IBM Blockchain Platform

Tutorial A8: Testing a smart contract

Estimated time: `20 minutes`

In the previous tutorial we used the VS Code debugger to step through our smart contract. In this tutorial we will:

- Look at the features for generating functional tests
- Generate functional tests for our smart contract
- Customize and run a sample test

In order to successfully complete this tutorial, you must have first completed tutorial A6: Upgrading a smart contract in the active workspace. It is desirable (but not mandatory) to have also completed tutorial A7: Debugging a smart contract.

☐ `A8.1`: Expand the first section below to get started.

▶ **Generate functional tests**

Throughout these tutorials we've been submitting and evaluating transactions in an ad-hoc manner using client applications and VS Code. This has helped us understand how blockchain smart contracts and applications work.

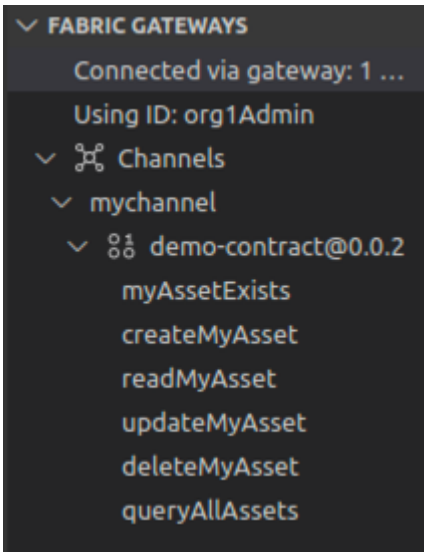
Of course, good practices for software development also apply to smart contracts and applications, which means that when you're developing real-world blockchain solutions it's important to use a structured framework to allow more formal testing of the code you write.

While an exhaustive discussion of these practices is beyond the scope of this tutorial, we will now look at the features in IBM Blockchain Platform and VS Code which facilitate smart contract functional testing.

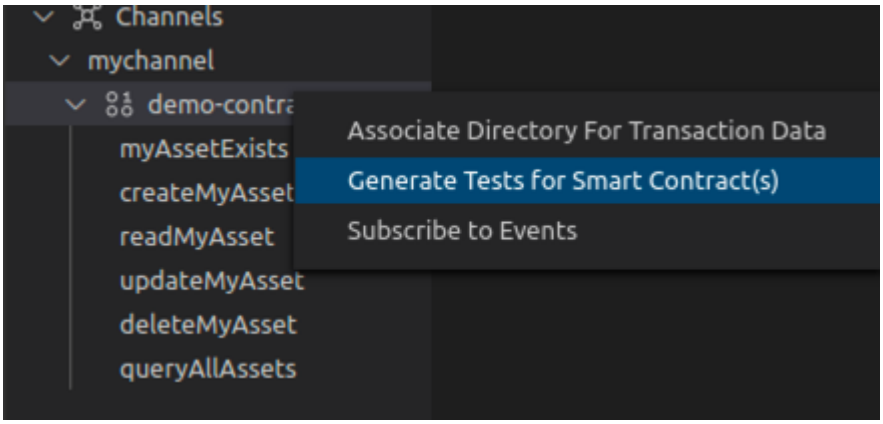
Let's start by creating some functional tests for our smart contract package.

☐ `A8.2`: Ensure that the Fabric Gateways view is visible and that the local network is connected.

If necessary, click the IBM Blockchain Platform activity bar icon to show the Fabric Gateways view, and click '1 Org Local Fabric' to connect to the gateway. The smart contract 'demo-contract@0.0.2' should be available on the 'mychannel' network.

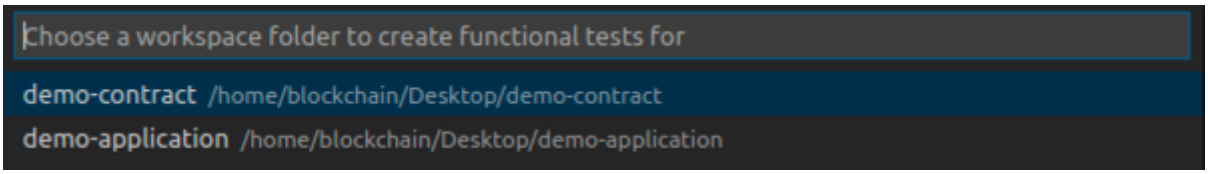


A8.3: Right-click 'demo-contract@0.0.2' and select 'Generate Tests for Smart Contract(s)'.



We want to generate tests for our smart contract.

A8.4: Click 'demo-contract'.



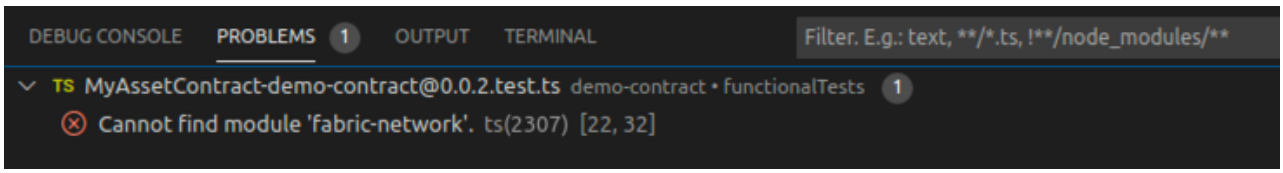
We will use the TypeScript language for our tests.

A8.5: Click 'TypeScript'.

A test application called 'MyAssetContract-demo-contract@0.0.2.test.ts' will be generated in a 'functionalTests' folder and shown in the editor. You might need to wait a minute or so while VS Code attempts to build the tests.

Some errors will appear and subsequently disappear during the build process.

Cannot find module?
You might be left with a single error that suggests that the 'fabric-network' module cannot be found:



This is a dependency that was resolved during the build process, but the reported error is not always removed automatically.

To tidy up the error, either close and re-open the *'MyAssetContract-demo-contract@0.0.2.test.ts'* editor, or select 'View' -> 'Command Palette' and run the 'Developer: Reload Window' command.

After doing this, you should see no errors reported.

The application contains a set of tests for the 'MyAsset' smart contract using the Mocha testing framework. Of course you may wish to use your own testing framework. Let's use Mocha and our sample test application to understand the principles of functional testing:

- Tests are grouped together using a `describe()` expression
- Individual tests within a group are identified with an `it()` expression
- Each smart contract method has a specific test `describe` group with one or more `it()` functional tests within it
- Each test executes a smart contract transaction method and checks the result with an `assert()` expression.
- The initial `describe` group sets up Mocha hooks for all subsequent test groups. These hooks run `before()` each subsequent `describe()` group and `beforeEach()` and `afterEach()` functional `it()` test. The hooks are used to set up a new gateway for each group of tests, and reuse the same gateway within a group of tests. In this way the individual functional test can focus on smart contract method testing.

Take some time to review the generated *MyAssetContract-demo-contract@0.0.2.test.ts* file before continuing.

In addition to some standard code to connect to a Fabric gateway using an identity specified near the top of the file, the test application contains clauses for each of the transactions described in our smart contract, and these attempt to call the transaction and check the output.

A8.6: Expand the next section of the tutorial to continue.

▶ **Customize and run functional tests**

If you look closely at the checks made by each of the transaction tests, you'll see that they simply make the assertion that true equals true. We need to customize these tests, replacing each check with one that compares the response from the transaction with the desired output.

In this section we'll update one of these test transactions and try it out.

A8.7: Scroll to the *myAssetExists* test.

```
Run Test | Debug Test
62    describe('myAssetExists', () => {
          Run Test | Debug Test
63        it('should submit myAssetExists transaction', async () => {
64            // TODO: populate transaction parameters
65            const myAssetId: string = '002';
66            const args: string[] = [ myAssetId];
67
68            const response: Buffer = await SmartContractUtil.submitTransaction('
69            // submitTransaction returns buffer of transaction return value
70            // TODO: Update with return value of transaction
71            assert.equal(JSON.parse(response.toString()), true);
72            // assert.equal(JSON.parse(response.toString()), undefined);
73        }).timeout(10000);
74    });
75
```

A8.8: Replace the *assert.equal(true, true);* statement with the line:

```
assert.equal(JSON.parse(response.toString()), true);
```

The new section of code should look like this:

```
const response: Buffer = await SmartContractUtil.submitTransac
// submitTransaction returns buffer of transcation return valu
// TODO: Update with return value of transaction
assert.equal(JSON.parse(response.toString()), true);
// assert.equal(JSON.parse(response.toString()), undefined);
```

It is checking that the output of the 'myAssetExists' transaction is true for the value of the input parameter 'EXAMPLE'. In order words, it is checking to see if the asset with the key 'EXAMPLE' exists.

A8.9: Save the file ('File' -> 'Save').

A8.10: Click the 'Run Test' hyperlink that is just before the *describe('myAssetExists')* clause.

```
Run Test Debug Test
describe('myAssetExists', () => {
    Run Test | Debug Test
    it('should submit myAssetExists transactio
        // TODO: populate transaction paramete
        const myAssetId: string = 'EXAMPLE';
        const args: string[] = [ myAssetId];
```

The test will now run. After a brief pause you will see the output in the terminal:

```
1) MyAssetContract-demo-contract@0.0.2
     myAssetExists
       should submit myAssetExists transaction:

    AssertionError [ERR_ASSERTION]: false == true
    + expected - actual

    -false
    +true

    at Context.it (functionalTests/MyAssetContract-demo-contract@0.0.2.test.ts:70:20)
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:189:7)
```

This test is failing as expected, because the 'EXAMPLE' key does not exist.

We will now edit the test to check for a key that we know exists. As a result of earlier tutorials, you should have assets with keys '002' and '003' described in your world state. (If not, first try submitting a 'createMyAsset' transaction again.)

- A8.11: Change 'EXAMPLE' in the *myAssetId* definition to `'002'`.

```
it('should submit myAssetExists transaction', async () => {
    // TODO: populate transaction parameters
    const myAssetId: string = '002';
    const args: string[] = [ myAssetId];
```

- A8.12: Save the file ('File' -> 'Save').
- A8.13: Click 'Run Test' again.

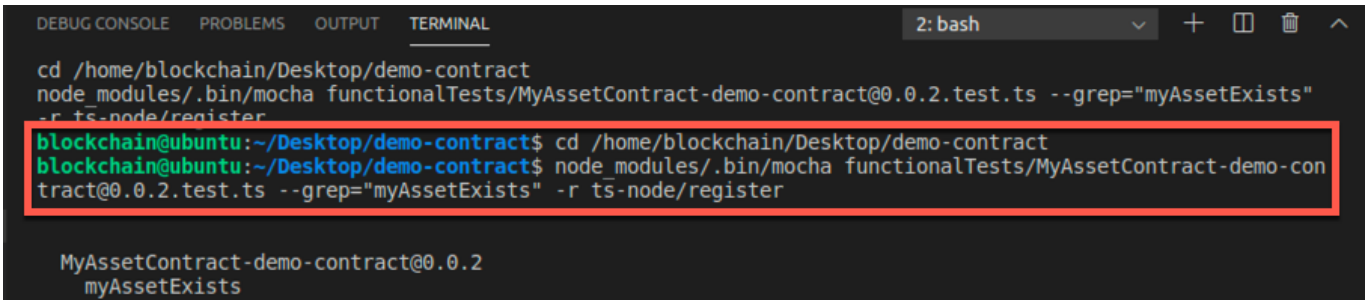
This time you will see that the test passes, because the asset with key '002' exists in the world state.

```
MyAssetContract-demo-contract@0.0.2
  myAssetExists
    ✓ should submit myAssetExists transaction (2310ms)


1 passing (3s)
```

While we have just tested a single smart contract method, the generated application can be run in its entirety so that all transactions in a smart contract can be functionally tested. You can see the command to do run the tests under the mocha framework in the Terminal:

```
node_modules/.bin/mocha functionalTests/MyAssetContract-demo-
contract@0.0.2.test.ts --grep="myAssetExists" -r ts-node/register
```



You can add additional tests to the application, and incorporate other testing frameworks with Mocha to build a comprehensive set of tests for your smart contract. Using the underlying commands also helps you integrate your tests in a CI/CD pipeline.

Summary

In this tutorial, we have seen how functional tests can be generated for our smart contracts. We have also seen how to customize these tests so that we can check that smart contracts are running correctly.

In the next tutorial, we will see how we can update our smart contract to generate an event, and how applications can register and receive event notifications.

→ Next: A9: Publishing an event