



## Tutorial A6: Upgrading a smart contract

---

Estimated time: 20 minutes

**Note:** This tutorial is 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 previous tutorial, we built and tested a TypeScript application that interacted with a Hyperledger Fabric network. In this tutorial we will:

- Make a change to a 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.

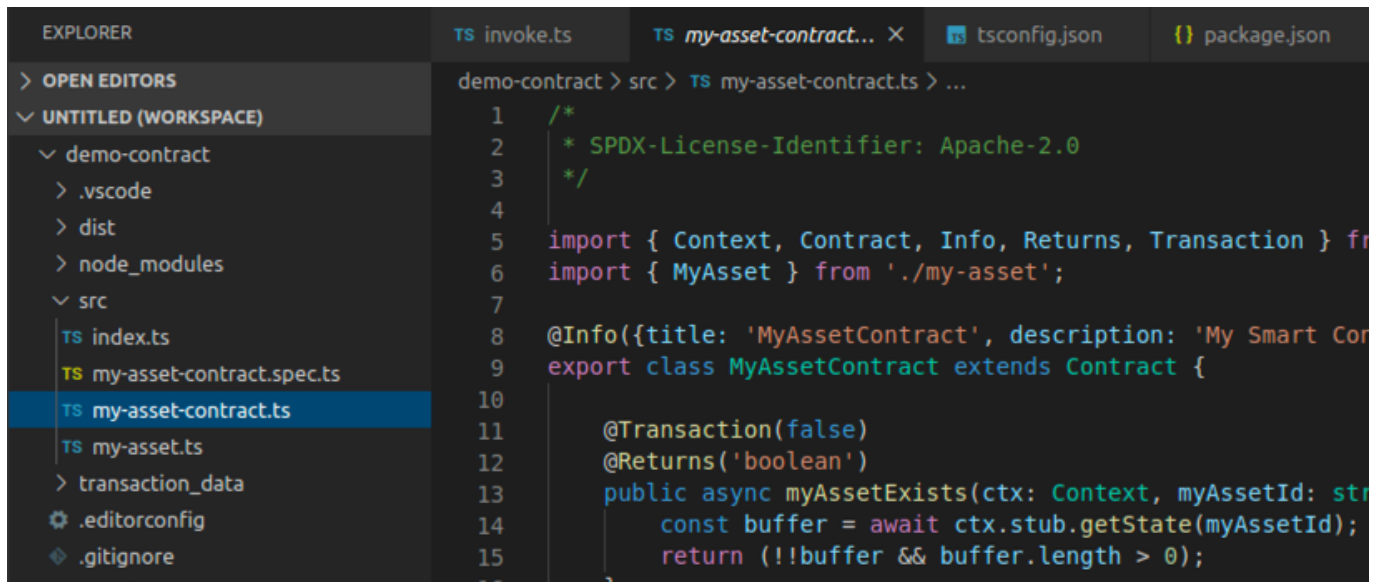
☐ **A6.1:** Expand the first section below to get started.

---

### ► 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 side bar to navigate to *my-asset-contract.ts* in the *src* folder of the *demo-contract* project.



```
1  /*
2   * SPDX-License-Identifier: Apache-2.0
3   */
4
5  import { Context, Contract, Info, Returns, Transaction } from '@openzeppelin/contracts';
6  import { MyAsset } from './my-asset';
7
8  @Info({title: 'MyAssetContract', description: 'My Smart Contract'})
9  export class MyAssetContract extends Contract {
10
11     @Transaction(false)
12     @Returns('boolean')
13     public async myAssetExists(ctx: Context, myAssetId: string): Promise<boolean> {
14         const buffer = await ctx.stub.getState(myAssetId);
15         return (!!buffer && buffer.length > 0);
16     }
17 }
```

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

A smart contract package has a version, and as smart contracts within a package evolve, the version number of the package should be incremented to reflect this change. So far, we've been working with version 0.0.1 of the demo-contract package.

We're going to learn about the smart contract package upgrade process as we enhance the MyAsset smart contract within the package. We are going to increment the package version to reflect this change.

### Smart contract evolution

Because the transactions created by a smart contract live forever on the blockchain, when a package is re-versioned, all the previously created states persist unchanged, and accessible by the new package. It means that a smart contract needs to maintain data compatibility between version boundaries as it will be working with state data created in all previous versions.

Practically speaking, it makes sense to use extensible data structures where possible, and to have sensible defaults when values are missing.

Our new transaction will not modify any data structures, so we do not need to consider cross-version compatibility.

**A6.3:** Using copy and paste, insert the following method after the closing brace of the deleteMyAsset 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()) {
```

```

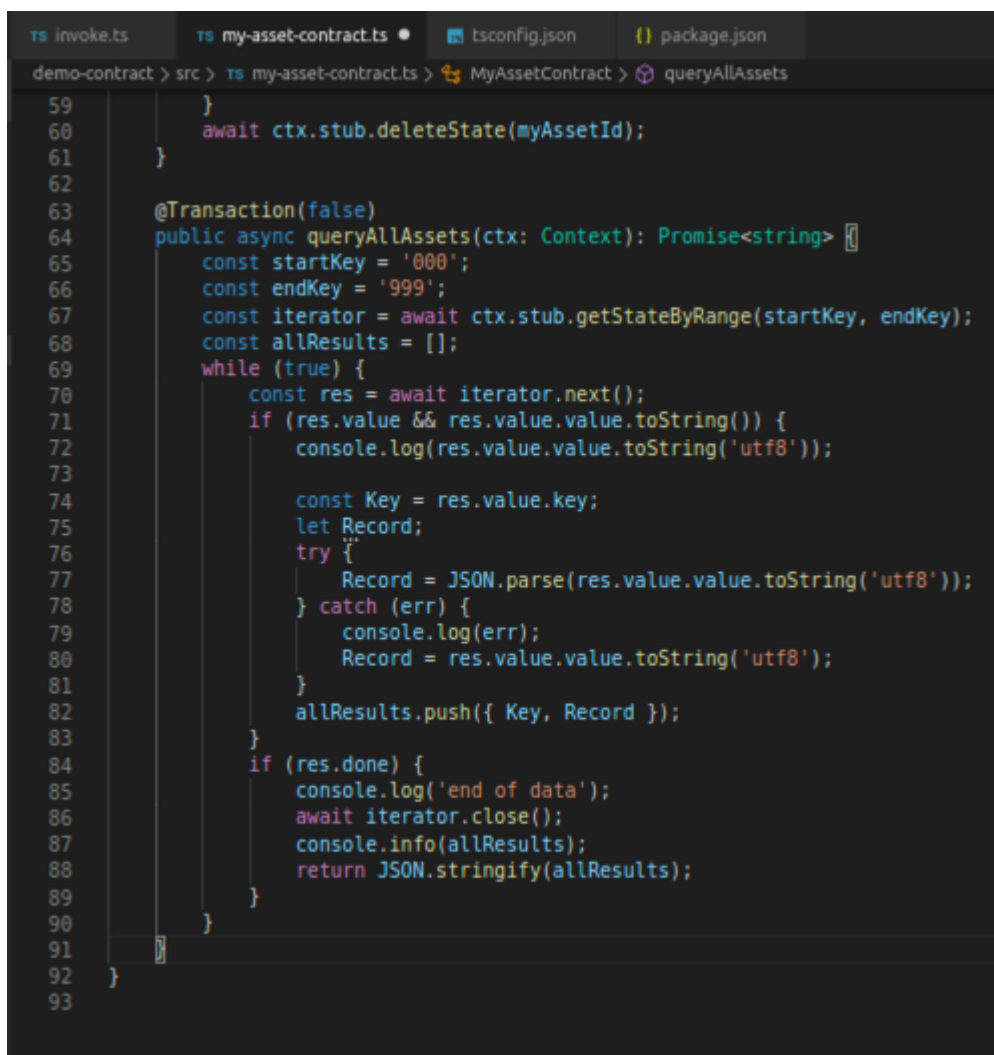
        console.log(res.value.value.toString('utf8'));

        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
demo-contract > src > TS my-asset-contract.ts > MyAssetContract > queryAllAssets
59     }
60     await ctx.stub.deleteState(myAssetId);
61 }
62
63 @Transaction(false)
64 public async queryAllAssets(ctx: Context): Promise<string> {
65     const startKey = '000';
66     const endKey = '999';
67     const iterator = await ctx.stub.getStateByRange(startKey, endKey);
68     const allResults = [];
69     while (true) {
70         const res = await iterator.next();
71         if (res.value && res.value.value.toString()) {
72             console.log(res.value.value.toString('utf8'));
73
74             const Key = res.value.key;
75             let Record;
76             try {
77                 Record = JSON.parse(res.value.value.toString('utf8'));
78             } catch (err) {
79                 console.log(err);
80                 Record = res.value.value.toString('utf8');
81             }
82             allResults.push({ Key, Record });
83         }
84     }
85     if (res.done) {
86         console.log('end of data');
87         await iterator.close();
88         console.info(allResults);
89         return JSON.stringify(allResults);
90     }
91 }
92 }
93

```

- 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 the package version number. In a production environment, an automated process would typically do this, but we will update the necessary file manually.

### Updating smart contract package versions is mandatory

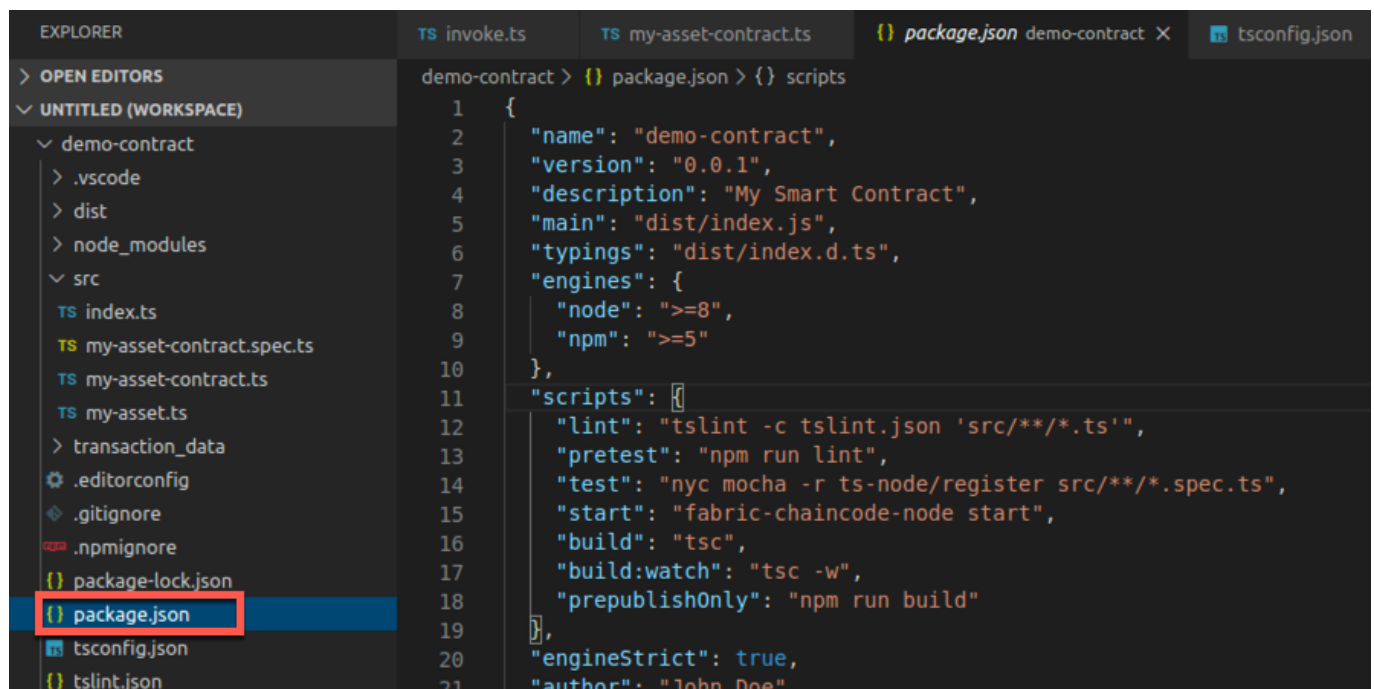
You must always create a new smart contract package version when packaging a smart contract, even when working in your own private development environment. That's because each package runs in its own Docker container which is named according to its package name and version. These containers persist.

Of course, it is even more important to use unique version numbers when distributing smart contracts around a network, because if different peers run different code, transactions will fail.

- A6.5: Switch to the editor for the demo-contract *package.json* file.

Again, this should be already loaded from earlier tutorials. If not, use the Explorer side bar to navigate to *package.json* in the root of the demo-contract project.

Take care to load the *demo-contract* copy of the file; you will recall that we created another *package.json* for *demo-application*.



```
1 {
2   "name": "demo-contract",
3   "version": "0.0.1",
4   "description": "My Smart Contract",
5   "main": "dist/index.js",
6   "typings": "dist/index.d.ts",
7   "engines": {
8     "node": ">=8",
9     "npm": ">=5"
10  },
11  "scripts": {
12    "lint": "tslint -c tslint.json 'src/**/*.ts'",
13    "pretest": "npm run lint",
14    "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
15    "start": "fabric-chaincode-node start",
16    "build": "tsc",
17    "build:watch": "tsc -w",
18    "prepublishOnly": "npm run build"
19  },
20  "engineStrict": true,
21  "author": "John Doe",
```

- A6.6: Edit the value of the version tag to "0.0.2".

```
demo-contract > {} package.json > version
1  {
2    "name": "demo-contract",
3    "version": "0.0.2",
4    "description": "My Smart Contract",
5    "main": "dist/index.js",
6    "typings": "dist/index.d.ts",
7    "engines": {
8      "node": ">=8",
9      "npm": ">=5"
10   },
11   "scripts": {
```

- A6.7: Save the changes ('File' -> 'Save').

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

- A6.8: Expand the next section of the tutorial to continue.

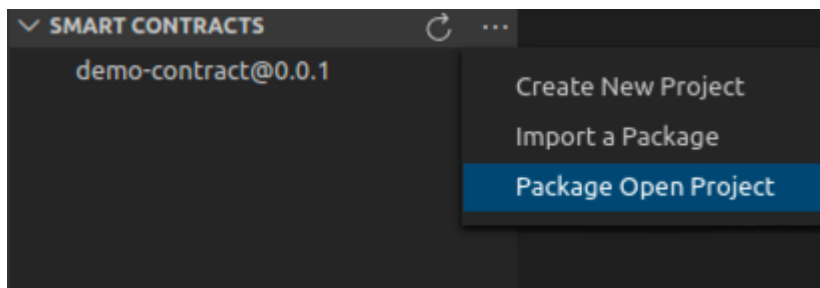
### ► Deploy the upgraded smart contract

In this section we will package the smart contract, install the new version on the peer, and then upgrade the package. Note that we are upgrading the package because it is *already* instantiated.

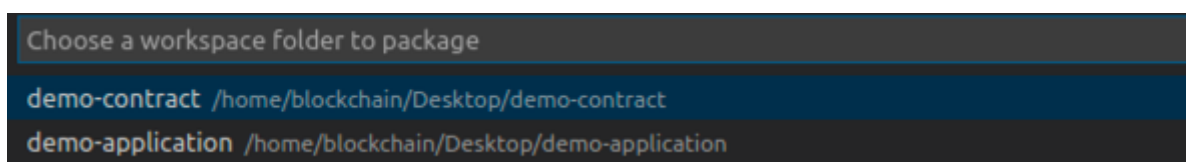
You will recall from tutorial [A3: Deploying a smart contract](#) that you can do these three actions in a single 'instantiate' action when using the IBM Blockchain Platform VS Code extension. This time however we will do the steps individually, which gives us greater control over the process.

### Package the smart contract

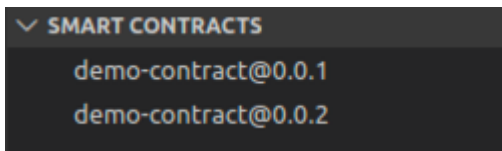
- A6.9: Click the IBM Blockchain Platform activity bar icon to show the IBM Blockchain Platform side bar.
- A6.10: Hover the mouse over the Smart Contracts view, click '...' and select 'Package Open Project'.



- A6.11: Select 'demo-contract'.




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



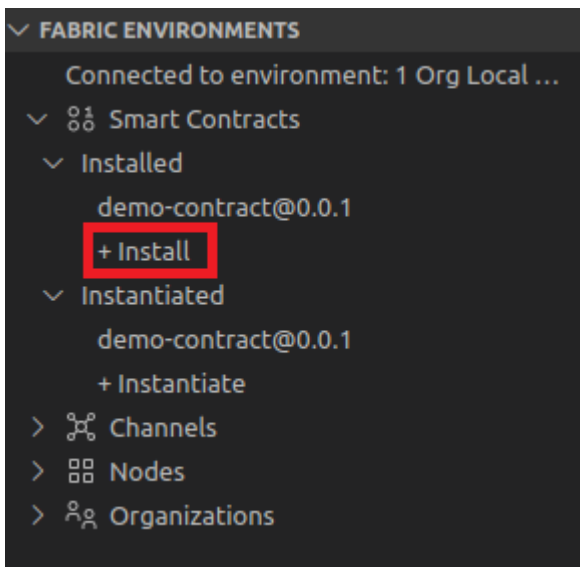
## Install the smart contract


A smart contract package needs to be installed on at least one peer of each organization required to sign transactions generated by the smart contracts contained within it. An endorsement policy is associated with each package, and describes which organizations must sign transactions for them to be considered valid.

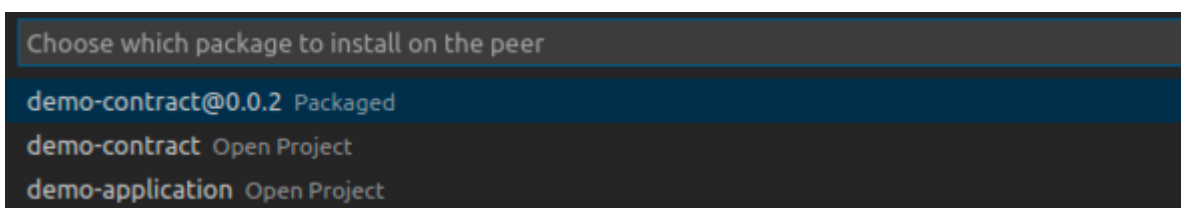
Our sample network only has one organization and a single peer, so there is only one possible endorsement policy -- that this organization must sign all transactions. Consequently this single peer needs to have an installed copy of the smart contract.

 A6.12: In the Fabric Environments view, click '+ Install' in the 'Smart Contracts' -> 'Installed' section.

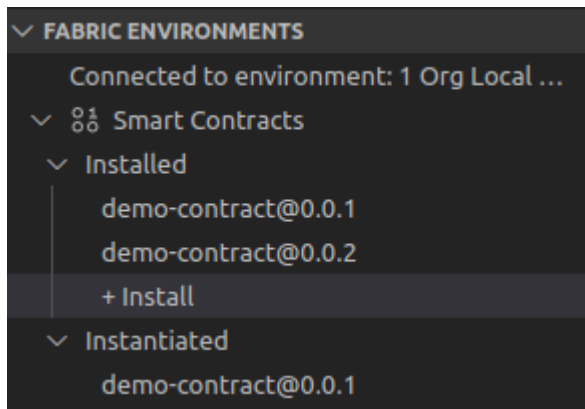
The local Fabric environment needs to be running in order to do this. If it is stopped for any reason, you will need to first click the '1 Org Local Fabric' environment in the Fabric Environments view to start it.



 A6.13: Select 'demo-contract@0.0.2'.



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




## Upgrade the smart contract

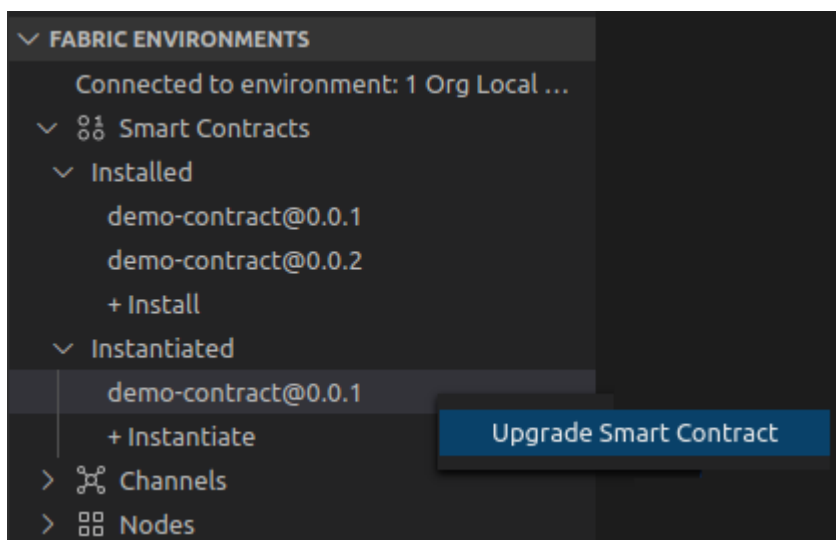
As you can see from the Installed list, multiple versions of a smart contract can be installed at the same time. However, for a given smart contract only one version can be instantiated.

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), regardless of the number of organizations.

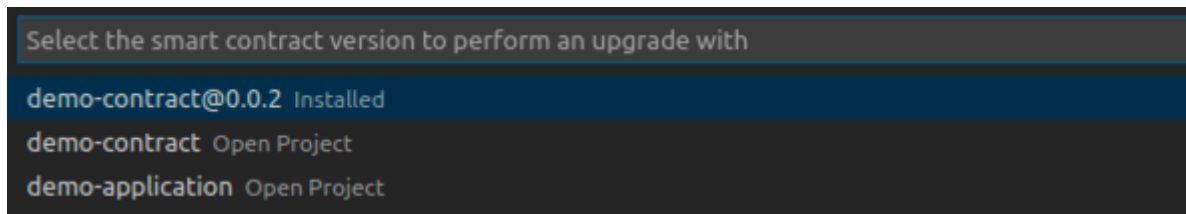
 **A6.14:** In the Fabric Environments view, right-click the *instantiated* demo-contract@0.0.1 and select 'Upgrade Smart Contract'.

Take care not to click on one of the *installed* demo-contracts by mistake.



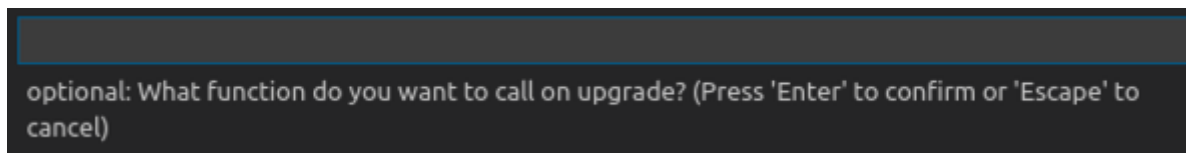
You now need to select which smart contract you want to replace it with.

 **A6.15:** Select 'demo-contract@0.0.2 Installed'.



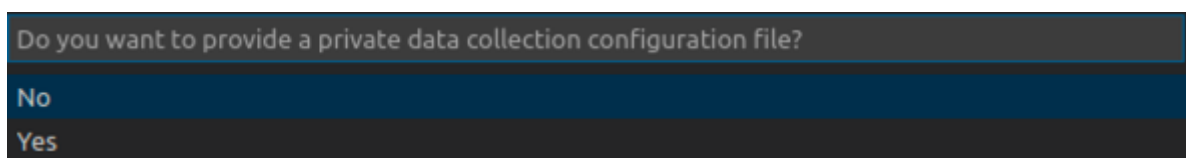
There are no reinitialization functions that we need to call in this smart contract.

- ☐ A6.16: Press Enter to skip calling a function.

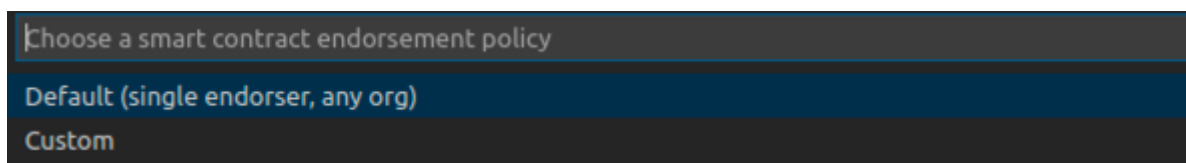


We are not using private data collections in this tutorial.

- ☐ A6.17: Press Enter to decline to provide a private data collection configuration file.

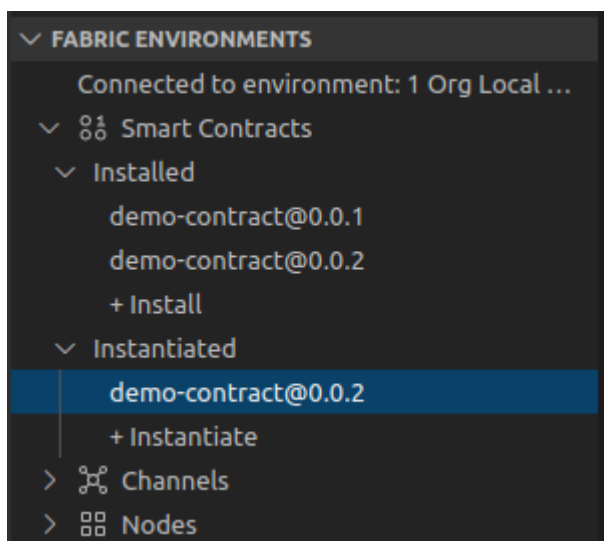


- ☐ A6.18: Click 'Default (single endorser, any org)' as the smart contract endorsement policy.



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

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



- ☐ A6.19: Expand the next section of the tutorial to continue.

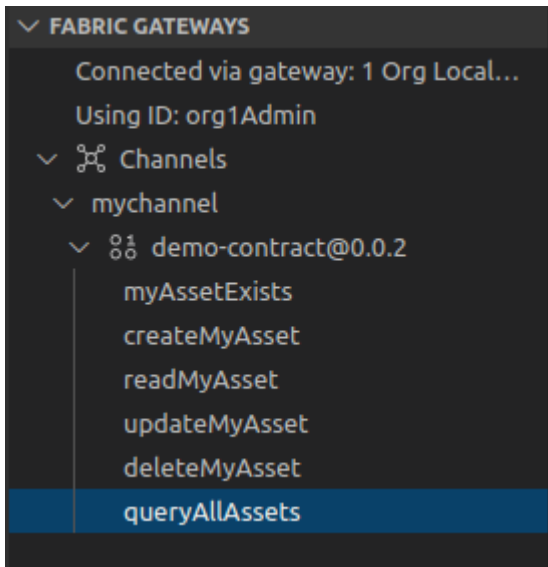


## ► Try out the upgraded smart contract

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

**A6.20:** In the connected Fabric Gateways view, expand 'Channels' -> 'mychannel' -> 'demo-contract@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](#).

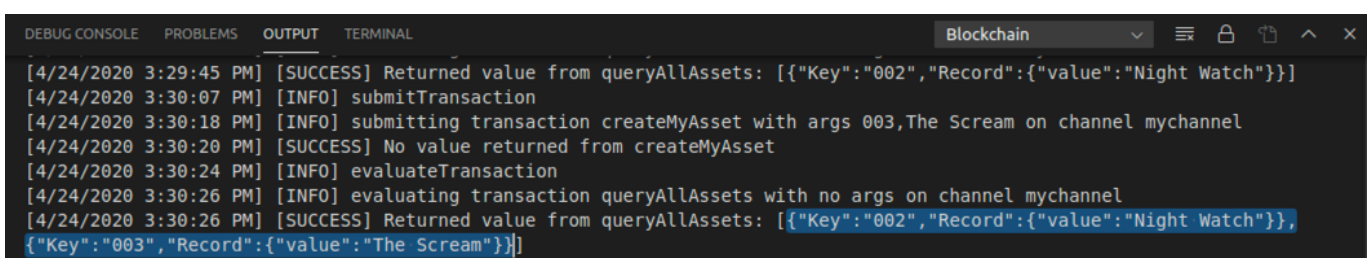
Therefore, to prove that we can return multiple values from our new transaction, we will first create a new asset '003'.

**A6.21:** 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.

**A6.22:** 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.)



Congratulations, you queried all the assets on the ledger!

## 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 makes the debugging of smart contracts easier.

---

→ **A7: Debugging a smart contract**