**Test Case Title: Component Creation**

**Test Suite:** Referral Component

**Test Purpose:** Ensure the component is created successfully.

**Test Description:**

* **Test Objective:**  
  This unit test verifies that the component is created correctly when the component's constructor is called. It checks if the component instance is truthy (i.e., it exists and is properly initialized).
* **Test Steps:**
  + 1. Initialize the component using Angular's TestBed.
    2. Assert that the component is created successfully
* **Test Data:**  
  No specific input data is used for this test, as the test only checks the instantiation of the component.

**Test Case Title: Form Controls Initialization**

**Test Suite:** Referral Component

**Test Purpose:** Verify that the form controls in the referralForm are initialized with the correct enabled/disabled state.

**Test Description:**

* **Test Objective:**  
  This unit test ensures that the form controls in the referralForm are initialized correctly, specifically that the appropriate form controls are enabled or disabled based on the expected behaviour. It checks the state (enabled/disabled) of each individual form control in the form.
* **Preconditions:**
  + The component must have the form (referralForm) initialized in the component's setup, most likely in ngOnInit.
  + The form controls are expected to be in a certain state (enabled or disabled) upon component initialization.
* **Test Steps:**
  1. Instantiate the component with TestBed to initialize the component and its form.
  2. Access the referralForm and verify the enabled/disabled state of each form control.
  3. Assert that each form control has been correctly set to the expected state.
* **Test Data:**  
  No specific input data is required. This test is based on the initial state of the form controls when the component is instantiated.

**Test Assertions:**

* **Assertion Examples:**
  + expect(component.referralForm).toBeDefined();  
    Ensures that the form is defined (i.e., it has been initialized).
  + expect(component.referralForm.get('account\_no')?.enabled).toBe(false);  
    Verifies that the account\_no form control is disabled when account no pass in the URL otherwise it should be enabled.
  + expect(component.referralForm.get('client\_name')?.enabled).toBe(true);  
    Verifies that the client\_name form control is enabled.
  + This pattern continues for each form control to ensure its enabled/disabled state is correct.

**Test Case Title: Handle Case When account\_no is Not Present in Query Params**

**Test Suite:** Referral Component

**Test Purpose:** Verify that the component correctly handles the case when the account\_no is not present in the query parameters.

**Test Description:**

* **Test Objective:**  
  This unit test ensures that when the account\_no is missing from the query parameters, the component does not triggeron any methods that would require it during the page loading. It also verifies that the form control for account\_no remains disabled and unaffected when no account\_no is present in the query parameters.
* **Preconditions:**
  + The component should be dependent on the account\_no query parameter for certain functionality.
  + mockActivatedRoute.queryParamMap is used to simulate the absence of the account\_no query parameter.
* **Test Steps:**
  1. Mock the query parameters to simulate the absence of the account\_no by using mockActivatedRoute.queryParamMap = of(new Map()).
  2. Call fixture.detectChanges() to trigger Angular's change detection and initialize the component.
  3. Spy on the methods (getAccountDetailsByAccountNo, getReporterDetails, getTransactionType) to ensure they are not called when account\_no is missing from the query parameters.
  4. Assert that the methods are not invoked by verifying that their spy calls are not made
  5. Verify that the account\_no form control remains enabled.
* **Test Data:**  
  No specific data is required for this test case. The test focuses on verifying the absence of account\_no in the query parameters.

**Test Assertions:**

* **Assertion Examples:**
  + expect(getAccountDetailsByAccountNoSpy).not.toHaveBeenCalled();  
    Verifies that getAccountDetailsByAccountNo is not called when account\_no is missing.
  + expect(component.referralForm.get('account\_no')?.enabled).toBe(true);  
    Verifies that the account\_no form control remains enabled.

**Test Case Title:** Verify getAccountDetailsByAccountNo is called on ngOnInit when account no is given in URL and proper form control and visibility behavior  
**Test Suite:** Referral Component  
**Test Purpose:** Ensure getAccountDetailsByAccountNo is called on ngOnInit when account\_no is provided, and check the form control state and visibility.

**Test Description:**  
• **Test Objective:**  
This unit test ensures that the getAccountDetailsByAccountNo method is called during the ngOnInit lifecycle hook if account\_no is provided in URL. It also verifies the following behaviors:

* The account\_no form control is disabled after ngOnInit.
* The isVisible property is set to true after initialization.

• **Test Steps:**

1. Create a spy on the getAccountDetailsByAccountNo method to monitor its call.
2. Trigger the ngOnInit lifecycle hook on the component.
3. Assert that the account\_no form control is disabled.
4. Assert that the isVisible property is set to true here IsVisible is responsible for to show and hide the Master RepId, Client Name and Client SSN.
5. Verify that the getAccountDetailsByAccountNo method was called.

**Test Case Title:** Verify getReportersDetails and getTransactionType is called on ngOnInit  
**Test Suite:** Referral Component  
**Test Purpose:** Ensure getReportersDetails, getTransactionType is called when the component initializes (ngOnInit).

**Test Description:**  
• **Test Objective:**  
This unit test verifies that the getReportersDetails, getTransactionType method is called during the component's ngOnInit lifecycle hook. The test ensures that the method is invoked when the component is initialized.  
• **Test Steps:**

1. Create a spy on the getReportersDetails and getTransactionType method to monitor its invocation.
2. Trigger the ngOnInit lifecycle hook on the component.
3. Assert that the getReportersDetails and getTransactionType method was called during ngOnInit.

**Test Case Title:** Verify the form is updated with reporter details when the API call is successful  
**Test Suite:** Referral Component  
**Test Purpose:** Ensure that the form is correctly populated with the reporter details when the API call to fetch reporter information succeeds.

**Test Description:**  
• **Test Objective:**  
This unit test verifies that when the getReporterDetails method successfully fetches reporter details from the API, the form controls are updated with the data received. It checks that the form fields for reporter\_first\_name, reporter\_last\_name, and reporter\_email are populated with the corresponding values from the API response.  
• **Test Steps:**

1. Mock the API service to return a successful response with reporter details (first name, last name, and email).
2. Call the getReporterDetails method, which triggers the API call.
3. Assert that the form controls for reporter\_first\_name, reporter\_last\_name, and reporter\_email are updated with the values from the mock API response.

• **Test Data:**

* Mock API response:

{

"reporter\_first\_name": "DevInt Vaishali",

"reporter\_last\_name": "Zachrich",

"reporter\_email": "vaishali.zachrich@lpl.com"

}

**Test Case Title:** Verify logging of "Reporter Details not found" when no reporter details are returned  
**Test Suite:** Referral Component  
**Test Purpose:** Ensure that the appropriate message is logged when no reporter details are returned from the API.

**Test Description:**  
• **Test Objective:**  
This unit test verifies that when the getReporterDetails method is called and the API returns no reporter details (i.e., a null response), the message "Reporter Details not found" is logged to the console.  
• **Test Steps:**

1. Mock the API service to return null (indicating no reporter details found).
2. Spy on the console.log method to capture the output.
3. Call the getReporterDetails method.
4. Assert that console.log was called with the message "Reporter Details not found" when no reporter details are returned.

• **Test Data:**

* Mock API response: null (indicating no reporter details found).

**Unit Test: should handle reporter details correctly when the form is empty initially**

**Description**:  
This test verifies that the component handles the case correctly when the reporter details are fetched, but no data is returned (i.e., the form is empty initially). It simulates a scenario where the getReporterDetails method is called and the mock API service returns null, representing the absence of reporter data. The test then checks if the form fields remain empty, confirming that no unintended updates occur when no data is available.

**Test Steps**:

1. Simulate the initial state where the form fields are empty and the mock API service returns null when getReporterDetails is called.
2. Call the getReporterDetails method on the component.
3. Verify that the value of the reporter\_first\_name form control is still an empty string ("").
4. Verify that the value of the reporter\_last\_name form control is still an empty string ("").
5. Verify that the value of the reporter\_email form control is still an empty string ("").

**Expected Outcome**:

* When no data is returned (i.e., null), the reporter\_first\_name, reporter\_last\_name, and reporter\_email form controls should retain empty string values (""), confirming that the form fields are not updated unexpectedly.

**Unit Test: should handle the reporter details error correctly if the API call fails**

**Description**:  
This test verifies that the component correctly handles errors when the API call to retrieve reporter details fails. It simulates a failure scenario where the getReporterDetails method triggers an error (mocked as an Error object). The test checks whether the component logs the error message to the console as expected.

**Test Steps**:

1. Mock an error scenario where the getReporterDetails method of the API service rejects with an error (e.g., mockError).
2. Create a spy on the console.log method to capture the log output.
3. Call the getReporterDetails method on the component.
4. Verify that the error message "Error Occured!" along with the actual error (mockError) is logged to the console.

**Expected Outcome**:

* When the API call fails, the component should log the message "Error Occured!" along with the error details (mockError) to the console, ensuring that the error is correctly handled and reported.

**Unit Test: should call getAccountDetails and populate form controls on success**

**Description**:  
This test verifies that when the getAccountDetailsByAccountNo method is called, the component successfully fetches the account details via the getAccountDetails API call and correctly populates the form controls with the retrieved data. It simulates a successful API response containing account details and checks if the form fields are populated as expected.

**Test Steps**:

1. Mock a successful API response (mockResponse) with account details, including client\_name, client\_ssn, and master\_repid.
2. Mock the getAccountDetails API method to resolve with this response when called.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Wait for the asynchronous operations to complete using fixture.whenStable().
5. Verify that the getAccountDetails method was called with the correct account number ('12345' in this case).
6. Check that the form controls client\_name, client\_ssn, and master\_rep\_id are populated with the corresponding values from the API response ('John Doe', '123-45-6789', and 'master123', respectively).

**Expected Outcome**:

* The getAccountDetails API should be called with the correct account number.
* The form controls for client\_name, client\_ssn, and master\_rep\_id should be populated with the values from the API response.

**Unit Test: should call getAccountDetails and not populate form controls on failure, and controls should be empty**

**Description**:  
This test verifies that when the getAccountDetailsByAccountNo method is called and the API returns a failure response, the form controls should not be populated with any data. Additionally, it checks that some form controls are disabled or empty as expected, ensuring that the UI remains in a consistent state when the account details retrieval fails.

**Test Steps**:

1. Mock a failure response from the getAccountDetails API (mockResponse) with a status of 'failure'. The mock response includes account details, but these should not be populated in the form.
2. Mock the getAccountDetails API method to resolve with this failure response.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Wait for the asynchronous operations to complete using fixture.whenStable().
5. Verify that the getAccountDetails method was called with the correct account number ('12345').
6. Ensure that the form controls (client\_name, client\_ssn, and master\_rep\_id) remain empty after the API call.
7. Verify that the form control client\_name remains enabled, while client\_ssn and master\_rep\_id are disabled (or hidden, if applicable).

**Expected Outcome**:

* The getAccountDetails API should be called with the correct account number.
* The form controls client\_name, client\_ssn, and master\_rep\_id should have empty values (""), indicating no data was populated due to the failure.
* The client\_name field should remain enabled, while the client\_ssn and master\_rep\_id fields should be disabled, indicating that the fields are hidden or inactive in the event of a failure.

**Unit Test: should reset account details when no client details are returned**

**Description**:  
This test verifies that when the getAccountDetailsByAccountNo method is called and the API returns an empty list of client details (clients\_details\_data), the component correctly resets the account details and clears the relevant form fields. Additionally, it ensures that any temporary data stored in the component (such as tempData) is also cleared, maintaining a clean state.

**Test Steps**:

1. Mock an API response (mockResponse) where clients\_details\_data is an empty array and master\_repid is an empty string, indicating that no client details were found.
2. Mock the getAccountDetails API method to resolve with this empty response.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Wait for the asynchronous operations to complete using fixture.whenStable().
5. Verify that the client\_name, client\_ssn, and master\_rep\_id form controls are reset to empty values ("").
6. Ensure that accountDetails is reset to an empty array ([]), confirming that the component's data was cleared.
7. Verify that tempData is also cleared and reset to an empty array ([]).
8. Optionally, ensure that an appropriate message, such as "No clients found for the given account number!", is logged to the console to inform the user.

**Expected Outcome**:

* The client\_name, client\_ssn, and master\_rep\_id form controls should have empty values (""), indicating that no data was populated due to the absence of client details.
* The accountDetails array should be reset to an empty array ([]), confirming that the component no longer holds any account details.
* The tempData array should also be cleared, ensuring that any temporary data is removed.
* A console log should confirm that no clients were found for the given account number.

**Unit Test: should handle error in getAccountDetailsByAccountNo**

**Description**:  
This test ensures that the component correctly handles errors when the getAccountDetailsByAccountNo method encounters an issue, such as an API failure. It checks if the component calls the appropriate error handling methods, like resetting account details, and if it logs the error message to the console.

**Test Steps**:

1. Mock an error (error) that simulates an API failure when the getAccountDetails method is called. The mockRejectedValueOnce method is used to mock the API call to reject with the error.
2. Create a spy on the resetAccountDetails method to verify that it is called when an error occurs.
3. Mock the console.log method to suppress the actual log output during the test and capture any log calls.
4. Call the getAccountDetailsByAccountNo method in the component.
5. Wait for the asynchronous operations to complete using fixture.whenStable().
6. Verify that the resetAccountDetails method was called, indicating that the component reset its data in response to the error.
7. Ensure that the error message ('Error Occured!') along with the error details is logged to the console.
8. Restore all mocks to their original state using jest.restoreAllMocks() to avoid side effects in other tests.

**Expected Outcome**:

* The resetAccountDetails method should be called, ensuring that any account data or temporary state is cleared when an error occurs.
* The console log should be triggered, displaying the error message 'Error Occured!' followed by the actual error object, confirming that the error has been properly logged.
* Mocks are restored at the end of the test to maintain test isolation

**Unit Test: should not update form controls if master\_rep\_id control is missing**

**Description**:  
This test ensures that when the master\_rep\_id control is removed from the form, it will not be updated, even if the API response contains a value for it. It verifies that the form fields (client\_name, client\_ssn, and master\_rep\_id) are handled correctly, and the master\_rep\_id control is not populated if the control has been removed from the form.

**Test Steps**:

1. Create a mock API response (mockResponse) containing account details, including client\_name, client\_ssn, and master\_repid.
2. Initialize the referralForm with form controls, including master\_rep\_id.
3. Mock the getAccountDetails API method to resolve with the mock response.
4. Call the getAccountDetailsByAccountNo method in the component.
5. Wait for the asynchronous operations to complete using fixture.whenStable().
6. Verify that the master\_rep\_id, client\_name and client\_ssn form controls are populated correctly with the values from the API response ('John Doe' and '123-45-6789').

**Expected Outcome**:

* The form controls master\_rep\_id , client\_name and client\_ssn should be updated with the corresponding values from the API response.

**Unit Test: should reset account details and log when response status is not success (error)**

**Description**:  
This test ensures that when the getAccountDetailsByAccountNo method receives a response with a status other than 'success' (in this case, 'error'), the component correctly resets the account details and clears the form fields. It also verifies that the resetAccountDetails method is called, and the form controls are reset to empty values.

**Test Steps**:

1. Create a mock API response (mockResponse) with a status of 'error' to simulate an unsuccessful account details retrieval. The clients\_details\_data is empty, and master\_repid is an empty string.
2. Spy on the resetAccountDetails method to track if it is called when the response is not successful.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Wait for the asynchronous operations to complete (if necessary).
5. Verify that the resetAccountDetails method was called, indicating that the account details were reset due to the error status.
6. Ensure that the form controls (client\_name, client\_ssn, and master\_rep\_id) are all reset to empty values (""), confirming that the data was cleared as expected.

**Expected Outcome**:

* The resetAccountDetails method should be called, ensuring the component resets its state when the response status is not 'success'.
* The client\_name, client\_ssn, and master\_rep\_id form controls should have empty values (""), indicating that the form has been cleared in response to the error status.

**Unit Test: should set SSN for the selected client**

**Description**:  
This test verifies that when a client is selected from the array list by their name, the corresponding SSN (client\_ssn) is correctly populated in the form. It checks that the client\_ssn form control is updated with the correct value based on the selected client.

**Test Steps**:

1. Create a mock client data (mockClientData) containing client\_name and client\_ssn.
2. Assign this mock client data to the tempData array in the component, simulating that client details are available.
3. Trigger the clientNameChange method with a change event where the value is set to 'John Doe' (the name of the selected client).
4. Verify that the client\_ssn form control is updated with the SSN ('123-45-6789') of the selected client.

**Expected Outcome**:

* The client\_ssn form control should be updated to '123-45-6789' when the client name 'John Doe' is selected, confirming that the correct SSN is set for the selected client.

**Unit Test: should update the client ssn when client name changes**

**Description**:  
This test verifies that when the client name is changed (through a user action such as selecting from a dropdown or an input), the corresponding SSN is updated in the form. It ensures that the client\_ssn field is correctly populated with the SSN value associated with the selected client name.

**Test Steps**:

1. Create a mock event object that simulates a change in the client name to 'John Doe'.
2. Set the tempData array in the component, which contains client details including the name ('John Doe') and SSN ('123-45-6789').
3. Call the clientNameChange method with the mock event.
4. Verify that the client\_ssn form control is updated to the SSN ('123-45-6789') of the selected client.

**Expected Outcome**:

* The client\_ssn form control should be updated to '123-45-6789' after selecting 'John Doe', confirming that the correct SSN is populated when the client name changes.

**Unit Test: should reset client ssn if client name does not match**

**Description**:  
This test verifies that when the client name is changed to one that does not match the current client name in the form, the corresponding SSN is updated correctly. It ensures that the client\_ssn form control is reset and updated to the SSN associated with the newly selected client.

**Test Steps**:

1. Set up tempData with multiple clients' details, each containing a client\_name and client\_ssn.
2. Create a mock event object that simulates a client name change to 'Jane Doe'.
3. Call the clientNameChange method with the mock event, which triggers the change of client name.
4. Verify that the client\_ssn form control is updated with '987-65-4321', the SSN associated with 'Jane Doe', reflecting the change in client name.

**Expected Outcome**:

* The client\_ssn form control should be updated to '987-65-4321' when the client name is changed to 'Jane Doe', ensuring the correct SSN is set based on the selected client.

**Unit Test: should reset client ssn if client name does not match any client**

**Description**:  
This test ensures that if the client name entered or selected does not match any client in the available data (tempData), the client\_ssn field is reset. It checks that when the client name is changed to a non-existent client, the client\_ssn form control is cleared (reset to an empty string).

**Test Steps**:

1. Set up tempData with a list of clients, ensuring only one client ('John Doe') with a valid SSN ('123-45-6789').
2. Simulate a client name change event where the new name is 'Jane Doe', which does not exist in the tempData list.
3. Cast the event to an HTMLSelectElement since the event is triggered by a selection change.
4. Call the clientNameChange method with the simulated event.
5. Assert that the client\_ssn form control is reset to an empty string (""), since the entered client name does not match any available client.

**Expected Outcome**:

* When the client name is changed to 'Jane Doe', which does not match any client in tempData, the client\_ssn field should be reset to an empty string, ensuring that no SSN is associated with the non-existent client.

**Unit Test: should hide master rep id, client name, and client ssn when user do not have access of that account no**

**Description**:  
This test verifies that when the user does not have access to the account (i.e., no client details are returned from the API), certain form fields (like master\_rep\_id, client\_name, and client\_ssn) are hidden. It ensures that the isVisible flag is set to false, indicating that the fields should not be displayed to the user.

**Test Steps**:

1. Create a mock API response with a status of 'success', a master\_repid value ('123'), and an empty clients\_details\_data array, indicating that there are no client details associated with the account.
2. Mock the getAccountDetailsByAccountNo method to return this response.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Assert that the isVisible flag is set to false, indicating that the fields for master\_rep\_id, client\_name, and client\_ssn should be hidden from the UI.

**Expected Outcome**:

1. The isVisible flag should be false, confirming that when no client details are available for the account, the corresponding form fields (master\_rep\_id, client\_name, and client\_ssn) are hidden.

**Unit Test: Should Handle Truthy accountNo (If Branch)**

**Description**

This test verifies that when a truthy accountNo (e.g., '123456789') is provided, the component correctly processes it by setting the account\_no form control value, disabling it to prevent further modifications, and calling the necessary methods to fetch account details, reporter details, and transaction type. Additionally, it ensures that the isVisible flag is set to true, indicating that the relevant data is ready to be displayed.

**Test Steps:**

1. Initialize a truthy accountNo (e.g., '123456789'), simulating the case where a valid account number is provided.
2. Create a spy on the getAccountDetailsByAccountNo, getReporterDetails, and getTransactionType methods to track their calls.
3. If the accountNo is truthy, set the value of the account\_no form control to the provided accountNo and disable it to prevent further modifications.
4. Call the methods getAccountDetailsByAccountNo, getReporterDetails, and getTransactionType to simulate fetching data based on the account number.
5. Set the isVisible flag to true to indicate that the component is ready to display the relevant data.
6. Verify that the account\_no form control has the expected value ('123456789') and is disabled.
7. Ensure that the getAccountDetailsByAccountNo, getReporterDetails, and getTransactionType methods have been called.
8. Check that the isVisible flag is set to true.

**Expected Outcome:**

1. The account\_no form control should have the value '123456789' and be disabled.
2. The methods getAccountDetailsByAccountNo, getReporterDetails, and getTransactionType should have been called.
3. The isVisible flag should be true, confirming that the component is ready to display the data.

**Unit Test: Should Handle Falsy accountNo (Else Branch)**

**Description**

This test verifies the behavior of the component when a falsy accountNo (an empty string) is provided. It ensures that the isVisible flag is set to false, meaning the relevant data is not displayed, and that the account\_no form control is enabled, allowing further interaction. Additionally, it confirms that the enable method is called on the form control.

**Test Steps:**

1. Initialize a falsy accountNo (e.g., ''), simulating the case where the accountNo is not provided or is empty.
2. If the accountNo is falsy, set the isVisible flag to false to hide the component and enable the account\_no form control by calling the enable method.
3. Verify that the isVisible flag is false, indicating that the data is not displayed.
4. Check that the account\_no form control is enabled (i.e., disabled should be false).
5. Ensure the enable method was called on the form control.

**Expected Outcome:**

1. The isVisible flag should be false, confirming that the component does not display the relevant data when the accountNo is falsy.
2. The account\_no form control should be enabled (i.e., disabled should be false).
3. The enable method should have been called, allowing further user interaction with the form control.

**Unit Test: Should Show Pop-up if User Does Not Have Access**

**Description**

This test verifies that when the user does not have access to the account (i.e., the API returns a failure response), the component will display a pop-up and hide the account details. It ensures that the isVisible flag is set to false and the showDialog flag is set to true, indicating that the relevant pop-up should be displayed to inform the user they do not have access.

**Test Steps:**

1. Create a mock API response with a status of 'failure', simulating that the user does not have access to the account.
2. Mock the getAccountDetailsByAccountNo method to return this failure response.
3. Call the getAccountDetailsByAccountNo method in the component.
4. Assert that the isVisible flag is set to false, meaning the account details should not be displayed.
5. Assert that the showDialog flag is set to true, ensuring the pop-up indicating lack of access is shown.

**Expected Outcome:**

1. The isVisible flag should be false, confirming that the account details are hidden when the user does not have access.
2. The showDialog flag should be true, indicating that the pop-up to notify the user they do not have access is displayed.

**Unit Test: Should Call applyPhoneNumberMask Correctly When Formatting Phone Number**

**Description**

This test verifies that the applyPhoneNumberMask method is called correctly when the formatPhoneNumber method is executed. It ensures that the phone number is formatted correctly (e.g., '123-456-7890') and that the applyPhoneNumberMask method receives the expected phone number value.

**Test Steps:**

1. Create a spy on the applyPhoneNumberMask method to track if it is called correctly.
2. Simulate a phone number input event with the value '1234567890'.
3. Call the formatPhoneNumber method, passing the simulated event.
4. Verify that the applyPhoneNumberMask method is called with the phone number '1234567890'.
5. Assert that the phone number is formatted as '123-456-7890' in the event.target.value.

**Expected Outcome:**

1. The applyPhoneNumberMask method should be called with the phone number '1234567890'.
2. The event.target.value should be formatted as '123-456-7890', confirming that the phone number is correctly formatted.

**Unit Test: Should Handle formatPhoneNumberOnBlur Correctly**

**Description**

This test verifies that the formatPhoneNumberOnBlur method correctly formats the phone number when the input field loses focus. It ensures that the applyPhoneNumberMask method is called with the correct phone number, and the phone number is updated to the properly formatted value (e.g., '123-456-7890') using the setValue method.

**Test Steps:**

1. Create a spy on the applyPhoneNumberMask method to track if it is called correctly.
2. Initialize the reporter\_phone\_no form control with a value of '1234567890'.
3. Create a spy on the setValue method of the reporter\_phone\_no form control to check if the value is updated.
4. Call the formatPhoneNumberOnBlur method to simulate the behavior when the input field loses focus.
5. Verify that the applyPhoneNumberMask method is called with the phone number '1234567890'.
6. Check that the setValue method was called with the formatted phone number '123-456-7890'.

**Expected Outcome:**

1. The applyPhoneNumberMask method should be called with the phone number '1234567890'.
2. The setValue method of the reporter\_phone\_no form control should be called with the formatted phone number '123-456-7890', confirming that the phone number is correctly formatted when the input field loses focus.

**Unit Test: Should Format Phone Number with Non-Numeric Characters**

**Description**

This test verifies that the formatPhoneNumber method correctly formats a phone number that contains non-numeric characters (such as parentheses and dashes). It ensures that the method removes the non-numeric characters and formats the phone number in the correct format (e.g., '123-456-7890').

**Test Steps:**

1. Simulate an event where the phone number input contains non-numeric characters, such as '(123) 456-7890'.
2. Call the formatPhoneNumber method, passing the simulated event.
3. Verify that the phone number is correctly formatted as '123-456-7890', with non-numeric characters removed.

**Expected Outcome:**

1. The event.target.value should be formatted as '123-456-7890', confirming that the non-numeric characters were removed and the phone number was correctly formatted.

**Unit Test: Should Not Format Phone Number If Input Exceeds 10 Digits**

**Description**

This test verifies that the formatPhoneNumber method does not alter the phone number if the input exceeds 10 digits. It ensures that phone numbers longer than the standard 10 digits are not formatted or changed.

**Test Steps:**

1. Simulate an event where the phone number input exceeds 10 digits, such as '1234567890123'.
2. Call the formatPhoneNumber method, passing the simulated event.
3. Verify that the phone number remains unchanged and the value is still '1234567890123'.

**Expected Outcome:**

1. The event.target.value should remain '1234567890123', confirming that the phone number was not formatted when it exceeds 10 digits.

**Unit Test: Should Skip Formatting on Blur If Phone Number Is Empty**

**Description**

This test verifies that the formatPhoneNumberOnBlur method does not attempt to format the phone number if the phone number field is empty. It ensures that the setValue method is not called when the form control for the phone number is empty.

**Test Steps:**

1. Initialize the reporter\_phone\_no form control with an empty string (''), simulating an empty phone number field.
2. Create a spy on the setValue method of the reporter\_phone\_no form control to track if it is called.
3. Call the formatPhoneNumberOnBlur method, simulating the behavior when the input field loses focus.
4. Verify that the setValue method was **not** called, confirming that the formatting logic was skipped for an empty phone number.

**Expected Outcome:**

1. The setValue method should **not** be called, confirming that the phone number is not formatted when the input field is empty.

**Unit Test: Should Format Phone Number Correctly on Blur Event**

**Description**

This test verifies that the formatPhoneNumberOnBlur method correctly formats the phone number when the input field loses focus (blur event). It ensures that the applyPhoneNumberMask method is called with the correct phone number and that the phone number is formatted as expected (e.g., '123-456-7890').

**Test Steps:**

1. Set the value of the reporter\_phone\_no form control to '1234567890', simulating a valid phone number.
2. Create a spy on the applyPhoneNumberMask method to track if it is called correctly.
3. Call the formatPhoneNumberOnBlur method to simulate the blur event.
4. Verify that the applyPhoneNumberMask method was called with the phone number '1234567890'.
5. Check that the phoneControl.value is correctly formatted as '123-456-7890'.

**Expected Outcome:**

1. The applyPhoneNumberMask method should be called with the phone number '1234567890'.
2. The phoneControl.value should be formatted as '123-456-7890', confirming that the phone number was correctly formatted on the blur event.

**Unit Test: Should Populate Transaction Types and Subtypes**

**Description**

This test verifies that the getTransactionType method correctly populates the transaction types and subtypes based on the response from the API. It ensures that the transaction types are correctly set in the component's state and that the default value for the transaction\_type form control is set to '--Select Transaction Type--'.

**Test Steps:**

1. Create a mock response for the getTransactionType API call, simulating the presence of transaction types ('type1', 'type2') and their corresponding subtypes ('subtype1', 'subtype2', etc.).
2. Mock the getTransactionType method to return the mock response.
3. Call the getTransactionType method in the component.
4. Verify that the transactionTypes array in the component is populated with the expected values, including '--Select Transaction Type--', 'type1', and 'type2'.
5. Check that the transaction\_type form control has the default value of '--Select Transaction Type--'.

**Expected Outcome:**

1. The transactionTypes array should contain ['--Select Transaction Type--', 'type1', 'type2'], indicating the correct population of transaction types.
2. The transaction\_type form control should have the value '--Select Transaction Type--', confirming that the default value is set correctly.

**Unit Test: Should Reset Subtypes If the Transaction Type Is Not Selected**

**Description**

This test verifies that when the user selects a transaction type, the corresponding subtypes are populated correctly. It ensures that if no transaction type is selected, the subtypes are reset to an empty state. This test ensures the component handles the resetting of subtypes when the transaction type is not selected.

**Test Steps:**

1. Mock a successful response from the API with a valid transaction type ('type1') and its corresponding subtypes ('subtype1', 'subtype2').
2. Simulate selecting a transaction type ('type1').
3. Subscribe to the valueChanges event of the transaction\_type form control and verify that the subtypes are populated.
4. Ensure that the subtypes are correctly updated when a transaction type is selected.

**Expected Outcome:**

1. The transactionSubTypes array should contain the values ['--Select Transaction SubType--', 'subtype1', 'subtype2'] after selecting a transaction type.

**Unit Test: Should Reset Subtypes When "Select Transaction Type" Is Selected**

**Description**

This test ensures that when the transaction type is reset to the default value ('--Select Transaction Type--'), the subtypes are also cleared. It tests the scenario where the user selects the default option for the transaction type, which should reset the subtypes.

**Test Steps:**

1. Mock a successful API response containing a transaction type ('type1') with subtypes ('subtype1', 'subtype2').
2. Simulate the selection of 'type1' and verify that the subtypes are populated.
3. Reset the transaction type to '--Select Transaction Type--' and check if the subtypes are cleared by asserting that the transactionSubTypes is an empty array.

**Expected Outcome:**

1. After resetting the transaction type to '--Select Transaction Type--', the transactionSubTypes array should be empty ([]), indicating the subtypes have been cleared.

**Unit Test: Should Handle the Case Where Transaction Type Has No Subtypes**

**Description**

This test ensures that when a transaction type is selected that has no associated subtypes, the subtypes array is populated with only the default option ('--Select Transaction SubType--'). This test ensures that the component handles transaction types with no subtypes correctly.

**Test Steps:**

1. Mock a successful API response with a transaction type ('type1') that has no subtypes.
2. Simulate selecting 'type1' as the transaction type.
3. Verify that the subtypes array is populated with only the default value ('--Select Transaction SubType--').

**Expected Outcome:**

1. The transactionSubTypes array should contain just the default value ['--Select Transaction SubType--'] when a transaction type with no subtypes is selected.

**Unit Test: Should Reset Form Values and Log Error When API Call Fails**

**Description**

This test verifies that when the API call for fetching transaction types fails, the component resets the form values and logs the error. It ensures that the form values for transaction\_type, transaction\_sub\_type, and other relevant fields are reset to their default values, and the error is logged correctly.

**Test Steps:**

1. Mock an error response from the getTransactionType API.
2. Call the getTransactionType method and simulate the API failure.
3. Verify that the form values for transaction\_type, transaction\_sub\_type, and other\_suspicious\_activity\_details are reset to their default values.
4. Check that the error message is logged to the console.

**Expected Outcome:**

1. The form values should be reset to ['--Select Transaction Type--'] for transaction\_type, ['--Select Transaction SubType--'] for transaction\_sub\_type, and an empty string for other\_suspicious\_activity\_details.
2. The error should be logged to the console using console.log.

**Unit Test: Should Set Default Value for Transaction Type and Sub Type When API Is Successful**

**Description**

This test ensures that when the API call for fetching transaction types succeeds, the default values for transaction\_type and transaction\_sub\_type are set correctly in the form. It verifies that the form controls are initialized with the correct default values upon a successful API response.

**Test Steps:**

1. Mock a successful API response with transaction types ('type1') and subtypes ('subtype1', 'subtype2').
2. Call the getTransactionType method.
3. Verify that the transaction\_type form control has the default value '--Select Transaction Type--'.
4. Verify that the transaction\_sub\_type form control has the default value ''.

**Expected Outcome:**

1. The transaction\_type form control should have the value '--Select Transaction Type--' as the default.
2. The transaction\_sub\_type form control should have the value '', indicating no subtype is selected initially.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to False Initially**

**Description**

This test verifies that the showSeniorInvestorInvolvedFlag is set to false when the selectedTransactionType is initially empty. This ensures the flag starts with the correct value when no transaction type is selected.

**Test Steps:**

1. Set the selectedTransactionType to an empty string.
2. Call the transactionTypeChange method.
3. Verify that the showSeniorInvestorInvolvedFlag is set to false.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be false when no transaction type is selected.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to True if selectedTransactionType is "Electronic Transfer"**

**Description**

This test checks that the showSeniorInvestorInvolvedFlag is set to true when the selectedTransactionType is 'Electronic Transfer'. This verifies that the flag is correctly updated for this specific transaction type.

**Test Steps:**

1. Set the selectedTransactionType to 'Electronic Transfer'.
2. Call the transactionTypeChange method.
3. Verify that the showSeniorInvestorInvolvedFlag is set to true.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be true when the selectedTransactionType is 'Electronic Transfer'.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to True if selectedTransactionType is "Check Fraud"**

**Description**

This test ensures that the showSeniorInvestorInvolvedFlag is set to true when the selectedTransactionType is 'Check Fraud'. It verifies that the flag responds correctly to this transaction type.

**Test Steps:**

1. Set the selectedTransactionType to 'Check Fraud'.
2. Call the transactionTypeChange method.
3. Verify that the showSeniorInvestorInvolvedFlag is set to true.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be true when the selectedTransactionType is 'Check Fraud'.

**Unit Test: Should Keep showSeniorInvestorInvolvedFlag False if selectedTransactionType is Neither "Electronic Transfer" Nor "Check Fraud"**

**Description**

This test ensures that the showSeniorInvestorInvolvedFlag remains false when the selectedTransactionType is neither 'Electronic Transfer' nor 'Check Fraud'. This verifies that the flag is not erroneously set to true for other transaction types.

**Test Steps:**

1. Set the selectedTransactionType to a non-matching type ('Identity Theft').
2. Call the transactionTypeChange method.
3. Verify that the showSeniorInvestorInvolvedFlag remains false.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should remain false when the selectedTransactionType is not 'Electronic Transfer' or 'Check Fraud'.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to True for "Identity Theft" with "Advisor or Client Impersonation"**

**Description**

This test verifies that the showSeniorInvestorInvolvedFlag is set to true when the selectedTransactionType is 'Identity Theft' and the transactionSubType is 'Advisor or Client Impersonation'.

**Test Steps:**

1. Set the selectedTransactionType to 'Identity Theft'.
2. Simulate the change event for the transactionSubType with the value 'Advisor or Client Impersonation'.
3. Verify that the showSeniorInvestorInvolvedFlag is set to true.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be true when the transactionSubType is 'Advisor or Client Impersonation' and the selectedTransactionType is 'Identity Theft'.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to True for "Other Suspicious Activity" with "Elder Financial Exploitation"**

**Description**

This test ensures that the showSeniorInvestorInvolvedFlag is set to true when the selectedTransactionType is 'Other Suspicious Activity' and the transactionSubType is 'Elder Financial Exploitation'.

**Test Steps:**

1. Set the selectedTransactionType to 'Other Suspicious Activity'.
2. Simulate the change event for the transactionSubType with the value 'Elder Financial Exploitation'.
3. Verify that the showSeniorInvestorInvolvedFlag is set to true.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be true when the transactionSubType is 'Elder Financial Exploitation' and the selectedTransactionType is 'Other Suspicious Activity'.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to True for "Other Suspicious Activity" with "Other"**

**Description**

This test ensures that the showSeniorInvestorInvolvedFlag is set to true when the selectedTransactionType is 'Other Suspicious Activity' and the transactionSubType is 'Other'. It verifies that the flag is correctly set for this specific subtype.

**Test Steps:**

1. Set the selectedTransactionType to 'Other Suspicious Activity'.
2. Simulate the change event for the transactionSubType with the value 'Other'.
3. Verify that the showSeniorInvestorInvolvedFlag is set to true.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should be true when the transactionSubType is 'Other' and the selectedTransactionType is 'Other Suspicious Activity'.

**Unit Test: Should Set showSeniorInvestorInvolvedFlag to False When No Conditions Match**

**Description**

This test verifies that the showSeniorInvestorInvolvedFlag is set to false when none of the conditions for setting it to true are met. It checks that the flag is not erroneously set in cases where the selectedTransactionType or transactionSubType do not match the predefined values.

**Test Steps:**

1. Set the selectedTransactionType to 'Non-matching Type'.
2. Simulate the change event for the transactionSubType with a random value.
3. Verify that the showSeniorInvestorInvolvedFlag remains false.

**Expected Outcome:**

1. The showSeniorInvestorInvolvedFlag should remain false when no matching conditions are found for selectedTransactionType or transactionSubType.

**Unit Test: Should Reset Account Details When Account Number is Empty**

**Description**

This test verifies that the accountDetails and clientNames are reset to empty arrays when the account number is cleared. It ensures the correct behavior when there is no account number.

**Test Steps:**

1. Set the account\_no field in the form to an empty string.
2. Call the getAccountDetailsByAccountNo() method.
3. Verify that accountDetails and clientNames are both set to empty arrays.

**Expected Outcome:**

1. The accountDetails and clientNames should both be empty when the account number is empty.

**Unit Test: Should Handle API Errors Correctly**

**Description**

This test ensures that the component handles errors from the API correctly. It verifies that accountDetails remains empty when the API call fails.

**Test Steps:**

1. Mock the getAccountDetails API method to reject with an error.
2. Call getAccountDetailsByAccountNo().
3. Verify that accountDetails is reset to an empty array after the error is thrown.

**Expected Outcome:**

1. The accountDetails should remain an empty array if the API call fails.

**Unit Test: Should Submit the Refferal Form and Reset on Success**

**Description**

This test ensures that the form is submitted correctly and reset after successful submission. It checks that the form values are cleared after a successful form submission.

**Test Steps:**

1. Set the form values for referralForm to mock data.
2. Enable the form controls for submission.
3. Mock successful API responses for all necessary API calls (submit form, get reporter details, transaction types, and account details).
4. Call submitReferralForm() and verify that the form data is passed to the API.
5. Verify that the form values are reset to their default state after the form submission.

**Expected Outcome:**

1. The form data should be submitted to the API.
2. After submission, the form fields should be reset to their initial state.

**Unit Test: Should Handle Error Response and Log Message**

**Description**

This test ensures that when an error response is received, the form is reset and an error message is logged.

**Test Steps:**

1. Set up a mock error response with status 'error'.
2. Check if resetReferralForm() is called upon receiving an error response.
3. Ensure the error message is logged using console.log().

**Expected Outcome:**

1. resetReferralForm() should be called, and the error message should be logged when an error response is received.

**Unit Test: Should Not Submit Form if Invalid**

**Description**

This test verifies that the form is not submitted if it is invalid. It checks that the form’s invalid state is respected during the submission process.

**Test Steps:**

1. Set the form values with invalid data (missing required fields).
2. Call the submitReferralForm() method.
3. Verify that the form is invalid after submission.

**Expected Outcome:**

1. The form should remain invalid, and no form submission should occur if the form is not filled out correctly.

**Unit Test: Should Set the Correct Value for trades\_placed\_flag**

**Description**

This test ensures that the value of the trades\_placed\_flag form field is correctly set when the form is submitted.

**Test Steps:**

1. Set the value of the trades\_placed\_flag field to 'Y'.
2. Call submitReferralForm().
3. Verify that the form field trades\_placed\_flag retains the value 'Y' after submission.

**Expected Outcome:**

1. The form's trades\_placed\_flag field should hold the value 'Y' after submission.

**Unit Test: Should Reset the Form Correctly**

**Description**

This test ensures that the form is correctly reset after calling resetReferralForm().

**Test Steps:**

1. Call the resetReferralForm() method.
2. Verify that the account\_no field value is reset to an empty string.
3. Check that the submitted flag is reset to false.

**Expected Outcome:**

1. After calling resetReferralForm(), the form values should be reset, and the submitted flag should be set to false.

**Unit Test: Should Log Invalid Controls if Form is Invalid**

**Description**

This test verifies that invalid form controls are logged and that a general invalidity message is logged when the form is invalid.

**Test Steps:**

1. Set the form values to invalid values (empty or null where necessary).
2. Spy on console.log() to capture the output.
3. Call submitReferralForm().
4. Verify that console.log() was called with the list of invalid controls and a general invalidity message.

**Expected Outcome:**

1. Invalid form controls should be logged.
2. A general message like 'Form is invalid. Please correct the errors and try again.' should also be logged.

**Unit Test: Should Handle API Failure and Show Error Message**

**Description**

This test ensures that the component handles an API failure correctly, logging the error message when the API request fails.

**Test Steps:**

1. Set the form to a valid state.
2. Mock the submitReferralForm method to reject with an error.
3. Spy on console.log() to capture the error message.
4. Call submitReferralForm().
5. Verify that the error message is logged in the console.

**Expected Outcome:**

1. When the API fails, the error should be logged with the message 'Error Occurred!'.

**Unit Test: Should Enable Form Controls Before Submission**

**Description**

This test ensures that certain form controls are enabled before the form is submitted, verifying that the correct controls are enabled during the submission process.

**Test Steps:**

1. Set the form to valid values.
2. Mock the enable() method for relevant form controls.
3. Call submitReferralForm().
4. Verify that the enable() method is called on the relevant form controls.

**Expected Outcome:**

1. The enable() method should be called on the form controls that need to be enabled before submission (e.g., master\_rep\_id, client\_ssn, etc.).

**Unit Test: onKeyDown - Should Not Call btnReset on Other Key Presses**

**Description**

This test verifies that pressing keys other than the ones meant to trigger the btnReset method (like the "A" key) does not trigger the reset functionality.

**Test Steps:**

1. Create a keyboard event for the "A" key (keyCode 65).
2. Spy on the btnReset method and preventDefault() method.
3. Call onKeyDown() with the keyboard event.
4. Verify that preventDefault() is not called, and btnReset is not triggered.

**Expected Outcome:**

1. The btnReset method should not be called, and the default behavior should not be prevented when a non-reset key is pressed.

**Unit Test: Should Navigate to the Root Path and Call Reset Methods in btnReset**

**Description**  
This test ensures that the btnReset method in the component correctly navigates to the root path and resets both the account details and the referral form.

**Test Steps:**

1. **Spying on methods**: Spy on the router.navigate, resetAccountDetails, and resetReferralForm methods to capture their calls.
2. **Calling btnReset**: Invoke the btnReset() method to simulate the reset action.
3. **Assertions**:
   * Verify that router.navigate is called with the argument [''] to navigate to the root path.
   * Ensure that resetAccountDetails is called to clear the account details.
   * Check that resetReferralForm is called to reset the referral form.

**Expected Outcome:** When the btnReset method is called, it should:

1. Navigate to the root path (['']).
2. Call the resetAccountDetails method.
3. Call the resetReferralForm method.

**Unit Test: Should Call router.navigate and Reset Form and Account Details**

**Description**  
This test ensures that the btnReset method in the component calls the router.navigate method to navigate to the root path and invokes both the resetAccountDetails and resetReferralForm methods to reset the account details and referral form.

**Test Steps:**

1. **Spying on methods**: Use jest.spyOn to spy on the navigate method of the router, resetAccountDetails, and resetReferralForm methods.
2. **Calling btnReset**: Invoke the btnReset() method to simulate the reset action.
3. **Assertions**:
   * Verify that navigate was called with the argument [''], ensuring navigation to the root path.
   * Ensure that resetAccountDetails is called to clear the account details.
   * Ensure that resetReferralForm is called to reset the form.

**Expected Outcome:** When the btnReset method is called, it should:

1. Call router.navigate with the argument [''] to navigate to the root path.
2. Call the resetAccountDetails method to reset the account details.
3. Call the resetReferralForm method to reset the referral form.

Top of Form

**Unit Test: Should Navigate to the Root Path and Call Reset Methods in btnReset**

**Description**  
This test ensures that the btnReset method behaves as expected, navigating to the root path and calling the resetAccountDetails and resetReferralForm methods to reset the form and account details.

**Test Steps:**

1. **Spying on methods**: Mock the resetAccountDetails and resetReferralForm methods using jest.fn().
2. **Calling btnReset**: Invoke the btnReset() method to simulate the reset action.
3. **Assertions**:
   * Verify that mockRouter.navigate was called with the argument [''], ensuring navigation to the root path.
   * Check that the resetAccountDetails method was called to reset account details.
   * Check that the resetReferralForm method was called to reset the referral form.

**Expected Outcome:** When the btnReset method is called:

1. The router.navigate method should be called with [''] to navigate to the root path.
2. The resetAccountDetails method should be invoked to reset the account details.
3. The resetReferralForm method should be invoked to reset the form.

**Unit Test: resetAccountDetails - Should Reset Account Details and Form Values Correctly**

**Description**  
This test ensures that the resetAccountDetails method correctly resets the account details and form values to their initial state.

**Test Steps:**

1. Call the resetAccountDetails() method on the component.
2. Verify that the accountDetails property is reset to an empty array ([]).
3. Verify that the clientNames property is reset to an empty array ([]).
4. Verify that the selectedClientName property is reset to an empty string ('').
5. Verify that the form controls client\_name, client\_ssn, and master\_rep\_id are cleared and reset to empty strings ('').

**Expected Outcome:**

1. The accountDetails property should be reset to an empty array ([]).
2. The clientNames property should be reset to an empty array ([]).
3. The selectedClientName property should be reset to an empty string ('').
4. The form controls client\_name, client\_ssn, and master\_rep\_id should be cleared and set to empty strings ('').

**Unit Test: resetReferralForm - Should Reset Referral Form and Call Necessary API Methods**

**Description**  
This test verifies that the resetReferralForm method correctly resets the referral form values to their initial state and calls the necessary API methods (getReporterDetails and getTransactionType).

**Test Steps:**

1. Spy on the getReporterDetails and getTransactionType methods to track their calls.
2. Call the resetReferralForm() method on the component.
3. Verify that all form controls (account\_no, master\_rep\_id, client\_name, client\_ssn, reporter\_first\_name, reporter\_last\_name, reporter\_email, reporter\_phone\_no, transaction\_type, transaction\_sub\_type, trades\_placed\_flag, other\_suspicious\_activity\_details, additional\_transaction\_details) are reset to their initial values (empty strings or null as applicable).
4. Verify that the submitted flag is set to false.
5. Verify that the showSeniorInvestorInvolvedFlag is set to false.
6. Verify that the isVisible flag is set to false.
7. Verify that the getReporterDetails and getTransactionType methods have been called.

**Expected Outcome:**

1. The form controls should be reset to their initial values:
   * account\_no, master\_rep\_id, client\_name, client\_ssn, reporter\_first\_name, reporter\_last\_name, reporter\_email, reporter\_phone\_no, transaction\_type, transaction\_sub\_type, other\_suspicious\_activity\_details, and additional\_transaction\_details should be empty strings.
   * trades\_placed\_flag should be null.
2. The submitted flag should be false.
3. The showSeniorInvestorInvolvedFlag should be false.
4. The isVisible flag should be false.
5. The getReporterDetails method should be called.
6. The getTransactionType method should be called.

**Unit Test: onConfirm - Should Set showDialog to False**

**Description**  
This test verifies that the onConfirm method correctly sets the showDialog property to false.

**Test Steps:**

1. Set the showDialog property of the component to true.
2. Call the onConfirm() method on the component.
3. Verify that the showDialog property is set to false after calling the method.

**Expected Outcome:**

1. The showDialog property should be set to false after calling the onConfirm method.

**Unit Test: onCloseDialog - Should Set showReferenceNumber to False**

**Description**  
This test verifies that the onCloseDialog method correctly sets the showReferenceNumber property to false when the dialog is closed.

**Test Steps:**

1. Set the showReferenceNumber property of the component to true to simulate the reference number dialog being open.
2. Call the onCloseDialog() method on the component.
3. Verify that the showReferenceNumber property is set to false after calling the method.

**Expected Outcome:**

1. The showReferenceNumber property should be set to false after calling the onCloseDialog method, indicating that the dialog has been closed.

**Unit Test: toggleClientNameValidation - Should Make client\_name Required Based on User Access**

**Description**  
This test ensures that the client\_name field's validation is updated dynamically based on the user’s access to the account number. When the user has access, the field should be required, and when the user does not have access, the field should not be required.

**Test 1: Should Make client\_name Required When User Has Access**

**Test Steps:**

1. Call the toggleClientNameValidation(true) method to simulate the scenario where the user has access to the account number.
2. Retrieve the client\_name form control from the referral form.
3. Check if the client\_name field has the Validators.required validator applied.

**Expected Outcome:**

1. The client\_name field should have the Validators.required validator applied, making the field required.

**Unit Test: toggleClientNameValidation - Should Make client\_name Required When User Has Access**

**Description**  
This test ensures that the client\_name field is marked as required when the user has access to the corresponding account number.

**Test Steps:**

1. Call the toggleClientNameValidation(true) method to simulate the scenario where the user has access to the account number.
2. Retrieve the client\_name form control from the referralForm.
3. Verify if the client\_name field has the Validators.required validator applied.

**Expected Outcome:**

1. The client\_name field should have the Validators.required validator, meaning that the field is required when the user has access to the account number.

**Unit Test: toggleClientNameValidation - Should Not Make client\_name Required When User Does Not Have Access**

**Description**  
This test ensures that the client\_name field is not marked as required when the user does not have access to the corresponding account number.

**Test Steps:**

1. Call the toggleClientNameValidation(false) method to simulate the scenario where the user does not have access to the account number.
2. Retrieve the client\_name form control from the referralForm.
3. Verify if the client\_name field does not have the Validators.required validator applied.

**Expected Outcome:**

1. The client\_name field should not have the Validators.required validator, meaning that the field is not required when the user does not have access to the account number.

**Unit Test: toggleClientNameValidation - Should Update Validation Dynamically Based on isVisible**

**Description**  
This test verifies that the client\_name field's validation is updated dynamically based on the isVisible flag. Specifically, it ensures that when isVisible is true, the field becomes required, and when isVisible is false, the field is not required.

**Test Steps:**

1. Retrieve the client\_name form control from the referralForm.
2. Set isVisible to false, and call toggleClientNameValidation(false).
3. Set the client\_name field value to an empty string ('') and verify if the field is valid (it should be valid since it is not required).
4. Set isVisible to true, and call toggleClientNameValidation(true).
5. Set the client\_name field value to an empty string ('') and verify if the field is invalid (it should be invalid because it is now required).
6. Set a valid value ('John Doe') for the client\_name field and verify if the field is valid.

**Expected Outcome:**

1. When isVisible is false, the client\_name field should not be required, and it should be valid even with an empty value.
2. When isVisible is true, the client\_name field should be required, and it should be invalid with an empty value.
3. The client\_name field should be valid when a valid value is entered (e.g., 'John Doe').

**API Unit Test**

**Unit Test: getAccountDetails - Should Call Rest.get with the Correct URL and Return Transformed Account Details**

**Description**  
This test ensures that the getAccountDetails method in the service correctly calls Rest.get with the proper URL and returns the expected transformed account details.

**Test Steps:**

1. **Prepare the Test Data:**
   * Define a mock accountNumber ('65497603').
   * Define a mock response that simulates the data returned from the API. This response includes account details and client information.
2. **Mock the Rest.get method:**
   * Use jest.mockResolvedValue to simulate the API response from Rest.get for the getAccountDetails method.
3. **Call the getAccountDetails method:**
   * Pass the accountNumber to service.getAccountDetails(accountNumber).
4. **Verify the URL Called by Rest.get:**
   * Use expect to verify that Rest.get was called with the correct URL, which is constructed using the accountNumber and the API endpoint ${environment.ReferralAPI}account-details/${accountNumber}.
5. **Verify the Transformed Response:**
   * Ensure that the result returned from getAccountDetails matches the expected transformed response, which should exclude unnecessary data and return the relevant account details.

**Expected Outcome:**

1. The Rest.get method should be called with the correct URL, including the accountNumber.
2. The method should return the transformed account details that include the master\_repid and clients\_details\_data fields, which are extracted and returned in the correct format.

**Unit Test: getReporterDetails - Should Call Rest.get with the Correct URL and Return Reporter Details**

**Description**  
This test ensures that the getReporterDetails method in the service correctly calls Rest.get with the appropriate URL and returns the transformed reporter details.

**Test Steps:**

1. **Prepare the Test Data:**
   * Define a mock response for the API, simulating the data structure returned when calling the getReporterDetails API endpoint. The mock data includes reporter details such as first name, last name, and email address.
2. **Mock the Rest.get method:**
   * Use jest.mockResolvedValue to simulate the API response from Rest.get for the getReporterDetails method.
3. **Call the getReporterDetails method:**
   * Call service.getReporterDetails() to invoke the service method.
4. **Verify the URL Called by Rest.get:**
   * Use expect to verify that Rest.get was called with the correct URL: ${environment.ReferralAPI}reporter-details.
5. **Verify the Transformed Response:**
   * Ensure that the result returned from getReporterDetails matches the expected transformed structure, with reporter\_first\_name, reporter\_last\_name, and reporter\_email fields mapped correctly.

**Expected Outcome:**

1. The Rest.get method should be called with the correct URL, which is ${environment.ReferralAPI}reporter-details.
2. The method should return the reporter details with the fields correctly mapped as:
   * reporter\_first\_name → 'John'
   * reporter\_last\_name → 'Doe'
   * reporter\_email → 'john.doe@lplfinancial.com'

**Unit Test: getTransactionType - Should Call Rest.get with the Correct URL and Return Transaction Types**

**Description**  
This test ensures that the getTransactionType method in the service correctly calls Rest.get with the appropriate URL and returns the transformed transaction types data.

**Test Steps:**

1. **Prepare the Test Data:**
   * Define a mock response that simulates the structure of the API response. The mock response contains the transaction\_types\_info object with categories of transaction types and their corresponding subtypes.
2. **Mock the Rest.get method:**
   * Use jest.mockResolvedValue to simulate the successful response from Rest.get for the getTransactionType method.
3. **Call the getTransactionType method:**
   * Call service.getTransactionType() to invoke the service method.
4. **Verify the URL Called by Rest.get:**
   * Use expect to verify that Rest.get was called with the correct URL: ${environment.ReferralAPI}transaction-types.
5. **Verify the Transformed Response:**
   * Ensure that the result returned from getTransactionType matches the expected transformed structure with the transactionType field correctly containing the transaction types information as structured in the mock response.

**Expected Outcome:**

1. The Rest.get method should be called with the correct URL, which is ${environment.ReferralAPI}transaction-types.
2. The method should return the transaction types structured as follows:
   * The key transactionType should contain a transaction\_types\_info object.
   * The transaction\_types\_info object should map categories (e.g., "Electronic Transfer", "Check Fraud", etc.) to arrays of transaction subtypes (e.g., "ACH Transaction", "Wire Transaction", etc.).

**Unit Test: submitReferralForm - Should Call Rest.post with the Correct URL and Return Case Details Status**

**Description**  
This test ensures that the submitReferralForm method calls Rest.post with the correct URL and successfully processes the response, returning the expected case details status and reference number.

**Test Steps:**

1. **Prepare the Test Data:**
   * Create an instance of ICaseDetailsModelRequest with valid form data for submitting a referral form.
2. **Mock the Rest.post Method:**
   * Use jest.mockResolvedValue to mock a successful API response containing the status and reference number.
3. **Call the submitReferralForm Method:**
   * Call service.submitReferralForm(formData) to invoke the method.
4. **Verify the URL and Form Data:**
   * Use expect to check that Rest.post was called with the correct URL: ${environment.ReferralAPI}case-details and the form data.
5. **Verify the Response:**
   * Ensure that the response returned from submitReferralForm contains the correct status ("success") and reference\_number (e.g., 126).

**Expected Outcome:**

1. The Rest.post method should be called with the correct URL and the form data as the body of the request.
2. The result returned from submitReferralForm should match the transformed structure:
   * The status should be "success".
   * The reference\_number should be 126.

**Unit Test: submitReferralForm - Should Handle Invalid Response Structure (Missing Data or Data.data)**

**Description**  
This test ensures that the submitReferralForm method handles cases where the response structure is invalid or missing expected data.

**Test Steps:**

1. **Prepare the Test Data:**
   * Create the same form data as in the previous test.
2. **Mock the Invalid Response:**
   * Mock an invalid response structure (e.g., missing data or data.data).
3. **Call the submitReferralForm Method:**
   * Call service.submitReferralForm(formData) to invoke the method.
4. **Verify the URL and Form Data:**
   * Ensure that Rest.post was called with the correct URL and form data.
5. **Verify the Result:**
   * Ensure that if the response structure is invalid, the method returns default values (status: '', reference\_number: '').

**Expected Outcome:**

1. The method should return default values (status: '', reference\_number: '') when the response structure is invalid or missing expected fields.

**Unit Test: submitReferralForm - Should Handle Empty Response (Undefined or Null)**

**Description**  
This test ensures that the submitReferralForm method gracefully handles cases where the API response is empty (null or undefined).

**Test Steps:**

1. **Prepare the Test Data:**
   * Create the form data as in the previous tests.
2. **Mock the Empty Response:**
   * Mock an empty response (null) using mockRestPost.mockResolvedValue(null).
3. **Call the submitReferralForm Method:**
   * Call service.submitReferralForm(formData).
4. **Verify the URL and Form Data:**
   * Check that Rest.post was called with the correct URL and the form data.
5. **Verify the Result:**
   * Ensure that the result is handled properly and returns default values (status: '', reference\_number: '') when the response is null or undefined.

**Expected Outcome:**

1. The method should return default values (status: '', reference\_number: '') when the response is empty (null or undefined).

**Unit Test: submitReferralForm - Should Handle Error Response Gracefully**

**Description**  
This test ensures that the submitReferralForm method handles error responses, such as network errors, gracefully.

**Test Steps:**

1. **Prepare the Test Data:**
   * Create the form data for submitting the referral form.
2. **Mock an Error Response:**
   * Mock a rejected Rest.post response with an error (e.g., new Error('Network error')).
3. **Call the submitReferralForm Method:**
   * Call service.submitReferralForm(formData) and handle the error.
4. **Verify the Error Handling:**
   * Use try-catch to catch the error and ensure that it is an instance of Error.

**Expected Outcome:**

1. The method should handle the error gracefully, and the error should be an instance of Error (with the message "Network error"). The test should ensure no unhandled exceptions occur.

**Validator Unit Test**

**Unit Test: lplAccountValidator - Should Return Null for a Valid Account**

**Description**  
This test ensures that the lplAccountValidator function returns null when a valid account number is provided. A valid account number is alphanumeric and has 10 or fewer characters.

**Test Steps:**

1. **Prepare a Valid Account Number:**
   * Create a mock control with a valid alphanumeric account number ('Valid123').
2. **Call the lplAccountValidator Method:**
   * Call ReferralCustomValidators.lplAccountValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null for the valid account number.

**Expected Outcome:**

1. The validator should return null for a valid account, indicating no errors.

**Unit Test: lplAccountValidator - Should Return Error Object for an Invalid Account (More Than 10 Characters)**

**Description**  
This test ensures that the lplAccountValidator function returns an error object when the account number exceeds 10 characters.

**Test Steps:**

1. **Prepare an Invalid Account Number:**
   * Create a mock control with an invalid account number longer than 10 characters ('InvalidAccount123').
2. **Call the lplAccountValidator Method:**
   * Call ReferralCustomValidators.lplAccountValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidAccount: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidAccount: true } for account numbers longer than 10 characters.

**Unit Test: lplAccountValidator - Should Return Error Object for an Invalid Account (Non-Alphanumeric Characters)**

**Description**  
This test ensures that the lplAccountValidator function returns an error object when the account number contains non-alphanumeric characters.

**Test Steps:**

1. **Prepare an Invalid Account Number:**
   * Create a mock control with an account number that contains special characters ('Invalid@123').
2. **Call the lplAccountValidator Method:**
   * Call ReferralCustomValidators.lplAccountValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidAccount: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidAccount: true } for account numbers containing non-alphanumeric characters.

**Unit Test: lplAccountValidator - Should Return Error Object for an Empty Account**

**Description**  
This test ensures that the lplAccountValidator function returns an error object when the account number is empty.

**Test Steps:**

1. **Prepare an Empty Account Number:**
   * Create a mock control with an empty account number ('').
2. **Call the lplAccountValidator Method:**
   * Call ReferralCustomValidators.lplAccountValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidAccount: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidAccount: true } when the account number is empty.

**Unit Test: phoneValidator - Should Return Null for a Valid Phone Number**

**Description**  
This test ensures that the phoneValidator function returns null when a valid phone number is provided. A valid phone number should be in the format xxx-xxx-xxxx where x represents a digit.

**Test Steps:**

1. **Prepare a Valid Phone Number:**
   * Create a mock control with a valid phone number ('123-456-7890').
2. **Call the phoneValidator Method:**
   * Call ReferralCustomValidators.phoneValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null for the valid phone number.

**Expected Outcome:**

1. The validator should return null for a valid phone number, indicating no errors.

**Unit Test: phoneValidator - Should Return Error Object for an Invalid Phone Number (Too Few Digits)**

**Description**  
This test ensures that the phoneValidator function returns an error object when the phone number has too few digits (less than 10 digits).

**Test Steps:**

1. **Prepare an Invalid Phone Number (Too Few Digits):**
   * Create a mock control with an invalid phone number ('123-45-789').
2. **Call the phoneValidator Method:**
   * Call ReferralCustomValidators.phoneValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidPhone: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidPhone: true } for phone numbers that have too few digits.

**Unit Test: phoneValidator - Should Return Error Object for an Invalid Phone Number (Wrong Format)**

**Description**  
This test ensures that the phoneValidator function returns an error object when the phone number is in the wrong format (e.g., using slashes instead of dashes).

**Test Steps:**

1. **Prepare an Invalid Phone Number (Wrong Format):**
   * Create a mock control with an invalid phone number ('123/456/7890').
2. **Call the phoneValidator Method:**
   * Call ReferralCustomValidators.phoneValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidPhone: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidPhone: true } for phone numbers that use an invalid format.

**Unit Test: phoneValidator - Should Return Error Object for an Empty Phone Number**

**Description**  
This test ensures that the phoneValidator function returns an error object when the phone number is empty.

**Test Steps:**

1. **Prepare an Empty Phone Number:**
   * Create a mock control with an empty phone number ('').
2. **Call the phoneValidator Method:**
   * Call ReferralCustomValidators.phoneValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidPhone: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidPhone: true } when the phone number is empty.

**Unit Test: emailValidator - Should Return Null for a Valid Email Address with @lplfinancial.com**

**Description**  
This test ensures that the emailValidator function returns null when the provided email address is valid and ends with @lplfinancial.com.

**Test Steps:**

1. **Prepare a Valid Email Address:**
   * Create a mock control with a valid email address ('valid@lplfinancial.com').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating the email is valid.

**Expected Outcome:**

1. The validator should return null for a valid email address ending with @lplfinancial.com.

**Unit Test: emailValidator - Should Return Null for a Valid Email Address with @lpl.com**

**Description**  
This test ensures that the emailValidator function returns null when the provided email address is valid and ends with @lpl.com.

**Test Steps:**

1. **Prepare a Valid Email Address:**
   * Create a mock control with a valid email address ('valid@lpl.com').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating the email is valid.

**Expected Outcome:**

1. The validator should return null for a valid email address ending with @lpl.com.

**Unit Test: emailValidator - Should Return Error Object for an Invalid Email Address with a Wrong Domain**

**Description**  
This test ensures that the emailValidator function returns an error object when the provided email address has a wrong domain (e.g., @google.com).

**Test Steps:**

1. **Prepare an Invalid Email Address:**
   * Create a mock control with an invalid email address ('invalid@google.com').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidEmail: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidEmail: true } for email addresses with a wrong domain.

**Unit Test: emailValidator - Should Return Error Object for an Empty Email Address**

**Description**  
This test ensures that the emailValidator function returns an error object when the provided email address is empty.

**Test Steps:**

1. **Prepare an Empty Email Address:**
   * Create a mock control with an empty email address ('').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidEmail: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidEmail: true } when the email address is empty.

**Unit Test: emailValidator - Should Return Error Object for an Email Address with a Missing Domain Part**

**Description**  
This test ensures that the emailValidator function returns an error object when the provided email address has a missing domain part (e.g., invalid@.com).

**Test Steps:**

1. **Prepare an Invalid Email Address with a Missing Domain:**
   * Create a mock control with an invalid email address ('invalid@.com').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidEmail: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidEmail: true } when the email address has a missing domain part.

**Unit Test: emailValidator - Should Return Error Object for an Email Address with an Invalid Domain (e.g., lpl.financial)**

**Description**  
This test ensures that the emailValidator function returns an error object when the provided email address has an invalid domain (e.g., @lpl.financial).

**Test Steps:**

1. **Prepare an Invalid Email Address with an Invalid Domain:**
   * Create a mock control with an invalid email address ('invalid@lpl.financial').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidEmail: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidEmail: true } for email addresses with invalid domain parts.

**Unit Test: emailValidator - Should Return Error Object for an Email Address with Extra Characters (e.g., test@@lplfinancial.com)**

**Description**  
This test ensures that the emailValidator function returns an error object when the provided email address has extra characters (e.g., multiple @ symbols).

**Test Steps:**

1. **Prepare an Invalid Email Address with Extra Characters:**
   * Create a mock control with an invalid email address ('test@@lplfinancial.com').
2. **Call the emailValidator Method:**
   * Call ReferralCustomValidators.emailValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is an error object { invalidEmail: true }.

**Expected Outcome:**

1. The validator should return the error object { invalidEmail: true } for email addresses with extra characters.

**Unit Test: otherDetailsValidator - Should Return null for Valid Other Details**

**Description:**  
This test checks if the otherDetailsValidator function returns null for valid other details (a string that contains alphanumeric characters and is not too long).

**Test Steps:**

1. **Prepare Valid Input:**
   * Create a mock control with valid other details: 'Valid details 123'.
2. **Call the otherDetailsValidator Method:**
   * Call ReferralCustomValidators.otherDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating the input is valid.

**Expected Outcome:**

* The validator should return null for valid details.

**Unit Test: otherDetailsValidator - Should Return Error Object for Invalid Other Details (Too Long)**

**Description:**  
This test checks if the otherDetailsValidator function returns an error object when the input is too long (more than 50 characters).

**Test Steps:**

1. **Prepare Invalid Input (Too Long):**
   * Create a mock control with an overly long string: 'This is a very long string that should be invalid because it exceeds fifty characters for the test.'.
2. **Call the otherDetailsValidator Method:**
   * Call ReferralCustomValidators.otherDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is the error object { invalidOtherDetails: true }.

**Expected Outcome:**

* The validator should return the error object { invalidOtherDetails: true } for input that exceeds 50 characters.

**Unit Test: otherDetailsValidator - Should Return Error Object for Invalid Other Details (Non-Alphanumeric Characters)**

**Description:**  
This test checks if the otherDetailsValidator function returns an error object when the input contains non-alphanumeric characters (e.g., punctuation marks).

**Test Steps:**

1. **Prepare Invalid Input (Non-Alphanumeric):**
   * Create a mock control with invalid input containing special characters: 'Invalid!@#details'.
2. **Call the otherDetailsValidator Method:**
   * Call ReferralCustomValidators.otherDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is the error object { invalidOtherDetails: true }.

**Expected Outcome:**

* The validator should return the error object { invalidOtherDetails: true } for input containing non-alphanumeric characters.

**Unit Test: otherDetailsValidator - Should Return null for an Empty Value**

**Description:**  
This test checks if the otherDetailsValidator function returns null for an empty value, which indicates that empty input is considered valid.

**Test Steps:**

1. **Prepare Empty Input:**
   * Create a mock control with an empty string: ''.
2. **Call the otherDetailsValidator Method:**
   * Call ReferralCustomValidators.otherDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating that empty input is valid.

**Expected Outcome:**

* The validator should return null for an empty value.

**Unit Test: additionalDetailsValidator - Should Return null for Valid Additional Details**

**Description:**  
This test checks if the additionalDetailsValidator function returns null for valid input (up to 2000 characters and alphanumeric).

**Test Steps:**

1. **Prepare Valid Input:**
   * Create a mock control with valid additional details: 'Valid additional details up to 2000 characters.'.
2. **Call the additionalDetailsValidator Method:**
   * Call ReferralCustomValidators.additionalDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating the input is valid.

**Expected Outcome:**

* The validator should return null for valid additional details.

**Unit Test: additionalDetailsValidator - Should Return Error Object for Invalid Additional Details (Too Long)**

**Description:**  
This test checks if the additionalDetailsValidator function returns an error object when the input exceeds the character limit of 2000.

**Test Steps:**

1. **Prepare Invalid Input (Too Long):**
   * Create a mock control with a string longer than 2000 characters (e.g., 'A'.repeat(2001)).
2. **Call the additionalDetailsValidator Method:**
   * Call ReferralCustomValidators.additionalDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is the error object { invalidAdditionalDetails: true }.

**Expected Outcome:**

* The validator should return { invalidAdditionalDetails: true } for input longer than 2000 characters.

**Unit Test: additionalDetailsValidator - Should Return Error Object for Invalid Additional Details (Non-Alphanumeric Characters)**

**Description:**  
This test checks if the additionalDetailsValidator function returns an error object when the input contains non-alphanumeric characters (such as symbols or punctuation marks).

**Test Steps:**

1. **Prepare Invalid Input (Non-Alphanumeric):**
   * Create a mock control with invalid input containing special characters: 'Invalid additional details @!#'.
2. **Call the additionalDetailsValidator Method:**
   * Call ReferralCustomValidators.additionalDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is the error object { invalidAdditionalDetails: true }.

**Expected Outcome:**

* The validator should return { invalidAdditionalDetails: true } for input containing non-alphanumeric characters.

**Unit Test: additionalDetailsValidator - Should Return null for an Empty Value**

**Description:**  
This test checks if the additionalDetailsValidator function returns null for an empty value, indicating that empty input is considered valid.

**Test Steps:**

1. **Prepare Empty Input:**
   * Create a mock control with an empty string: ''.
2. **Call the additionalDetailsValidator Method:**
   * Call ReferralCustomValidators.additionalDetailsValidator(control) with the mock control.
3. **Verify the Result:**
   * Ensure that the result is null, indicating that empty input is valid.

**Expected Outcome:**

* The validator should return null for an empty value.

**App Component Unit Test**

**1. Test: Should Create the App**

**Description:**  
This test checks if the component is created successfully.

**Test Steps:**

1. Retrieve the component instance using fixture.componentInstance.
2. Use the expect function to verify that the component instance is truthy (i.e., the component is successfully created).

**Expected Outcome:**

* The component should be created without issues, and the test should pass.

**2. Test: Should Have Correct Title**

**Description:**  
This test checks if the title property of the component is set correctly.

**Test Steps:**

1. Retrieve the component instance using fixture.componentInstance.
2. Use expects to check that the title property is equal to 'report-unusual-activity'.

**Expected Outcome:**

* The title property should be 'report-unusual-activity', and the test should pass.

**3. Test: Should Call rest.config and LwkThemes.setTheme on ngOnInit**

**Description:**  
This test checks if the rest.config and LwkThemes.setTheme methods are called correctly during the ngOnInit lifecycle hook.

**Test Steps:**

1. Create spies for rest.config and LwkThemes.setTheme using jest.spyOn.
2. Call component.ngOnInit() to trigger the ngOnInit lifecycle method.
3. Use expect to verify that rest.config is called with the correct parameters and LwkThemes.setTheme is called with the correct theme.

**Expected Outcome:**

* Both rest.config and LwkThemes.setTheme should be called with the expected arguments, and the test should pass.

**4. Test: Should Set Title Property Correctly**

**Description:**  
This test checks if the title property is correctly initialized in the component.

**Test Steps:**

1. Directly check the title property of the component after initialization.
2. Use expect to verify that the title property is set to 'report-unusual-activity'.

**Expected Outcome:**

* The title property should be set to 'report-unusual-activity', and the test should pass.

**App Module Unit Test**

**1. Test: Should Create the AppModule**

**Description:**  
This test checks if the AppModule can be instantiated correctly.

**Test Steps:**

1. Use TestBed.inject(AppModule) to get an instance of AppModule.
2. Use expect to verify that the AppModule is truthy (i.e., the module is correctly created).

**Expected Outcome:**

* The test should pass, confirming that the AppModule is set up correctly.

**2. Test: Should Call Rest.applicationName with "reportSuspiciousActivityWeb"**

**Description:**  
This test checks if Rest.applicationName is correctly called with the expected value.

**Test Steps:**

1. Use expect(Rest.applicationName).toHaveBeenCalledWith('reportSuspiciousActivityWeb') to verify that the method is called with the correct argument.

**Expected Outcome:**

* The test should pass, indicating that Rest.applicationName is invoked with the expected string.

**3. Test: Should Declare AppComponent and ReferralComponent**

**Description:**  
This test checks if both AppComponent and ReferralComponent can be created successfully.

**Test Steps:**

1. Create the AppComponent and ReferralComponent using TestBed.createComponent().
2. Use expect to verify that both components are created successfully.

**Expected Outcome:**

* Both components (AppComponent and ReferralComponent) should be instantiated correctly, and the test should pass.

**4. Test: Should Import Necessary Modules**

**Description:**  
This test checks if the necessary Angular modules are imported and available in the testing environment.

**Test Steps:**

1. Use TestBed.inject() to inject the necessary modules (e.g., BrowserModule, RouterModule, HttpClientModule, etc.).
2. Use expect to verify that each of these modules is available (truthy).

**Expected Outcome:**

* The test should pass if all necessary modules are correctly imported and available.

**5. Test: Should Provide ReferralApiService**

**Description:**  
This test checks if ReferralApiService is correctly provided and available in the testing environment.

**Test Steps:**

1. Use TestBed.inject(ReferralApiService) to retrieve an instance of ReferralApiService.
2. Use expect to check that the service instance is truthy.

**Expected Outcome:**

* The test should pass if ReferralApiService is provided and available for injection.

**App Route Module**

**1. Test: Should Create the AppRoutingModule**

**Description:**  
This test checks if the AppRoutingModule is created correctly in the testing environment.

**Test Steps:**

1. Use TestBed.inject(AppRoutingModule) to retrieve an instance of AppRoutingModule.
2. Use expect() to assert that the module is truthy (i.e., successfully created).

**Expected Outcome:**

* The AppRoutingModule should be successfully instantiated, confirming that the routing module is correctly set up.

**2. Test: Should Have Correct Routes Configured**

**Description:**  
This test ensures that the routes are configured correctly in the appRoutes array and that the paths and components are as expected.

**Test Steps:**

1. Define the expected routes as routes: Routes = appRoutes.
2. Use expect() to check that each route has the correct path and associated component.

**Expected Outcome:**

* The test should pass if the routes are correctly configured as follows:
  + /report-suspicious-activity/:account\_no routes to ReferralComponent.
  + /referral/:account\_no routes to ReferralComponent.
  + /referral routes to ReferralComponent.

**3. Test: Should Correctly Import and Export RouterModule**

**Description:**  
This test verifies that the RouterModule is correctly imported and exported within the AppRoutingModule.

**Test Steps:**

1. Use TestBed.inject(AppRoutingModule) to ensure the module is created.
2. Use TestBed.inject(RouterModule) to verify that the RouterModule is correctly injected into the testing environment.
3. Assert that both modules are truthy or defined.

**Expected Outcome:**

* The test should pass if the AppRoutingModule and RouterModule are properly configured and injected.

Bottom of Form