

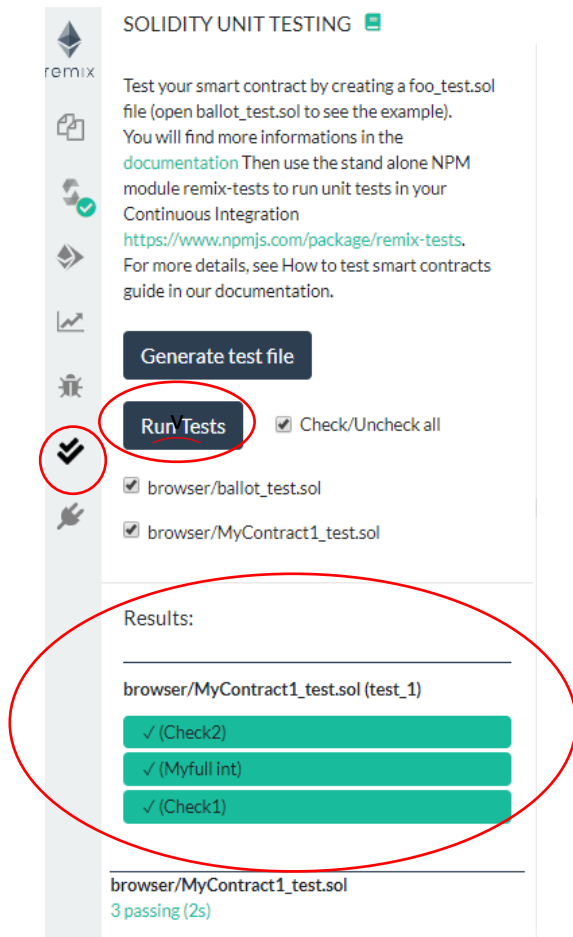
### Smart Contract Testing

Smart Contracts can be tested in using Remix, Truffle, or other approaches. This approach is called Unit testing. Developers can also ad hoc their contracts as much as needs. Smart contract Unit testing is a more formal, documented process, and is a “good practice”.

Unit Tests can be written in Solidity, or JavaScript. Solidity Tests have a set of libraries to verify data.

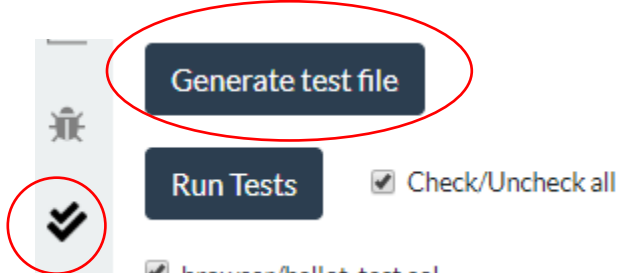
The Remix IDE has a testing module that will create contract frameworks for testing. It also adds a “\_test” suffix to the name of the test contract. Using the suffix, Remix recognizes the test contracts and is able to bundle the tests into Remix’s Testing Module.

Remix lists the test with a checkbox. When the Testing Module’s **Run Tests** button, all the checked tests will run, and the results will be written to a console below. If desired, the Remix tests can be run as a suite.



### Creating Remix Tests

Remix tests are also created in the Remix testing module. When the **Generate test file** button in the Testing module is clicked, a testing file is generated with the “\_test” suffix. The testing file contains a framework to run contract tests.



When the testing file is generated it only contains “out-of-the-box”, generic contract source code, that makes a test framework. Some instructions on how to build tests are included in the test contract’s comments. No contract code or variables are brought into the test contract. The contracts and tests in the testing files must be edited to call the Smart Contracts functions and variables.

```

1  pragma solidity >=0.4.0 <0.6.0;
2      import "remix_tests.sol"; // this import is automatically injected by Remix.
3
4      // file name has to end with '_test.sol'
5      contract test_1 {
6
7          function beforeAll() public {
8              // here should instantiate tested contract
9              Assert.equal(uint(4), uint(3), "error in before all function");
10         }
11
12         function check1() public {
13             // use 'Assert' to test the contract
14             Assert.equal(uint(2), uint(1), "error message");
15             Assert.equal(uint(2), uint(2), "error message");
16         }
17
18         function check2() public view returns (bool) {
19             // use the return value (true or false) to test the contract
20             return true;
21         }
22     }
23
24     contract test_2 {
25
26         function beforeAll() public {
27             // here should instantiate tested contract
28             Assert.equal(uint(4), uint(3), "error in before all function");
29         }
30
31         function check1() public {
32             // use 'Assert' to test the contract
33             Assert.equal(uint(2), uint(1), "error message");
34             Assert.equal(uint(2), uint(2), "error message");
35         }
36
37         function check2() public view returns (bool) {
38             // use the return value (true or false) to test the contract
39             return true;
40         }
41     }
  
```

A contract as shown below will have its functions and variables tested using Remix testing. The contract will be instantiated in the test contract and the functions will be called in the test file.

```
1 pragma solidity ^0.5.1;
2 import "remix_tests.sol"; // this import is automatically injected by Remix.
3
4 contract MyContract1 {
5     string public value;
6     string public myVal;
7     int256 public myInt;
8     string public str1;
9
10
11     constructor() public {
12
13         set("Good","Day");
14
15         //value = "myValue";
16         //myVal = "myString";
17     }
18
19     function hint (int256 i) public returns (int256 fullInt){
20
21         myInt=i;
22         fullInt = myInt + 10;
23         return fullInt;
24     }
25
26
27     function get() public view returns(string memory , string memory) {
28         return (value, myVal);
29     }
30
31
32     function myGet() public returns(string memory) {
33         str1 = myVal;
34         return str1;
35     }
36
37
38     function set(string memory _value, string memory _MyVal1) public {
39         value = _value;
40         myVal = _MyVal1;
41     }
42
43 }
```

## Introduction to Blockchain Development with Ethereum by Coding-Bootcamps.com

### Session 9- Testing Ethereum Contracts

The Remix testing file below is built from the Contract shown above. The test contract shown below is instantiating the contracts functions and returning results to validate testing.

The screenshot shows the Remix IDE interface. On the left, the 'SOLIDITY UNIT TESTING' sidebar is visible. It contains instructions on how to test smart contracts and a 'Run Tests' button, which is circled in red. Below the button, there are checkboxes for different test files: 'browser/ballot\_test.sol', 'browser/MyContract1\_test.sol' (checked), and 'browser/test\_test.sol'. The 'Results' section shows the test results for 'browser/MyContract1\_test.sol (test\_1)', with three green bars indicating successful tests: '(Check2)', '(Myfull int)', and '(Check1)'. The bottom of the sidebar shows '3 passing (23s)'. On the right, the 'MyContract1\_test.sol' file is open, showing Solidity code for testing the contract. The code includes imports for 'remix\_tests.sol' and 'MyContract1.sol', and defines a 'test\_1' contract with functions for testing the contract's behavior.

### Remix Testing Functions

The Remix tests include a group of functions to validate contracts. These functions evaluate values returned and look for equality, or inequality to expected values and behavior. These functions are called the **Assert** functions.

The available Assert libraries are as follows:

Assert.ok()	bool
Assert.equal()	uint, int, bool, address, bytes32, string
Assert.notEqual()	uint, int, bool, address, bytes32, string
Assert.greaterThan()	uint, int
Assert.lessThan()	uint, int

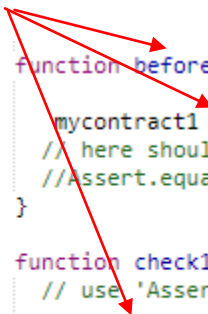
The Assert functions follow a template for evaluations. The template follows: `Assert.equal(returned value, expected value, "message")`

Available special functions for Solidity Testing:

- `beforeEach()` - runs before each test. This function run to be sure a new instance of the contract is created if needed.
- `beforeAll()` – instantiates the contract to be tested
- `before()` function runs before tests, instantiates the contract to be tested, and initializes.

```
15  function beforeAll() public {
16
17      mycontract1 = new MyContract1();
18      // here should instantiate tested contract
19      //Assert.equal(uint(4), uint(3), "error in before all function");
20  }
21
```

Calling the instantiated contracts, functions and variables.



```
15  function beforeAll() public {
16
17      mycontract1 = new MyContract1();
18      // here should instantiate tested contract
19      //Assert.equal(uint(4), uint(3), "error in before all function");
20  }
21
22  function check1() public {
23      // use 'Assert' to test the contract
24
25      myfullInt = mycontract1.hint(5);
26      mycontract1.set("Hello","World");
27  }
```

## Continuous Integration

Solidity Remix testing has a command line interface (CLI). The remix-tests can be configured in an automated continuous integration (CI) environment which supports Node.js. The CI environment can also include deployments.

## Node Package Manager and remix-tests

The NPM package for Remix tests is called: `remix-tests`. The package can be installed using: `npm -g install remix-tests`.

#### Testing Structure

Truffle includes an automated testing framework. This framework lets you write simple and manageable tests in two different ways:

- JavaScript
- Also, Solidity

Similar to Remix testing, JavaScript tests also have special functions to call Solidity Contracts, evaluate results and return results.

JavaScript tests using the Truffle console to instantiate contracts, call functions and evaluate results. JavaScript tests uses special functions, such as **async()/await()** to run JavaScript tests.

The truffle JavaScript tests also have initialization function such as the, **beforeEach** function. The truffle JavaScript tests use their own **assert** functions to evaluate results.

A typical Truffle JavaScript test includes a **require()** function which can be used to import the contract and instantiate to be tested. JavaScripts tests also have their own **before()** function which calls contracts.

Each test runs in one **it()** function block with a name and list of contract functions calls and an evaluation of results. JavaScript testing also uses its own **assert()** functions for evaluations of equality and inequality.

The JavaScript test below, test the same contract we tested using the Remix testing. The JavaScript tests have the functions:

require()

before()

async()

await()

assert(), for equality, or inequality.

The JavaScript tests validate deployment, and call functions that return integers and strings.



```
1  const MyContract1 = artifacts.require('/MyContract1.sol')
2
3  contract('MyContract1', (accounts) => {
4    before(async () => {
5      this.mycontract1 = await MyContract1.deployed()
6    })
7
8    //Is the contract successfully deployed?
9    //Does it have an Ethereum address?
10
11    it('deploys successfully', async () => {
12      const address = await this.mycontract1.address
13      assert.notEqual(address, 0x0)
14      assert.notEqual(address, '')
15      assert.notEqual(address, null)
16      assert.notEqual(address, undefined)
17    })
18
19    //Test numeric results
20    it('set hint', async () => {
21      const int0 = await this.mycontract1.hint(25)
22      const nint = await this.mycontract1.fullInt
23      assert.notEqual(nint, 0)
24    })
25
26    //Test string results
27    it('set string', async () => {
28      const s1 = await this.mycontract1.set('my', 'test')
29      const s2 = await this.mycontract1.get()
30
31      assert.notEqual(s2.value, '')
32      assert.notEqual(s2.myVal, '')
33
34    })
35
36
37
38 })
```

The results of running the JavaScript tests are below. The contract is successfully deployed, ie. it has an Ethereum address, the integer values are tested and the string values are tested.

#### Command Prompt

```
C:\ethereumTesting>truffle test
Using network 'development'.
```

```
Contract: MyContract1
  ✓ deploys successfully
  ✓ set hint (44ms)
  ✓ set string (112ms)
```

```
3 passing (244ms)
```

```
C:\ethereumTesting>
```

## More Resources

To read more on blockchain and understand it in depth, the reading the following articles are highly recommended:

- [History and Evolution of Blockchain Technology from Bitcoin](#)
- [Overview of Blockchain evolution and phases from Ethereum to Hyperledger](#)
- [Comprehensive overview and analysis of blockchain use cases in many industries](#)
- [Overview of blockchain technology and blockchain development](#)

Also, the following are more tutorials and resources on Ethereum blockchain development.

- [How to Write Ethereum Smart Contracts with Solidity in 1 hour](#)
- [How to Build Auction DApp with Ethereum and Solidity Programming Language](#)
- [How to Work with Ethereum Blockchain Applications through Remix IDE](#)
- [Certified Solidity Professional Certification exam](#)
- [Learn Ethereum: Build your own DApps with Ethereum and smart contracts book by Brian Wu](#)