**IBM Blockchain Platform**

← A1: Introduction

# Tutorial A2: Creating a smart contract

Estimated time: `10 minutes`

A *smart contract* contains the business rules required to implement a transaction. In Hyperledger Fabric, a smart contract is a piece of code that interacts with a database called a *world state*. Smart contracts are run by multiple endorsing peers on the network. The network then agrees on the order and output of each transaction.

Smart contracts can be built and tested using the IBM Blockchain Platform VS Code extension. In this tutorial you will:

- Create a new smart contract project
- Implement a basic smart contract using a standard template
- Understand what the smart contract does

In order to successfully complete this tutorial, you must have the IBM Blockchain Platform VS Code extension installed. You are recommended to start with an empty workspace.

Remember to complete every task that begins with a blue square like this one:

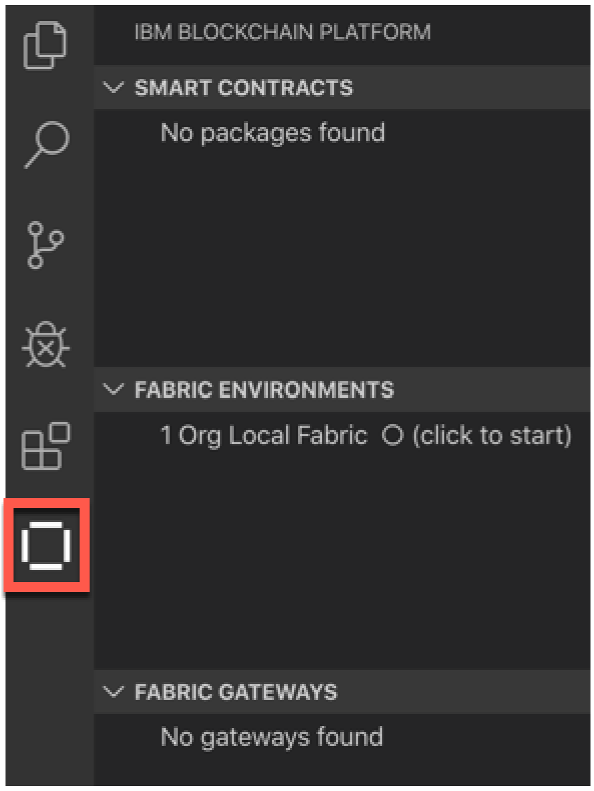
□ `A2.1`: Expand the first section below to get started.

## ▶ Create a smart contract project

When working with Hyperledger Fabric assets in the IBM Blockchain Platform VS Code extension, it is usually convenient to show the IBM Blockchain Platform sidebar, which contains the Smart Contracts, Fabric Environments, Fabric Gateways and Fabric Wallets views.

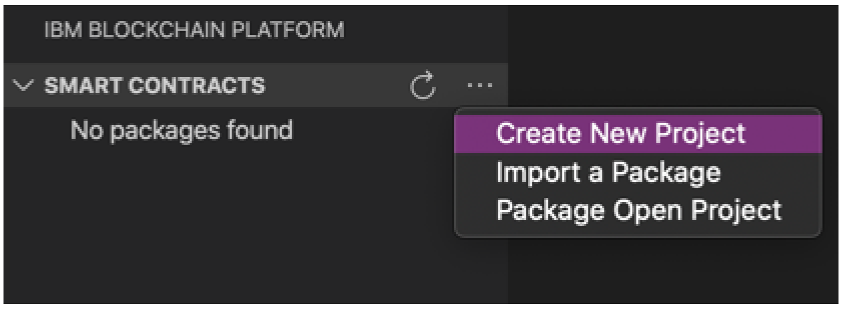
You can show the sidebar by clicking on the IBM Blockchain Platform icon in the VS Code activity bar. However, note that the icon is a toggle: if you click on it while the sidebar is already shown, the sidebar will be hidden.

□ `A2.2`: If the IBM Blockchain Platform sidebar is not already shown, click on IBM Blockchain Platform icon in the activity bar.

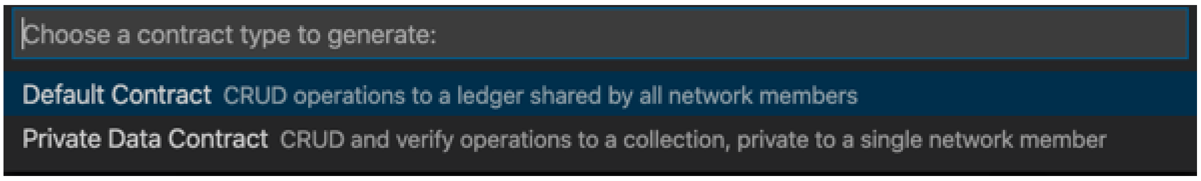


We will now create a smart contract project that will house the files we need for our smart contract. IBM Blockchain Platform will create for us a skeleton smart contract that we can customize later.

❏ A2.3: Move the mouse over the title bar of the Smart Contracts view, click the "..." that appears and select "Create New Project".

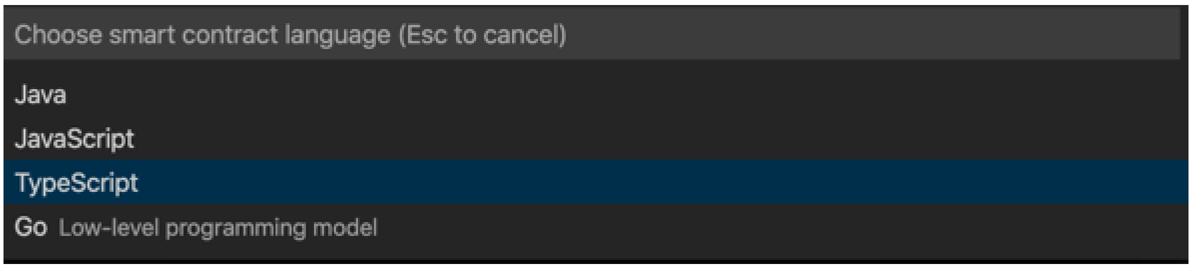


❏ A2.4: Press Enter to accept the Default Contract type.


Choose a contract type to generate:
Default Contract CRUD operations to a ledger shared by all network members
Private Data Contract CRUD and verify operations to a collection, private to a single network member

In this tutorial we will be using the TypeScript language.

❏ A2.5: Click 'Typescript'.


Choose smart contract language (Esc to cancel)

Java

JavaScript

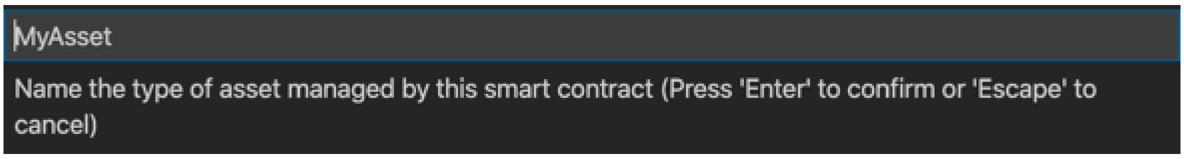
TypeScript

Go Low-level programming model

The skeleton smart contract will provide us with the ability to share a single asset type on the blockchain. In this context, an asset type is a group of related objects (e.g. Cars), which by convention begins with a capital letter.

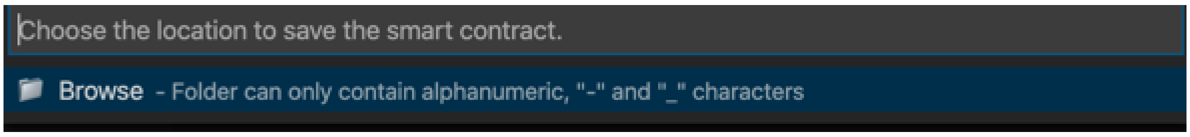
We can extend the smart contract with additional asset types if we wish; the world state is a simple key-value store whose data format is up to the developer. For now however, we will just accept the default asset type presented to us.

☐ A2.6: Press Enter to accept the default asset type ("MyAsset").


MyAsset

Name the type of asset managed by this smart contract (Press 'Enter' to confirm or 'Escape' to cancel)

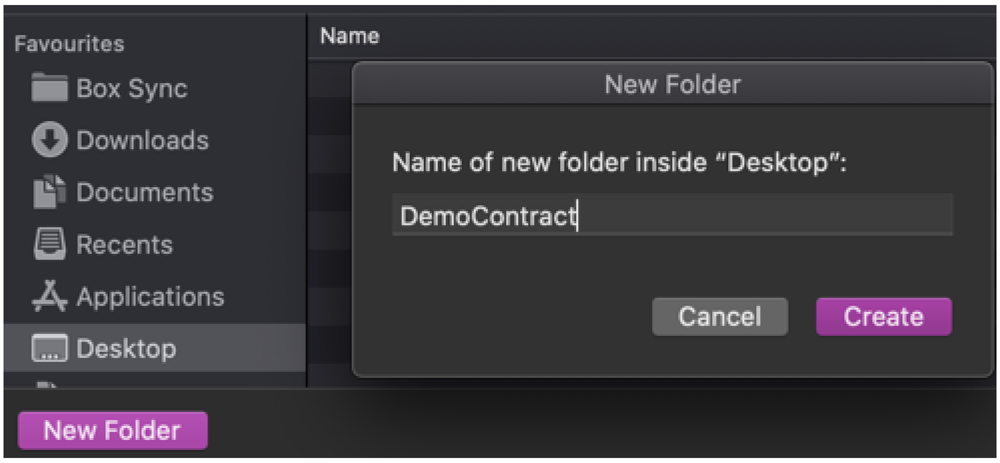
☐ A2.7: Click Browse to choose a target location of the project on the file system.


Choose the location to save the smart contract.

Browse - Folder can only contain alphanumeric, "-" and "_" characters

Navigate to a suitable folder on your file system if necessary.

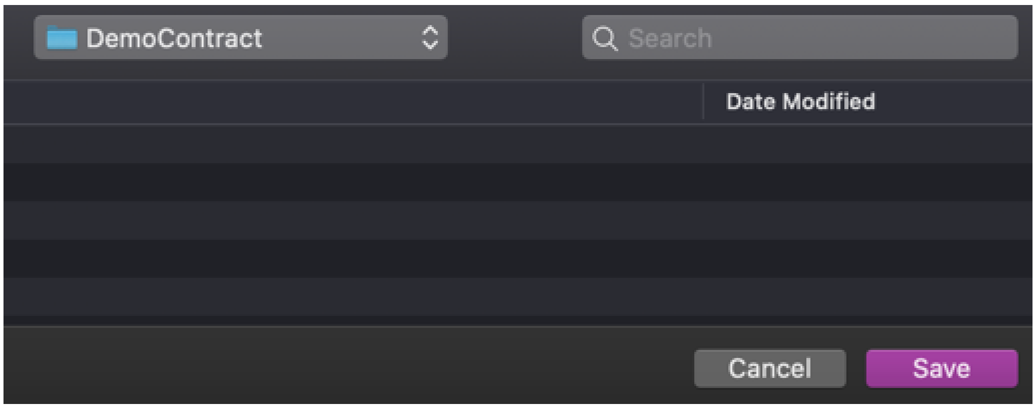
☐ A2.8: Click "New folder" to create a new folder to store the smart contract project, and name it "DemoContract".



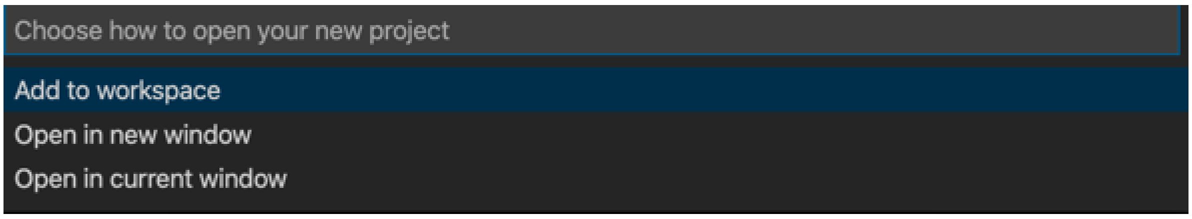
Name of new folder inside "Desktop":

DemoContract

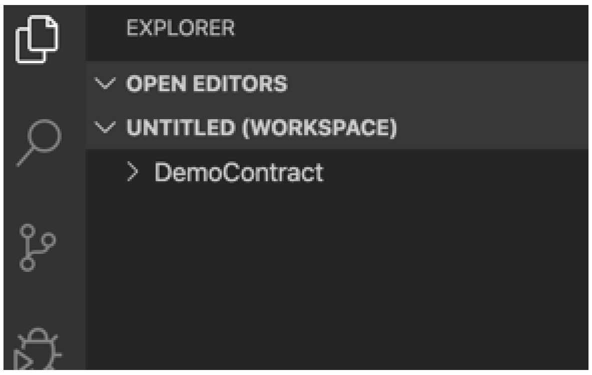
☐ A2.9: Click Save to select the new folder as the project root.



☐ A2.10: Select "Add to workspace" to tell IBM Blockchain Platform to add the project to your workspace.



Generating the smart contract project will take up to a minute to complete. When it has successfully finished, the IBM Blockchain Platform sidebar will be hidden and the Explorer sidebar will be shown. The Explorer sidebar will show the new project that has been created.



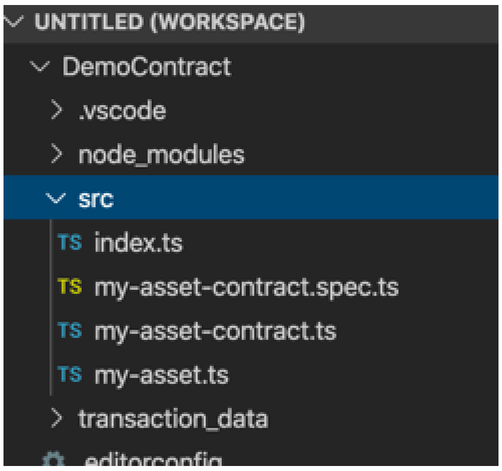
☐ A2.11: Expand the next section of the tutorial to continue.

---

▶ **Learn about the smart contract**

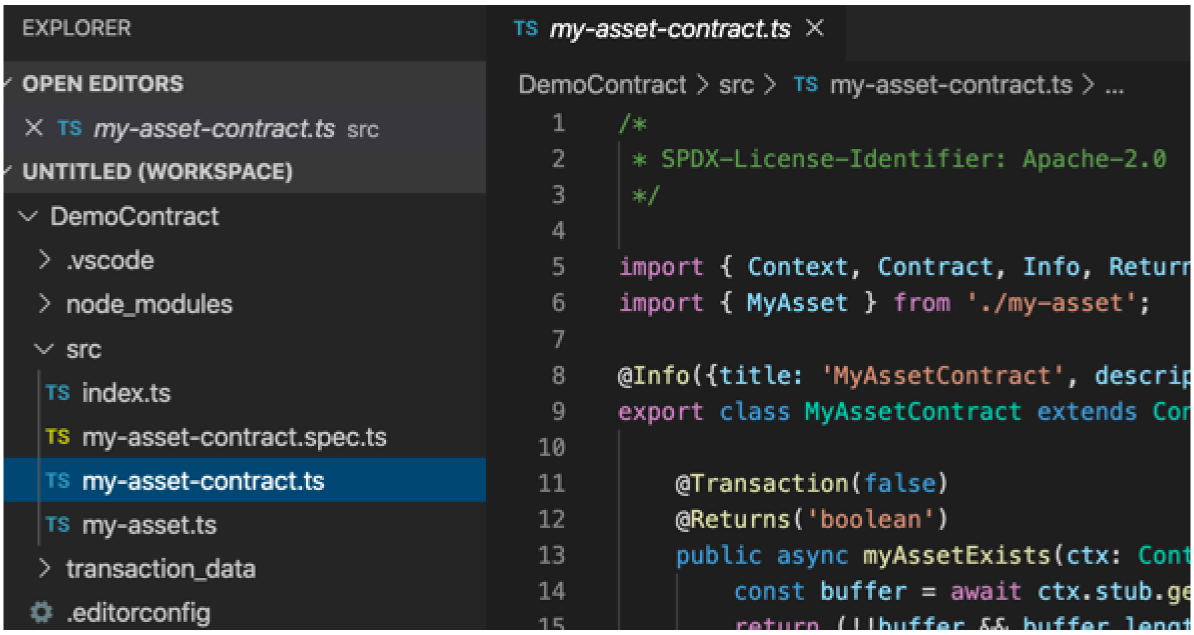
We will now look at the files that have been created to see what they do.

☐ A2.12: In the Explorer sidebar, expand "DemoContract" -> "src".



The smart contract is contained within the 'my-asset-contract.ts' file. The file name has been generated from the asset type you gave earlier.

☐ `A2.13`: Click on 'my-asset-contract.ts' to load it in the VS Code editor.



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

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

@Info({title: 'MyAssetContract', descrip
export class MyAssetContract extends Con

    @Transaction(false)
    @Returns('boolean')
    public async myAssetExists(ctx: Cont
        const buffer = await ctx.stub.ge
```

Have a read through the code.

The import statement at the top of the file makes the Hyperledger Fabric classes available.

```
import { Context, Contract, Info, Returns, Transaction } from 'fabric-contract-api';
```

The smart contract is a standard piece of TypeScript. The only line that identifies it as a smart contract is in the class definition itself:

```
export class MyAssetContract extends Contract {
```

The *extends* clause states that this is a Hyperledger Fabric smart contract using the Contract class imported earlier.

The rest of the file implements the transaction logic that the smart contract exposes.

## Transactions

In Hyperledger Fabric, a transaction is a request to run a smart contract method using a defined set of parameters; methods in the smart contract code provide the implementation of transaction logic.

Any method that is prefixed with:

```
@Transaction()
```

is a transaction that can be invoked by applications using the blockchain. Transactions can be described as read-only (*@transaction(false)*), or read/write (*@transaction(true)* or simply *@transaction()*).

Look at the first method:

```
@Transaction(false)
public async myAssetExists(ctx: Context, myAssetId: string):
Promise<boolean> {
```

This method, like any other transaction implementation, takes as input a *Context* object and a set of other parameters that describe the variant parts of the transaction. For example, a transaction to determine whether an asset called *myCar* exists could map down to the invocation of the myAssetExists method with a myAssetId input parameter of 'myCar'.

## Working with the world state

In Hyperledger Fabric, each transaction can interact with a key/value store called the *world state*, which contains the data shared with the rest of the network. You might consider the output from each transaction as being the request to read from or write to this key/value store.

The Context object (ctx) provides the developer with a means of interacting with this world state. The *myAssetExists* method contains the following statement:

```
const buffer = await ctx.stub.getState(myAssetId);
```

The *getState* method returns from the world state the current value associated with the key described by *myAssetId*. This transaction does not change any state itself, and as such is marked as read-only (*@transaction(false)*).

Now take a look at the second method, which implements the read/write transaction *createMyAsset*.

```
@Transaction()
public async createMyAsset(ctx: Context, myAssetId: string, value:
string): Promise<void> {
```

This ends with the line:

```
await ctx.stub.putState(myAssetId, buffer);
```

This is the request to write to the world state a record whose key is the value of the input parameter *myAssetId*, and whose value is the contents of *buffer*, which is generated in the preceding lines.

## Smart contract determinism

Note that the *putState* method does not update the world state directly. Behind the putState method is a highly involved process of endorsement and consensus with the other nodes on the blockchain network.

As developers we generally don't need to worry about this, except that each endorsing peer on the network will run the same code, with the same parameters, and agree on any updates to the world state. Therefore, as our code will be run multiple times for a single transaction, whatever we write needs to be *deterministic*.

This means that for the same input, our code needs to output the same results to the world state. When different peers run non-deterministic transactions, they will probably disagree over the updates to the world state and consequently the transactions will fail.

This means that attempting to update the world state with, for example, a random number, timestamp or some other transient value is not recommended.

## Summary

In this tutorial we have generated a basic smart contract.

We looked at the smart contract code and how each transaction it defines is implemented as a single method. These call the *putState* and *getState* methods, which provide the foundations of sharing data between network members on the blockchain.

In order to test these transactions out, we must first deploy them to an instance of Hyperledger Fabric; we will do that in the next tutorial.

---

→ **A3: Deploying a smart contract**