correctly retrieves the employees and returns the expected response.

8. **Database Connection Closure:** Tests the close method to ensure proper closure of the database connection.

How Tests Pass: Each test case utilizes assertions to verify:

- The expected return values from the DBControllerFactory methods.

- The interaction with the DBControllerFactory, ensuring correct method calls and handling of responses.

1.1.10 Accounts Router Testing Documentation

What We're Testing:
The test suite for Express route handlers ensures that the API routes defined in the accounts module of the application behave as expected. It covers user registration, user details retrieval, employee registration, employee details retrieval, and retrieval of property employees.

Components Involved:

- **Express**: Web application framework used for handling API requests.
- **AccountsMaster**: Repository class responsible for account operations.
- **Supertest**: Library used for testing HTTP requests.
- **Router**: Express router containing the routes for account-related operations.
- **Mocked AccountMaster methods**: registerUser, getUserDetails, registerEmployee, getEmployeeDetails, getPropertyEmployees.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to mock the relevant methods of the AccountsMaster class before each test. Utilizes afterEach hook to clear mocked methods between tests.

2. **Error Handling:** Defines a helper function errorHandler to test error handling for different methods. Tests ensure that the Express route handlers properly handle errors and send the appropriate HTTP response with a 500 status code and an error message.

3. **User Registration:** Tests the /register route for user registration. Mocks the registerUser method of AccountsMaster to simulate successful user registration. Verifies that the route correctly registers the user and returns the expected response.

4. **User Details Retrieval:** Tests the /users route for retrieving user details. Mocks the getUserDetails method of AccountsMaster to simulate successful retrieval of user details. Verifies that the route correctly retrieves the user details and returns the expected response.

5. **Employee Registration:** Tests the /register/employee route for employee registration. Mocks the registerEmployee method of AccountsMaster to simulate successful employee registration. Verifies that the route correctly registers the employee and returns the expected response.
6. **Employee Details Retrieval:**Tests the /employees route for retrieving employee details. Mocks the getEmployeeDetails method of AccountsMaster to simulate successful retrieval of employee details. Verifies that the route correctly retrieves the employee details and returns the expected response.
7. **Retrieve Property Employees:** Tests the /employees/property-agents route for retrieving property employees. Mocks the getPropertyEmployees method of AccountsMaster to simulate successful retrieval of property employees. Verifies that the route correctly retrieves the property employees and returns the expected response.

How Tests Pass Each test case utilizes assertions to verify:

- The expected HTTP response status and body.
- The interaction with the AccountsMaster class, ensuring correct method calls and handling of responses.

```
// /employees tests
describe("/employees", () => {
  it("should get employee details", async () => {
    let req = {
      body: {
        email: "testuser@example.com",
        password: "testPassword",
      },
    };
    const response = await request(app).post("/employees").send(req.body);
    expect(response.status).toEqual(202);
    expect(response.body).toEqual(getEmployeeOutput);
    expect(accountPrototype.getEmployeeDetails).toHaveBeenCalledTimes(1);
  });

  errorHandler("/employees", "getEmployeeDetails");
});

// /employees/property-agents tests
describe("/employees/property-agents", () => {
  it("should get property employees", async () => {
    let req = {
      body: {
        property_id: "test-property-id",
      },
    };
    const response = await request(app)
      .post("/employees/property-agents")
      .send(req.body);
    expect(response.status).toEqual(200);
    expect(response.body).toEqual(getPropertyEmployeesOutput);
    expect(accountPrototype.getPropertyEmployees).toHaveBeenCalledTimes(1);
    expect(accountPrototype.getPropertyEmployees).toHaveBeenCalledWith(
      req.body.property_id
    );
  });
```

1.1.11 Testing the DBController class :

1.1.11.1 DBController Unit methods Test Documentation

What We're Testing:
The test suite for the DBController class ensures that the database operations defined in the controller behave as expected. It covers unit creation, unit retrieval by ID, and retrieval of all units associated with a property.

Components Involved:

- **DBController:** Controller class responsible for database operations related to units.

- **Mocked Dependencies:**
    - **sqlite3:** Mocked to simulate database interactions.
    - **fs:** Mocked for file system operations.
    - **uuid:** Mocked for generating UUIDs.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize an instance of the DBController class before each test. Utilizes afterEach hook to clear all mocked methods and reset the state between tests.
2. **Unit Creation:** Tests the createNewUnit method of the DBController class. Mocks the necessary database interaction methods to simulate successful unit creation. Verifies that the method correctly creates a new unit and returns the expected response.
3. **Unit Retrieval by ID:** Tests the getUnit method of the DBController class. Mocks the recordExists method to simulate whether the unit exists in the database or not. Verifies that the method correctly retrieves unit information if the unit exists, and rejects with an appropriate error message if the unit does not exist.
4. **Retrieval of All Units Associated with a Property:** Tests the getAllUnits method of the DBController class. Mocks the recordExists method to simulate whether any units are associated with the given property ID. Verifies that the method correctly returns all units associated with the property if they exist, and resolves with an appropriate message if no units are found.

How Tests Pass Each test case utilizes assertions to verify:

- The expected behavior of the DBController methods, including correct method calls and handling of responses.
- The interaction with the mocked dependencies, ensuring that database operations are performed as expected.

1.1.11.2 DBController Public User methods Test Documentation

What We're Testing:

The test suite for the Public User functionalities within the DBController class ensures that the database operations related to public user registration and retrieval behave as expected. It covers creating a new public user and retrieving public user details.

Components Involved:

- DBController: Controller class responsible for database operations related to public users.
- Mocked Dependencies:
    - sqlite3: Mocked to simulate database interactions.
    - fs: Mocked for file system operations.
    - uuid: Mocked for generating UUIDs.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize an instance of the DBController class before each test. Utilizes afterEach hook to clear all mocked methods and reset the state between tests.

2. **Create New Public User:** Tests the createNewPublicUser method for registering a new public user. Mocks the recordExists method to simulate the existence of the user. Verifies that the method correctly returns a 400 status code with an appropriate message if the user already exists. Verifies that the method adds a new user and returns a 201 status code with the inserted ID if the user doesn't exist.

3. **Retrieve Public User:** Tests the getPublicUser method for retrieving public user details.Mocks the recordExists method to simulate the existence of the user. Verifies that the method returns 'pass' to indicate a successful fetch if the user exists. Verifies that the method returns a 400 status code with an appropriate message if the user doesn't have an account.

How Tests Pass Each test case utilizes assertions to verify:

- The expected return values from the methods being tested.
- The interaction with the database, ensuring correct queries are executed and values are inserted/retrieved accurately.

1.1.11.3 DBController Property methods Test Documentation

What We're Testing:

The Property Testing suite within the DBController class ensures that the database operations related to properties behave as expected. It covers creating a new property, retrieving property details, and retrieving all properties associated with an employee.

Components Involved:

- **DBController:** Controller class responsible for database operations related to properties.
- **Mocked Dependencies:**
    - **sqlite3:** Mocked to simulate database interactions.
    - **fs:** Mocked for file system operations.
    - **uuid:** Mocked for generating UUIDs.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize an instance of the DBController class before each test. Utilizes afterEach hook to clear all mocked methods and reset the state between tests.

2. **Create New Property:** Tests the createNewProperty method for adding a new property. Mocks the recordExists method to simulate property existence. Verifies that the method correctly creates a new property and returns a 201 status code with the inserted ID if the property doesn't exist. Verifies that the method rejects with an appropriate message if the property already exists.

3. **Retrieve Property:** Tests the getProperty method for retrieving property details. Mocks the recordExists method to simulate property existence. Verifies that the method returns property information if the property exists. Verifies that the method rejects with an appropriate message if the property doesn't exist.

4. **Retrieve All Properties:** Tests the getAllProperties method for retrieving all properties associated with an employee. Mocks the recordExists method to simulate the existence of the employee. Verifies that the method returns a 204 status code with a message if no properties were found for the employee. Verifies that the method returns all properties if found.

How Tests Pass Each test case utilizes assertions to verify:

- The expected return values from the methods being tested.
- The interaction with the database, ensuring correct queries are executed and values are inserted/retrieved accurately.

```
describe("getAllProperties", () => {
  let testEmpID = "test-id";
  let spy;
  it("should resolve to a status 204 if no properties were found", async () =>
    spy = jest
      .spyOn(dbController, "recordExists")
      .mockResolvedValueOnce(false);

    await expect(dbController.getAllProperties(testEmpID)).resolves.toEqual({
      status: 204,
      message: "No properties found for given employee",
    });

    recordExistsTest(spy, {
      tableName: "CMC_Admin",
      fieldName: "admin_id",
      value: testEmpID,
    });
  });

  it("should return all properties if found", async () => {
    spy = jest
      .spyOn(dbController, "recordExists")
      .mockResolvedValueOnce(true);

    let getAllPropertiesSpy = jest.spyOn(dbController, "getAllProperties");

    await expect(
      dbController.getAllProperties(testEmpID)
    ).resolves.toBeTruthy();
    expect(dbController.db.all).toHaveBeenCalled();
    expect(dbController.db.all).toHaveBeenCalledWith(
      (dbController.db.all as jest.Mock).mock.calls[0][0],
      [testEmpID]
```

1.1.11.4 DBController Post methods Test Documentation

What We're Testing:

The Posts Testing suite within the DBController class ensures that the database operations related to posts behave as expected. It covers creating a new post, retrieving all posts by a user, retrieving all replies to a post, and retrieving all posts related to a property.

Components Involved:

- **DBController:** Controller class responsible for database operations related to posts.
- **Mocked Dependencies:**
  - **sqlite3:** Mocked to simulate database interactions.
  - **fs:** Mocked for file system operations.
  - **uuid:** Mocked for generating UUIDs.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize an instance of the DBController class before each test. Utilizes afterEach hook to clear all mocked methods and reset the state between tests.
2. **Create New Post:** Tests the createNewPost method for adding a new post. Verifies that the method correctly creates a new post and returns a 201 status code with the inserted ID.
3. **Retrieve All User Posts:** Tests the getAllUserPosts method for retrieving all posts by a user. Verifies that the method returns a 204 status code with a message if no posts were found for the user. Verifies that the method returns all user posts if found.
4. **Retrieve All Post Replies:** Tests the getAllPostsReplies method for retrieving all replies to a post. Verifies that the method returns a 204 status code with a message if no replies were found for the post. Verifies that the method returns all post replies if found.
5. **Retrieve All Property Posts:** Tests the getAllPropertyPosts method for retrieving all posts related to a property. Verifies that the method returns a 204 status code with a message if no posts were found for the property. Verifies that the method returns all property posts if found.

How Tests Pass Each test case utilizes assertions to verify:

- The expected return values from the methods being tested.
- The interaction with the database, ensuring correct queries are executed and values are inserted/retrieved accurately.

1.1.11.5 DBController Employee methods Test Documentation

What We're Testing:
The Employee Testing suite within the DBController class ensures that the database operations related to employees behave as expected. It covers creating a new employee account, retrieving employee information, and retrieving all employees associated with a property.

Components Involved:

- **DBController:** Controller class responsible for database operations related to employees.

- **Mocked Dependencies:**
    - **sqlite3:** Mocked to simulate database interactions.
    - **fs:** Mocked for file system operations.
    - **uuid:** Mocked for generating UUIDs.

Test Structure and Logic:

1. **Setup and Teardown:** Utilizes beforeEach hook to initialize an instance of the DBController class before each test. Utilizes afterEach hook to clear all mocked methods and reset the state between tests.
2. **Create New Employee:** Tests the createNewEmployee method for creating a new employee account. Verifies that the method creates an employee account even if the employee already has a public account. Verifies that the method adds a new employee and a new public user if the employee does not yet have an account.
3. **Retrieve Employee:** Tests the getEmployee method for retrieving employee information. Verifies that the method resolves to indicate a successful fetch if the employee exists. Verifies that the method returns an error if the employee does not have an account.
4. **Retrieve All Employees:** Tests the getAllEmployees method for retrieving all employees associated with a property. Verifies that the method resolves to a status 204 with a message if no employees were found for the property. Verifies that the method returns all employees associated with the property if found.

How Tests Pass Each test case utilizes assertions to verify:

- The expected return values from the methods being tested.
- The interaction with the database, ensuring correct queries are executed and values are inserted/retrieved accurately.

1.2 Front-End

1.2.1 SignUp Testing Documentation

What We're Testing:

The test suite focuses on the `SignUp` component, ensuring it renders correctly, handles form validation errors, and navigates to the login view when the corresponding link is clicked.

Components Involved:

- `SignUp`: The component under test, responsible for rendering a sign-up form.
- `React`: Required for rendering and interacting with React components.
- `@testing-library/react`: Provides utilities for testing React components, such as `render` and `fireEvent`.

Test Structure and Logic:

1. Rendering Test:
   - Checks if the sign-up form renders correctly with all necessary elements.
   - Verifies that specific text content and input fields are present.

2. Form Validation Error Test:
   - Simulates form submission with invalid input.
   - Validates that the appropriate error message is displayed when form validation fails.

3. Navigation Test:
   - Mocks the `setView` function to track calls.
   - Simulates a click on the "Log in" link and verifies that `setView` is called with the correct view name.

How Tests Pass: Each test case utilizes assertions to verify:

- The presence of expected elements and text content in the rendered component.
- The display of error messages upon form validation failure.
- The invocation of specific functions, such as `setView`, with the expected parameters.

```
describe('SignUp component', () => {
  it('renders correctly', () => {
    const { getByText, getByLabelText } = render(<SignUp />);
    expect(getByText('Create an Account')).toBeInTheDocument();
    expect(getByLabelText('Full Name:')).toBeInTheDocument();
    expect(getByLabelText('Email:')).toBeInTheDocument();
    expect(getByLabelText('Password:')).toBeInTheDocument();
    expect(getByLabelText('Confirm Password:')).toBeInTheDocument();
    expect(getByText('Sign Up')).toBeInTheDocument();
    expect(getByText('Already have an account?')).toBeInTheDocument();
  });

  it('displays error message if form validation fails', () => {
    const { getByLabelText, getByText } = render(<SignUp />);
    const passwordInput = getByLabelText('Password:');
    fireEvent.change(passwordInput, { target: { value: 'password' } });
    fireEvent.submit(getByText('Sign Up'));
    expect(getByText('The fix the following to continue:')).toBeInTheDocument();
    expect(getByText('1. Passwords must match')).toBeInTheDocument();
  });
```