← A5: Invoking a smart contract from an external application

Tutorial A6: Upgrading a smart contract

Estimated time: `20 minutes`

> **Note:** This tutorial in based on Hyperledger Fabric v1.x. There is a new chaincode lifecycle feature in v2 that improves the upgrade process. Check out the Hyperledger Fabric v2.0 chaincode documentation for details.

In the last tutorial we built and tested a Typescript application that interacted with our Hyperledger Fabric network. In this tutorial we will:

- Make a change to our smart contract
- Package, install and instantiate the new smart contract
- Try out the new smart contract

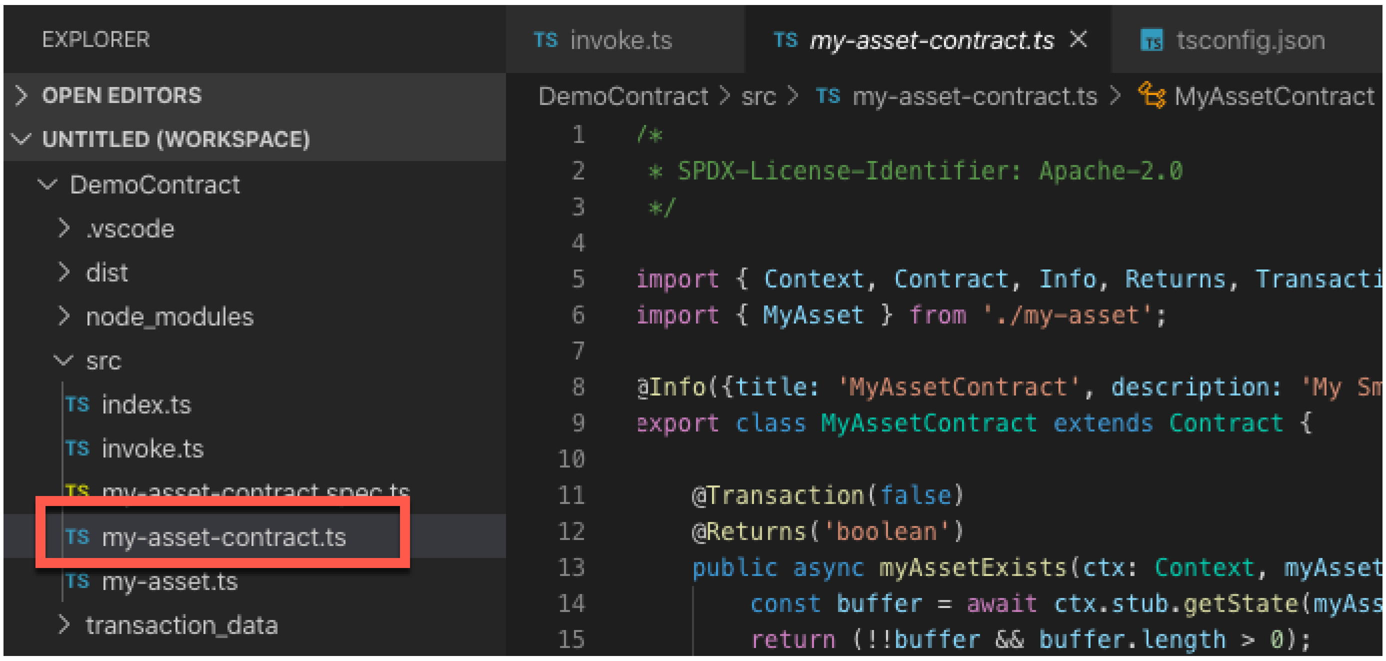
In order to successfully complete this tutorial, you must have first completed tutorial A5: Invoking a smart contract from an external application in the active workspace.

`A6.1`: Expand the first section below to get started.

▶ **Modify the smart contract**

`A6.2`: Focus the VS Code editor on the *my-asset-contract.ts* file.

You should be able to switch directly to this tab as it should still be loaded from earlier tutorials. If it is not, use the Explorer sidebar to navigate to my-asset-contract.ts in the src folder of the DemoContract project.



```
/*
 * SPDX-License-Identifier: Apache-2.0
 */

import { Context, Contract, Info, Returns, Transacti
import { MyAsset } from './my-asset';

@Info({title: 'MyAssetContract', description: 'My Sm
export class MyAssetContract extends Contract {

    @Transaction(false)
    @Returns('boolean')
    public async myAssetExists(ctx: Context, myAsset
        const buffer = await ctx.stub.getState(myAss
        return (!!buffer && buffer.length > 0);
```

We're going to add a new transaction to our smart contract which will return all of the available assets between '000' and '999'.

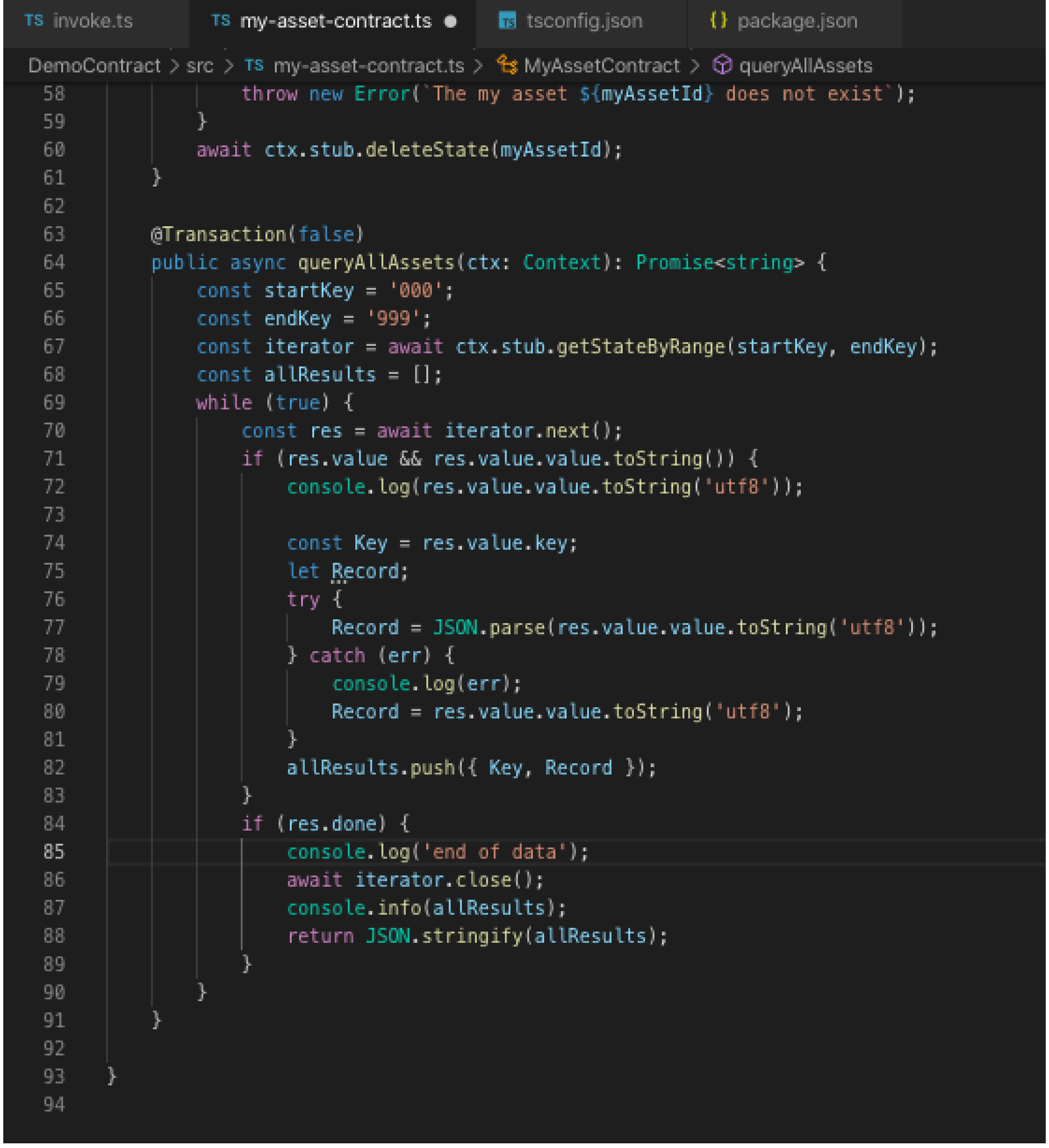
☐ A6.3: Using copy and paste, insert the following method after the closing brace of the updateMyAsset method, but before the final closing brace of the whole file:

```
    @Transaction(false)
    public async queryAllAssets(ctx: Context): Promise<string> {
        const startKey = '000';
        const endKey = '999';
        const iterator = await ctx.stub.getStateByRange(startKey, endKey);
        const allResults = [];
        while (true) {
            const res = await iterator.next();
            if (res.value && res.value.value.toString()) {
                console.log(res.value.value.toString('utf8'));

                const Key = res.value.key;
                let Record;
                try {
                    Record = JSON.parse(res.value.value.toString('utf8'));
                } catch (err) {
                    console.log(err);
                    Record = res.value.value.toString('utf8');
                }
                allResults.push({ Key, Record });
            }
            if (res.done) {
                console.log('end of data');
                await iterator.close();
                console.info(allResults);
                return JSON.stringify(allResults);
            }
        }
    }
```

You can also get the source for this method from here.

Your source file should now look similar to this:


TS invoke.ts
TS my-asset-contract.ts
tsconfig.json
{} package.json
DemoContract > src > TS my-asset-contract.ts > MyAssetContract > queryAllAssets

```
            throw new Error(`The my asset ${myAssetId} does not exist`);
        }
        await ctx.stub.deleteState(myAssetId);
    }

    @Transaction(false)
    public async queryAllAssets(ctx: Context): Promise<string> {
        const startKey = '000';
        const endKey = '999';
        const iterator = await ctx.stub.getStateByRange(startKey, endKey);
        const allResults = [];
        while (true) {
            const res = await iterator.next();
            if (res.value && res.value.value.toString()) {
                console.log(res.value.value.toString('utf8'));

                const Key = res.value.key;
                let Record;
                try {
                    Record = JSON.parse(res.value.value.toString('utf8'));
                } catch (err) {
                    console.log(err);
                    Record = res.value.value.toString('utf8');
                }
                allResults.push({ Key, Record });
            }
            if (res.done) {
                console.log('end of data');
                await iterator.close();
                console.info(allResults);
                return JSON.stringify(allResults);
            }
        }
    }

}

```

☐ A6.4: Save the updated file ('File' -> 'Save').

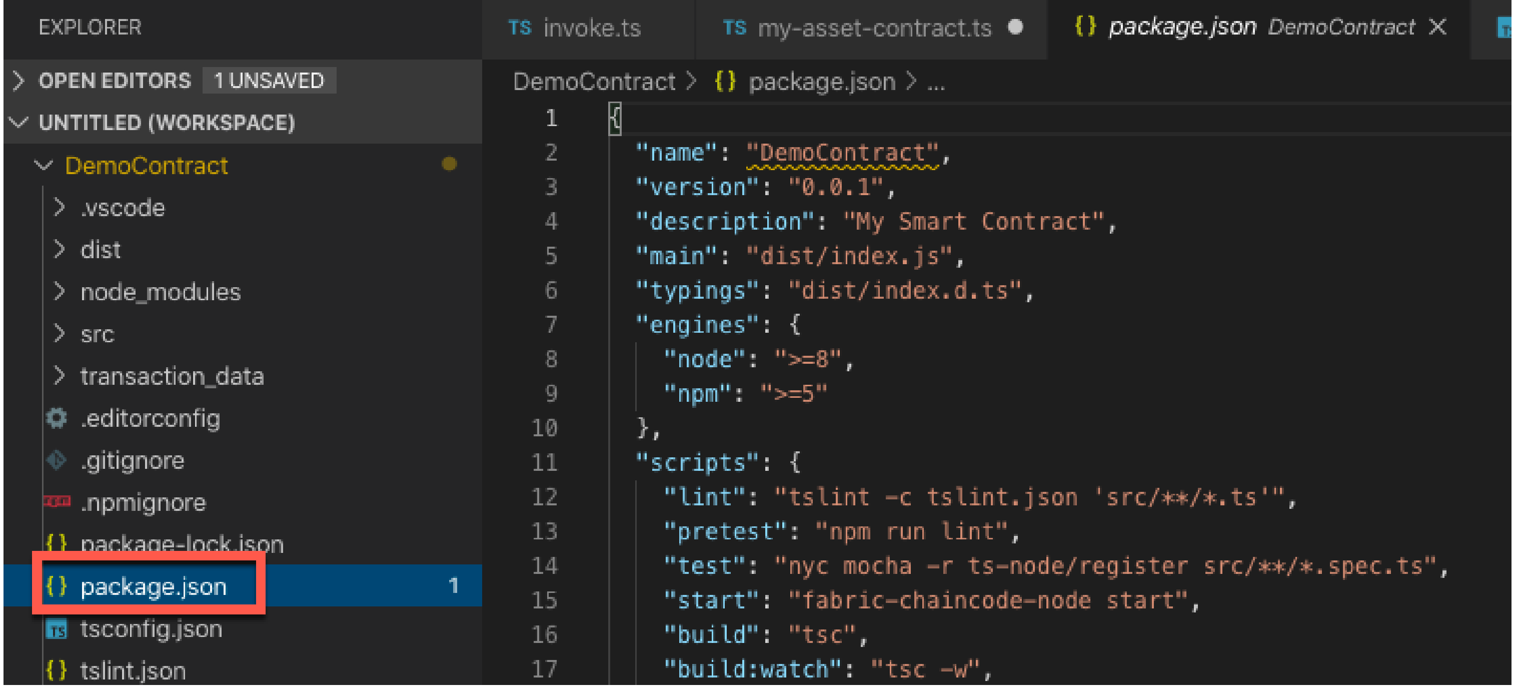
There should be no compilation errors.

Before we can package our new smart contract we need to update its version number. In a development environment for production systems an automated process would typically do this for us, but here we will update the necessary file manually.

☐ A6.4: Switch to the editor for the DemoContract *package.json* file.

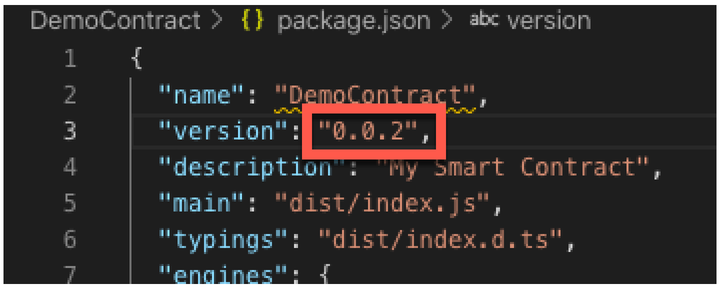
Again, this should be already loaded from earlier tutorials. If not, use the Explorer sidebar to navigate to package.json in the root of the DemoContract project.

Take care to load the *DemoContract* copy of the file; you will recall that we created another package.json for DemoApplication.



```
{
  "name": "DemoContract",
  "version": "0.0.1",
  "description": "My Smart Contract",
  "main": "dist/index.js",
  "typings": "dist/index.d.ts",
  "engines": {
    "node": ">=8",
    "npm": ">=5"
  },
  "scripts": {
    "lint": "tslint -c tslint.json 'src/**/*.ts'",
    "pretest": "npm run lint",
    "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
    "start": "fabric-chaincode-node start",
    "build": "tsc",
    "build:watch": "tsc -w",
```

☐ A6.5: Edit the value of the version tag to "0.0.2".



```
{
  "name": "DemoContract",
  "version": "0.0.2",
  "description": "My Smart Contract",
  "main": "dist/index.js",
  "typings": "dist/index.d.ts",
  "engines": {
```

☐ A6.6: Save the changes ('File' -> 'Save').

In the next section we will deploy the new smart contract to our peer.

☐ A6.7: Expand the next section of the tutorial to continue.

▸ **Deploy the upgraded smart contract**

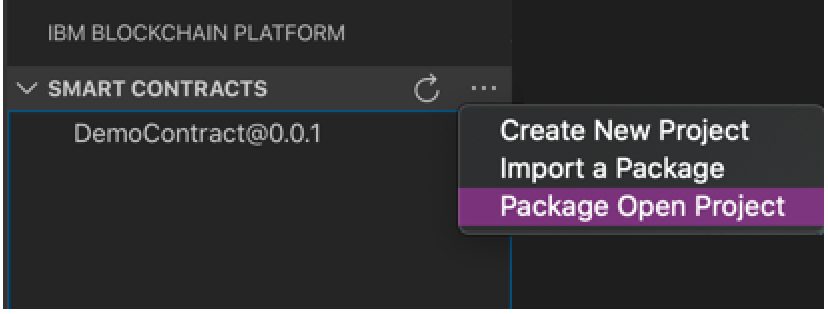
In this section we will *package* the smart contract as a deployable container, *install* the package on the peer then *instantiate* it (or rather, because the earlier version already exists, *upgrade* it).

You will recall from tutorial A3: Deploying a smart contract that you can do these three actions in a single 'instantiate' action when using the IBM Blockchain Platform VS Code extension. This time however we will do the steps individually, which gives us greater flexibility over the process.

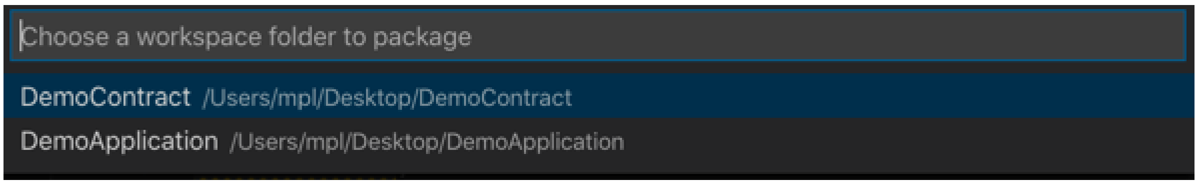
Package the smart contract

☐ A6.8: Click the IBM Blockchain Platform sidebar icon to show the IBM Blockchain Platform sidebar.

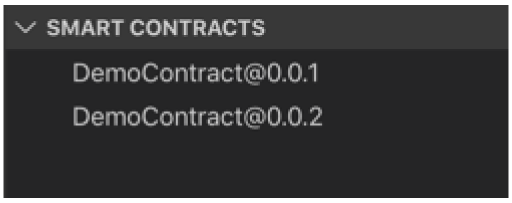
☐ A6.9: Hover the mouse over the Smart Contracts view, click '...' and select 'Package Open Project'.



A6.10: Select 'DemoContract'.


Choose a workspace folder to package
DemoContract /Users/mpl/Desktop/DemoContract
DemoApplication /Users/mpl/Desktop/DemoApplication

After a brief pause while the packaging completes, the newer version of DemoContract will be shown in the Smart Contracts view underneath the older one:



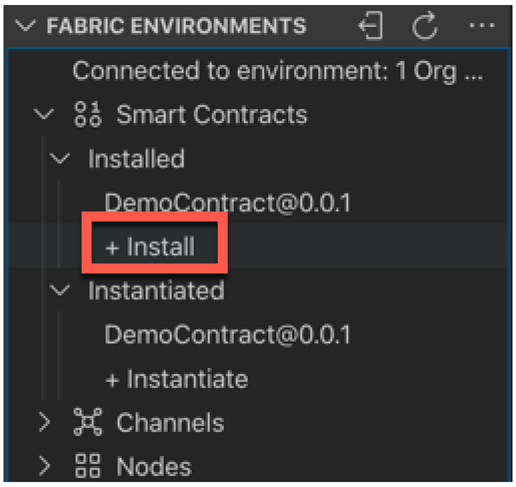
Install the smart contract

Installation of the smart contract needs to take place on every peer that will use it to endorse transactions. Endorsement is the process of signing an assertion to the rest of the network of the output of the transaction: what was read and written in the world state, for example.

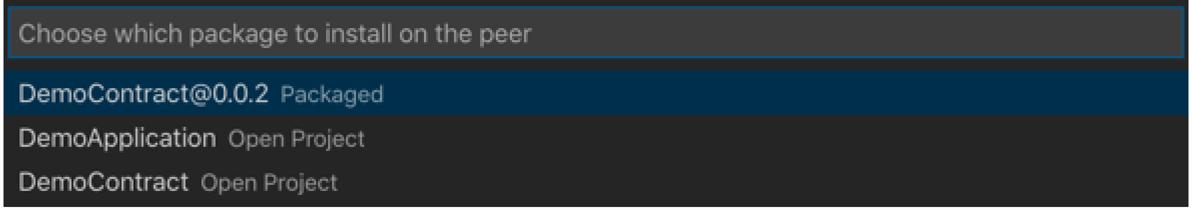
In our network, there is only one peer and so this is largely immaterial; that peer needs to have an installed copy of the smart contract.

A6.11: In the Fabric Environments view, click '+ *Install*' in the 'Smart Contracts' -> 'Installed' section.

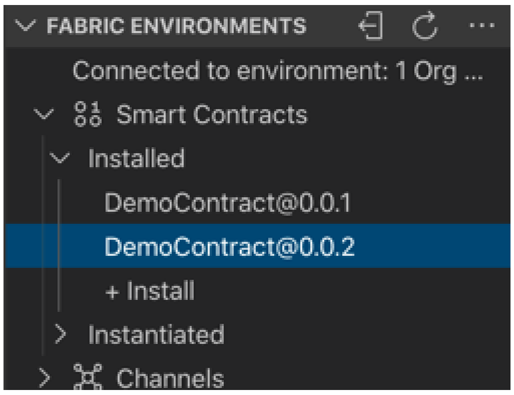
The local Fabric environment needs to be running in order to do this. If it is stopped for any reason, you will need to first click the '1 Org Local Fabric' environment in the Fabric Environments view to start it.



A6.12: Select 'DemoContract@0.0.2'.


Choose which package to install on the peer
DemoContract@0.0.2 Packaged
DemoApplication Open Project
DemoContract Open Project

After a brief pause, you will see DemoContract@0.0.2 appear underneath the Installed list in the Fabric Environments view.


FABRIC ENVIRONMENTS
Connected to environment: 1 Org ...
Smart Contracts
Installed
DemoContract@0.0.1
DemoContract@0.0.2
+ Install
Instantiated
Channels

Upgrade the smart contract

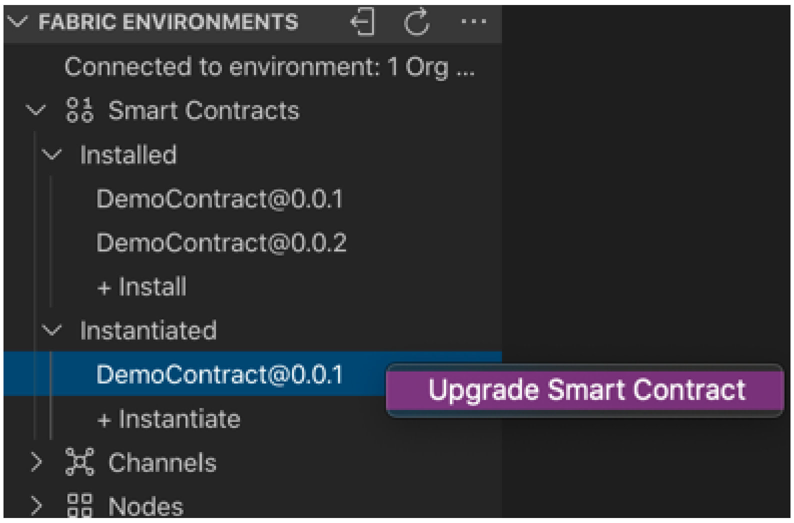
As you can see from the Installed list, multiple versions of a smart contract can be installed at the same time. However, for a given smart contract only one version can be instantiated and running.

We now need to tell Hyperledger Fabric to use version 0.0.2 of the smart contract. This process is called ***upgrading***, although it applies to any switch between versions; it is possible to upgrade from 0.0.2 back to 0.0.1, for example.

Unlike installation, upgrading (or instantiating) a smart contract only needs to be done once per network (channel). Again, this is immaterial for our network as we only have one peer anyway.

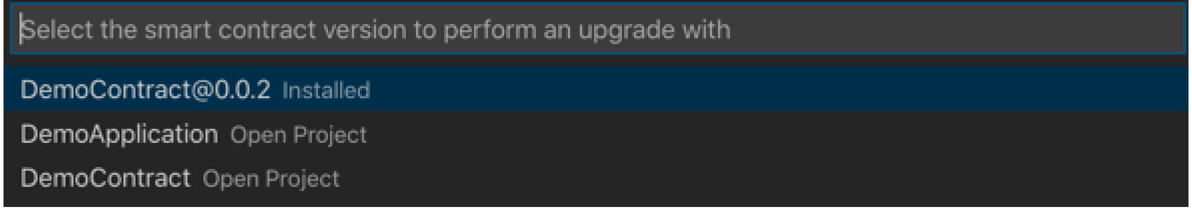
☐ `A6.13`: In the Fabric Environments view, right-click the ***instantiated*** DemoContract@0.0.1 and select 'Upgrade Smart Contract'.

Take care not to click on one of the ***installed*** DemoContracts by mistake.


FABRIC ENVIRONMENTS
Connected to environment: 1 Org ...
Smart Contracts
Installed
DemoContract@0.0.1
DemoContract@0.0.2
+ Install
Instantiated
DemoContract@0.0.1
Upgrade Smart Contract
+ Instantiate
Channels
Nodes

You now need to select which smart contract you want to replace it with.

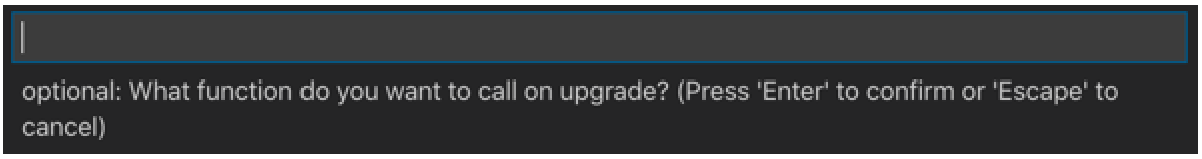
☐ `A6.14`: Select 'DemoContract@0.0.2 Installed'.


Select the smart contract version to perform an upgrade with

DemoContract@0.0.2 Installed
DemoApplication Open Project
DemoContract Open Project

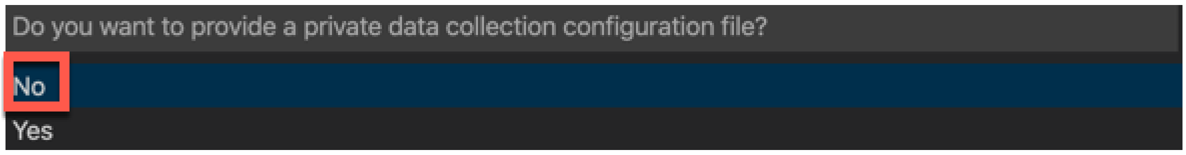
There are no reinitialization functions that we need to call in this smart contract.

☐ A6.15: Press Enter to decline calling a function.


optional: What function do you want to call on upgrade? (Press 'Enter' to confirm or 'Escape' to cancel)

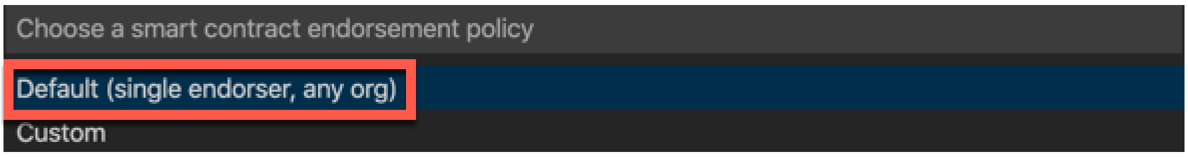
We are not using private data collections in this tutorial.

☐ A6.16: Press Enter to decline to provide a private data collection configuration file.


Do you want to provide a private data collection configuration file?

[✓ No]
Yes

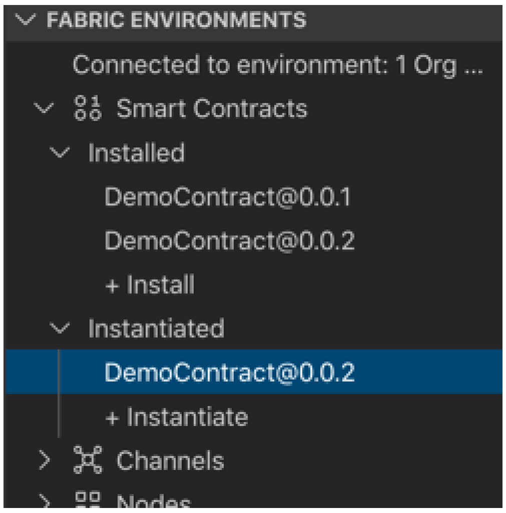
☐ A6.17: Click 'Default (single endorser, any org)' as the smart contract endorsement policy.


Choose a smart contract endorsement policy

[✓ Default (single endorser, any org)]
Custom

The smart contract will now be upgraded. This may take a minute to complete.

Once complete, you'll see that DemoContract@0.0.1 in the Instantiated section of the Fabric Environments view is replaced with DemoContract@0.0.2.



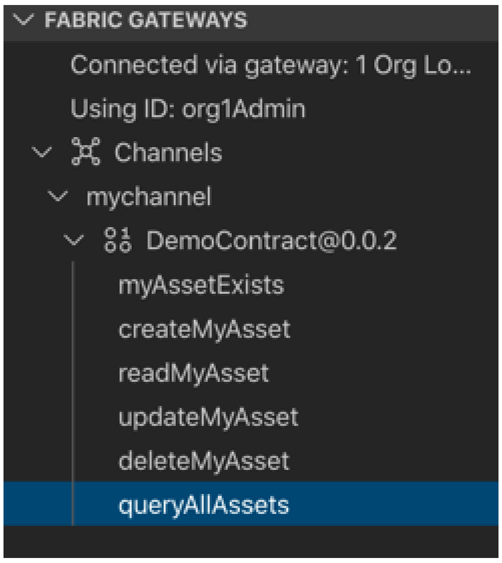
☐ A6.18: Expand the next section of the tutorial to continue.

▶ **Try out the upgraded smart contract**

Finally in this tutorial we will try out our new *queryAllAssets* transaction to make sure it works. We will do this using the Fabric Gateways view.

☐ A6.19: In the connected Fabric Gateways view, expand 'Channels' -> 'mychannel' -> 'DemoContract@0.0.2'.

You will see the new *queryAllAssets* transaction listed among the others.



If you have completed all the previous steps in this set of tutorials, your blockchain world state will only contain one asset at this point ('002'), as we deleted asset '001' at the end of tutorial A4: Invoking a smart contract from VS Code.

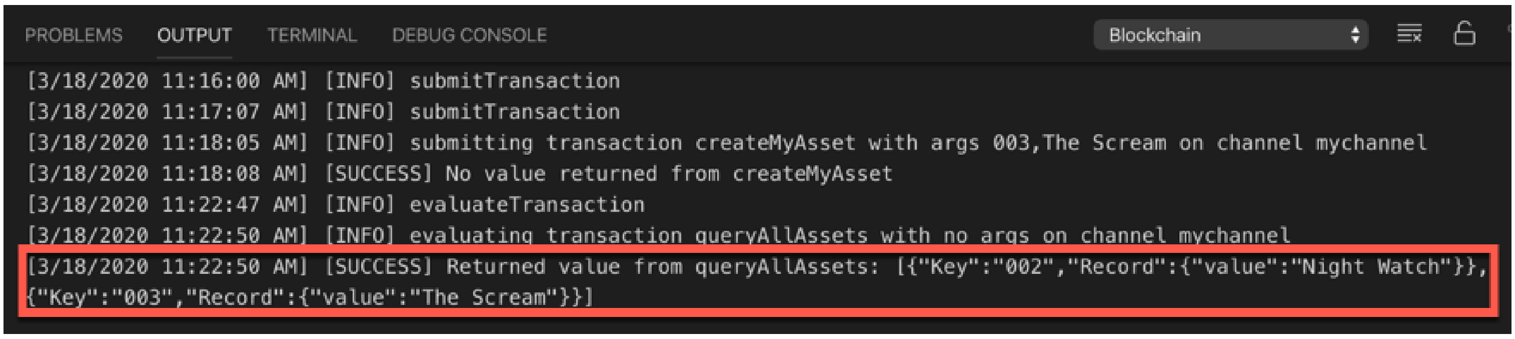
Therefore, to prove that we can return multiple values from our new transaction, we will first create a new asset '003'.

☐ `A6.20`: Right-click the *createMyAsset* transaction and select 'Submit Transaction'. Create an asset with the input parameters `["003","The Scream"]`. There is no transient data.

With the new asset created, we will now try out the queryAllAssets transaction. It is a read-only transaction and so we can invoke it using the *evaluate* option.

☐ `A6.21`: Right-click the *queryAllAssets* transaction and select 'Evaluate Transaction'. Press Enter twice to select the defaults for both the input parameters (there are none) and transient data.

You will see the results of the transaction displayed in the Output view; particularly, records for asset '002' and '003'. (Close the "Successfully submitted transaction" notifications if the output is obscured.)



```
[3/18/2020 11:16:00 AM] [INFO] submitTransaction
[3/18/2020 11:17:07 AM] [INFO] submitTransaction
[3/18/2020 11:18:05 AM] [INFO] submitting transaction createMyAsset with args 003,The Scream on channel mychannel
[3/18/2020 11:18:08 AM] [SUCCESS] No value returned from createMyAsset
[3/18/2020 11:22:47 AM] [INFO] evaluateTransaction
[3/18/2020 11:22:50 AM] [INFO] evaluating transaction queryAllAssets with no args on channel mychannel
[3/18/2020 11:22:50 AM] [SUCCESS] Returned value from queryAllAssets: [{"Key":"002","Record":{"value":"Night Watch"}},
{"Key":"003","Record":{"value":"The Scream"}}]
```

Summary

In this tutorial we looked at the smart contract upgrade process in Hyperledger Fabric v1.x. We started by making a change to our existing smart contract, then we packaged it, installed it on our peer and upgraded the instantiated version of it. We then tried it out.

In the next tutorial we will look at some features in the IBM Blockchain Platform VS Code extension that facilitate the debugging of smart contracts.

→ **A7: Debugging a smart contract**