Automation Framework Setup

Tools Required

- Programming Language: JavaScript/TypeScript
- Testing Framework: Jest or Mocha for unit testing
- Mobile Testing Tool: Appium (for cross-platform testing) or Detox (for React Native apps)
- Version Control: GitHub (for storing test results and reports)

Project Structure

```plaintext
scopex-money-tests/
│
├── src/
│   ├── tests/
│   │   ├── login.test.ts
│   │   ├── addRecipient.test.ts
│   │   └── logout.test.ts
│   └── utils/
│       └── helpers.ts
│
├── screenshots/
├── logs/
└── package.json
```

Test Scenarios

1. Login with Registered User
Objective: Verify that a registered user can log in successfully.
Steps:
  1. Launch the Scopex Money app.
  2. Enter valid credentials (username and password).
  3. Click the "Login" button.
  4. Assert that the user is redirected to the home screen.

2. Adding a Recipient
Objective: Ensure that a user can add a recipient successfully.
Steps:
  1. Navigate to the "Add Recipient" section.
  2. Fill in recipient details (name, account number, etc.).

3. Click the "Add" button.
4. Assert that a success message is displayed.

3. Logout
Objective: Confirm that the user can log out of the application.
Steps:
1. Click on the "Logout" button from the settings menu.
2. Assert that the user is redirected to the login screen.

Implementation Example

Here’s an example of how you might implement one of these tests using TypeScript with Appium:

```typescript
import { remote } from 'webdriverio';

describe('Scopex Money App Tests', () => {
  let driver;

  before(async () => {
    driver = await remote({
      logLevel: 'info',
      capabilities: {
        platformName: 'Android',
        deviceName: 'YourDeviceName',
        app: 'path/to/your/app.apk',
        automationName: 'UiAutomator2'
      }
    });
  });

  after(async () => {
    await driver.deleteSession();
  });

  it('should login with registered user', async () => {
    const usernameField = await driver.$('~username');
    const passwordField = await driver.$('~password');
    const loginButton = await driver.$('~loginButton');

    await usernameField.setValue('your_username');
    await passwordField.setValue('your_password');
    await loginButton.click();

    const homeScreen = await driver.$('~homeScreen');
    expect(await homeScreen.isDisplayed()).toBe(true);
  });

  // Additional tests for adding recipient and logout...
});
```

Capturing Screenshots and Logs

To capture screenshots at critical steps, you can add the following code in your test:

```typescript
```

```
await driver.saveScreenshot(`./screenshots/test-step.png`);
```

For logging, integrate a logging library like Winston or simply use console logs to capture test execution details.

Storing Test Results and Reports

Utilize a CI/CD pipeline (e.g., GitHub Actions) to run tests automatically on each commit and store results in your GitHub repository. You can generate reports using tools like Allure or Mocha's built-in reporters.

Best Practices for Test Scripts

Modularization: Keep your test scripts modular by creating reusable functions for common actions (e.g., login, logout).
Clear Naming Conventions: Use descriptive names for test cases and functions to enhance readability.
Version Control: Regularly commit code changes and maintain branches for different features or fixes.