**Software Testing and Test Automation in Web 3.0**

**By: Sufian Sohail, Zohaib Sohail and Moazam Quereshi**

# **Overview:**

In contrast to normal web applications, which can readily replicate a user workflow for end-to-end testing, web3 apps fail to replicate this procedure. This is mostly owing to two crucial elements that are unique to their species. To begin, web3 apps run on an ever-changing backend, which is most commonly referred to as a blockchain and is driven by one or more smart contracts. The second requirement is that users sign transactions using wallet-like third-party software and submit them through a provider in order to connect with any web3 app.

I'll talk on the importance of end-to-end testing, with an emphasis on and examples from the Ethereum blockchain.

# **Web 3.0 Applications Architecture**

Independent of the blockchain they employ, current decentralised applications are structured by the following two elements: The business logic of the product or protocol is provided through smart contracts and the user interface is used to communicate with the users. It would not be incorrect to state that a DApp's user interface is its front end and that smart contracts serve as its back end in the context of contemporary software applications.

DApps can take on a variety of shapes and forms, but it's safe to say that the majority function primarily as websites, with browsers serving as their primary interface. Instead of relying on a graphical user interface, knowledgeable blockchain users may typically engage directly with the smart contracts.

# **Testing of DApp’s**

Truffle includes an automated testing framework as standard to make it simple to test your contracts. With this framework, you can create straightforward tests in two different ways:

## **Mocha and Chai for JavaScript Tests**

Truffle gives you a strong foundation from which to develop your JavaScript tests by utilising the Mocha testing framework and Chai for assertions. Let's get started and explore how Truffle expands on Mocha to simplify contract testing.

**Use contract() instead of describe()**

Your tests should mostly follow Mocha's in terms of structure. Your tests must have a.js extension, be located in the./test directory, and contain code that Mocha can identify as an automated test. Truffle tests differ from Mocha tests in that they use the contract() method. The only difference between this function and describe() is that this one allows Truffle's clean-room capabilities.

It operates as follows:

* Your contracts are redeployed to the active Ethereum client before each contract() function is called, ensuring that the tests included inside it run with a clean contract state.
* You may utilise the list of accounts made accessible by your Ethereum client provided by the contract() method to create tests.

Because Truffle utilises Mocha internally, whenever Truffle's clean-room features are not required, you may still run standard Mocha tests using describe().

**Use of Contract Abstraction**

Contract abstractions serve as the framework (essentially, our flux capacitor) that enables contract interaction from JavaScript. Truffle has no method of figuring out which contracts you'll need to interact with throughout your tests, so you'll have to explicitly request those contracts. Using the artefacts is how you accomplish this. Truffle's need() function enables you to ask for a workable contract abstraction for a particular Solidity contract. The example that follows will demonstrate how to utilise this abstraction to ensure that your contracts are operating properly.

**Use of artifacts.require()**

Using artifacts.require() in your tests is the same as using it in your migrations; you only need to supply the contract name. For further information, see the Migrations section's artifacts.require() reference.

**Use of Web 3.0**

Each test file has a web3 instance that is configured to the appropriate provider. As a result, invoking web3.eth.getBalance just works.

**Specification of Test Case**

You may limit the tests that are run to a certain file by doing the following:

truffle test ./test/metacoin.js

## **Solidity Testing**

Javascript tests and Solidity test contracts coexist in.sol files. As specified in the test contract, they will be executed as a separate test suite as part of the truffle test. All the advantages of the Javascript tests, such as a clean environment for each test suite, easy access to your deployed contracts, and the capacity to import any contract dependency, are still present in these contracts. The following problems were taken into consideration when developing Truffle's Solidity testing framework in addition to these features:

* No contract should be used as a basis for solidity tests (like a Test contract). This minimises your testing requirements and offers you total authority over the contracts you draught.
* No assertion library should be required for solidity testing. Truffle gives you access to a default assertion library, but you may alter it whenever you need to to suit your needs.
* Any Ethereum client should be able to execute your Solidity tests.

### **Test Structure**

The structure of writing the test cases in solidity is discusses now.

#### **Assertion**

The truffle/Assert.sol package offers you assertion routines like Assert.equal(). This is the default assertion library, but you are free to use another one as long as it has some loose integration with Truffle's test runner and fires off the appropriate assertion events. Assert.sol contains all of the accessible assertion functions.

#### **Deployed Addresses**

The truffle/DeployedAddresses.sol library provides access to the addresses of your deployed contracts (i.e., those that were deployed as a result of your migrations). This is offered by Truffle and is recompiled and relinked before each suite is performed to offer your tests a clean room environment using Truffle. All of your deployed contracts can use the features this library offers, such as:

DeployedAddresses.<contract name>();

This will return an address from which you may view the contract. The usage is demonstrated in the above example test.

You must import the contract code into your test suite in order to use the deployed contract. In the example, import "../contracts/MetaCoin.sol"; This import refers to the test contract, which is located in the./test directory, and it searches outside of the test directory for the MetaCoin contract. It then casts the address to the MetaCoin type using that contract.

#### **Naming Test Contracts**

All test contracts must begin with the word Test, capitalised. This separates this contract from test helpers and project contracts (i.e., the contracts under test) and informs the test runner which contracts constitute test suites.

#### **Naming of Test Functions**

All test functions, like test contract names, must begin with test, lowercase. Each test function is run as a separate transaction in the order they appear in the test file (like your Javascript tests). Truffle/Assert provides assertion functions. The test runner analyses sol trigger events to determine the test outcome. Assertion functions produce a boolean expressing the assertion's conclusion, which you may use to exit the test early to avoid execution mistakes (as in, errors that Ganache or Truffle Develop will expose).

#### **Before / After Hooks**

You are given a plethora of test hooks, as demonstrated in the sample below. These hooks are beforeAll, beforeEach, afterAll, and afterEach, and they are the same hooks that Mocha provides in your Javascript tests. These hooks can be used to perform setup and teardown tasks before and after each test, or before and after each suite is run. Each hook, like test functions, is executed as a single transaction. It should be noted that certain sophisticated tests will require a large amount of setup, which may exceed the gas limit of a single transaction; you can work around this constraint by generating many hooks with different suffixes.

#### **Advance Features**

Truffle comes up with some extra features for testing of block chain contracts like

* Testing for exceptions
* Testing ether Transactions