

← [A7: Debugging a smart contract](#)



## Tutorial A8: Testing a smart contract

---

Estimated time: **20 minutes**

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

- Look at the features in IBM Blockchain Platform 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

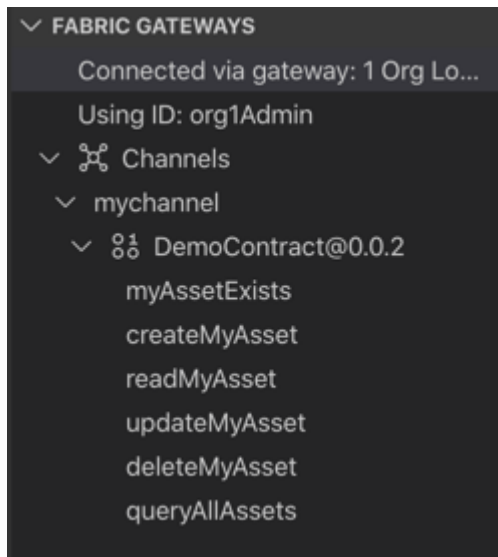
Throughout these tutorials we've been submitting and evaluating transactions individually, using client applications and VS Code.

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 framework to allow more formal testing of the code you write.

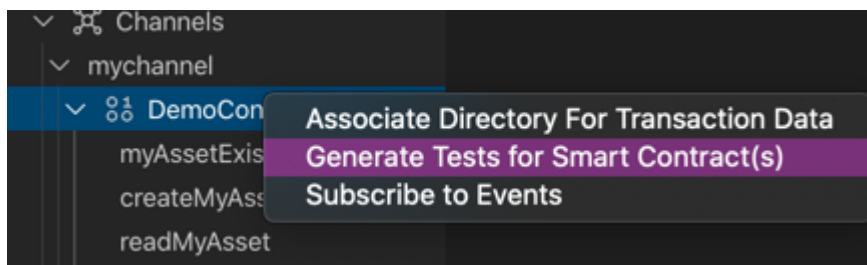
While discussion of these practices is beyond the scope of this tutorial, we will now look at the features in IBM Blockchain Platform and VS Code to facilitate the creation of functional tests for smart contracts.

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

If necessary, click the IBM Blockchain Platform sidebar icon to show the Fabric Gateways view, and click '1 Org Local Fabric' to connect to the gateway. DemoContract@0.0.2 should be instantiated on the 'mychannel' network.

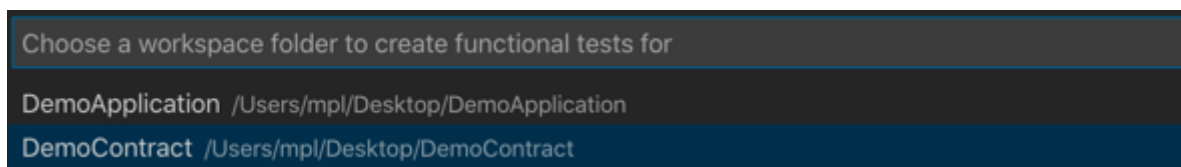


□ A8.3: Right-click 'DemoContract@0.0.2' and select 'Generate Tests for Smart Contract(s)'.



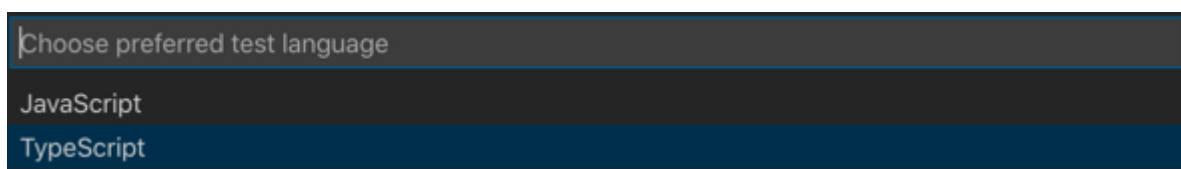
We want to generate tests for our smart contract.

□ A8.4: Click 'DemoContract'.



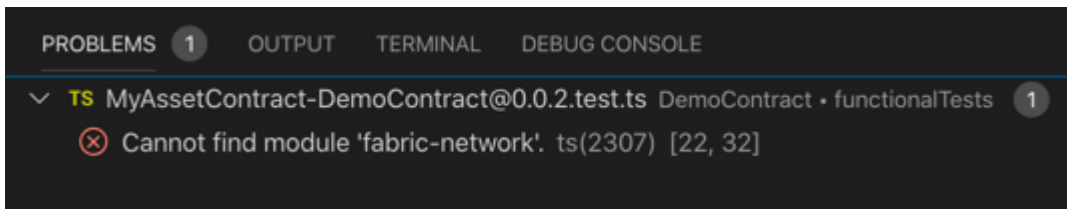
We will use the TypeScript language for our tests.

□ A8.5: Click 'TypeScript'.

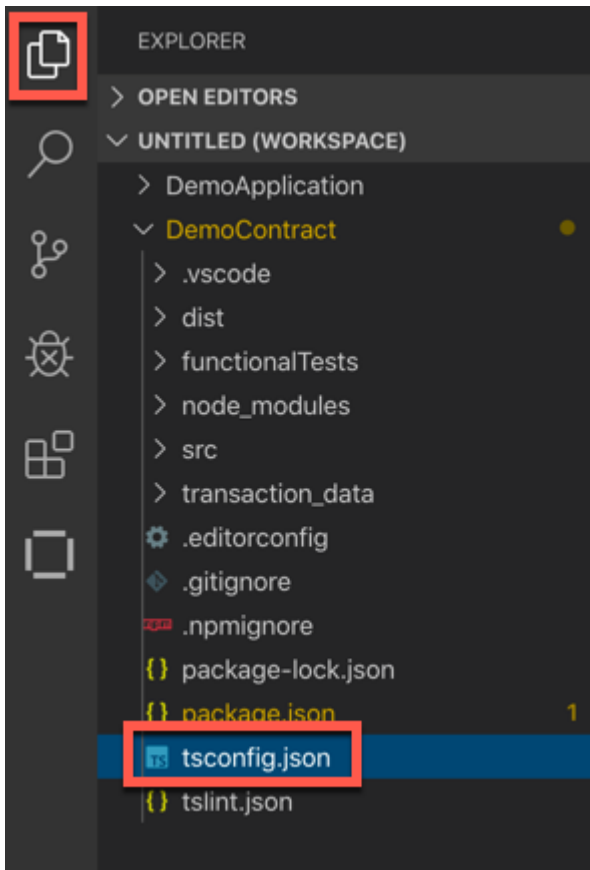


The test application will be generated in a new functionalTests folder and shown in the editor. You might need to wait a minute or so while VS Code attempts to build the tests.

Unfortunately, the tests will not build correctly because our DemoContract TypeScript project already contains a tsconfig.json file, which will need updating to include the new source file. When building has completed, if you click on the Problems tab you will see the following error:



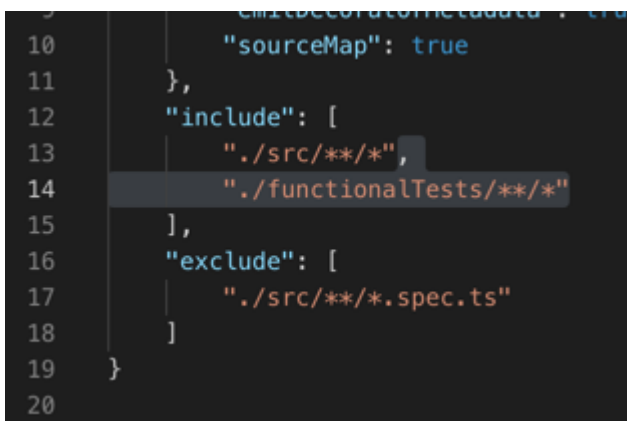
□ A8.6: Click on the Explorer sidebar icon and load the DemoContract -> tsconfig.json file.



We need to add a reference to our new functional tests directory to this file.

□ A8.7: Inside the *include* clause, after the `"/src/**/*"` expression add a comma (,) and then insert the line `"/functionalTests/**/*"`.

The new clause should look like this:



□ A8.8: Save the file ('File' -> 'Save').

After a pause for compilation, the errors will disappear.

No problems have been detected in the workspace so far.

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

In addition to some standard code to connect to the Fabric gateway, 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.

**A8.9:** 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 replace each check with one that looks at the response from the transaction and compares it with the desired output.

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

**A8.10:** Scroll to the *myAssetExists* test.

```

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

```

**A8.11:** Replace the *assert.equal(true, true);* statement with the line

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

```

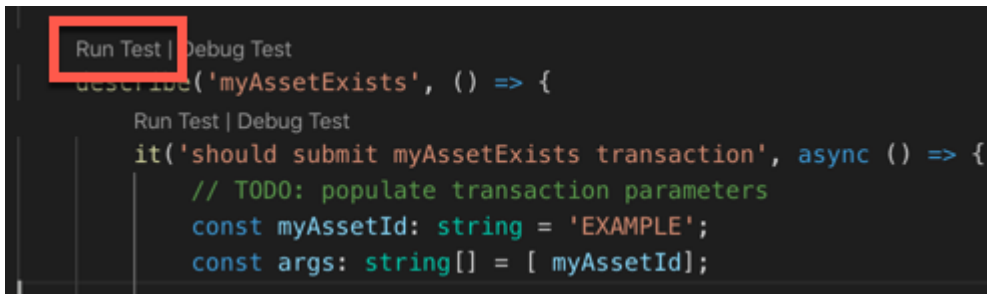
const response: Buffer = await SmartContractUtil.submitTransaction
// submitTransaction returns buffer of transaction return value
// TODO: Update with return value of transaction
assert.equal(JSON.parse(response.toString()), true);
// assert.equal(JSON.parse(response.toString()), undefined);

```

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

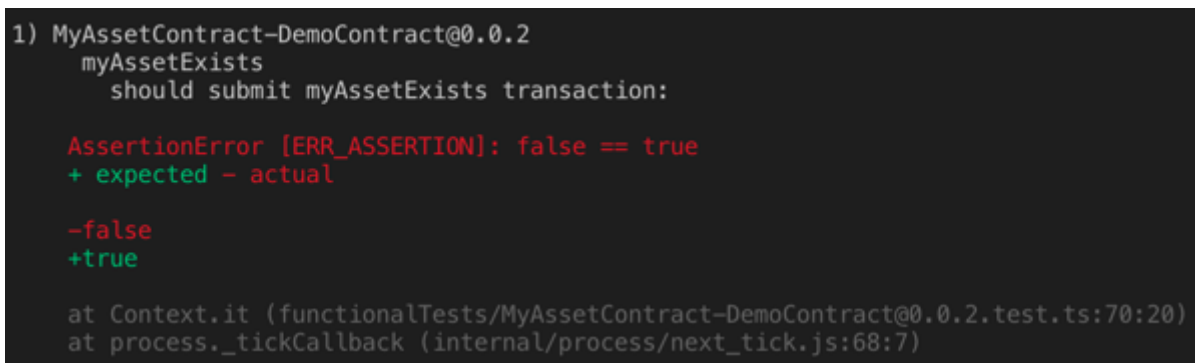
□ A8.12: Save the file ('File' -> 'Save').

□ A8.13: 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 transaction', async () => {
    // TODO: populate transaction parameters
    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-DemoContract@0.0.2
   myAssetExists
     should submit myAssetExists transaction:

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


-false
+true

at Context.it (functionalTests/MyAssetContract-DemoContract@0.0.2.test.ts:70:20)
at process._tickCallback (internal/process/next_tick.js:68:7)
```

This test is failing as expected, because the 'EXAMPLE' key does not exist in our blockchain's world state.

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.)

□ A8.14: Change 'EXAMPLE' in the `myAssetId` definition to '002'.




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

□ A8.15: Save the file ('File' -> 'Save').

□ A8.16: 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-DemoContract@0.0.2
  myAssetExists
    ✓ should submit myAssetExists transaction (2398ms)

1 passing (3s)
```

While we have just tested a single transaction here, the generated application can be run in its entirety so that all transactions in a smart contract can be functionally tested. You can add additional tests to the application, and the tests can also be combined with a more comprehensive testing framework and integrated into a build 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 allow other applications to be notified when an interesting event occurs.

---

→ [A9: Publishing an event](#)