IBM Blockchain Platform

← **A1: Introduction**

Tutorial A2: Creating a smart contract

Estimated time: `10 minutes`

A ***smart contract*** contains the business logic, expressed in code, used to generate a transaction response by a single organization. The response is digitally signed by that organization and contains the the state change of a set of business objects. In Hyperledger Fabric, a smart contract uses a ***state*** database, which contains the current value of all business objects in the ledger. Each smart contract runs on a set of peers in eaach organization.

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:

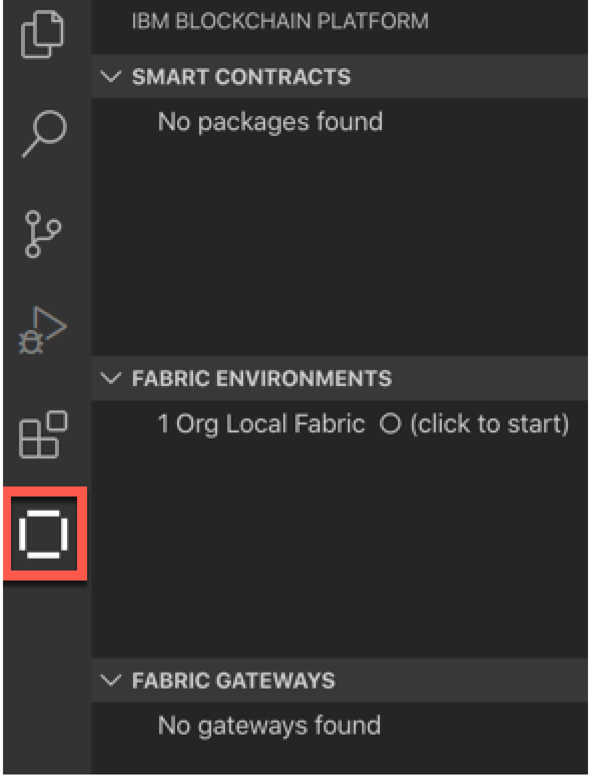
□ `A2.1`: Expand the first section below to get started.

▶ 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.

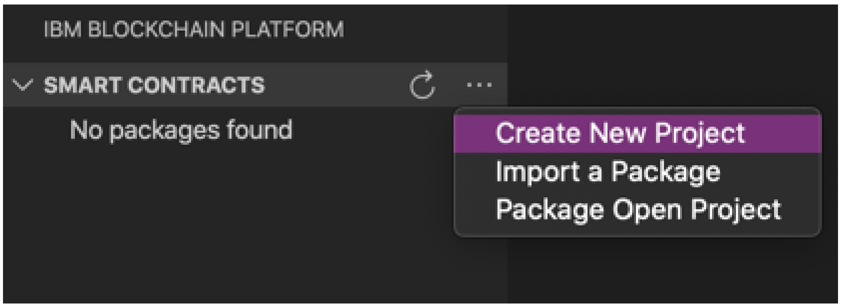
You can show the sidebar by clicking on the IBM Blockchain Platform icon in the VS Code activity bar. However, note that the icon is a toggle: if you click on it while the sidebar is already shown, the sidebar will be hidden.

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

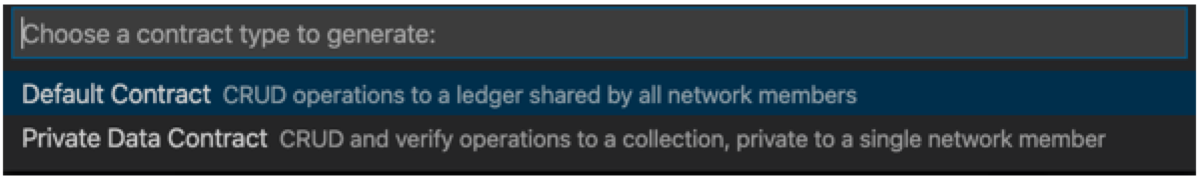


We will now create a smart contract project that will house the files we need for our smart contract. IBM Blockchain Platform will create for us a skeleton smart contract that we can customize later.

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

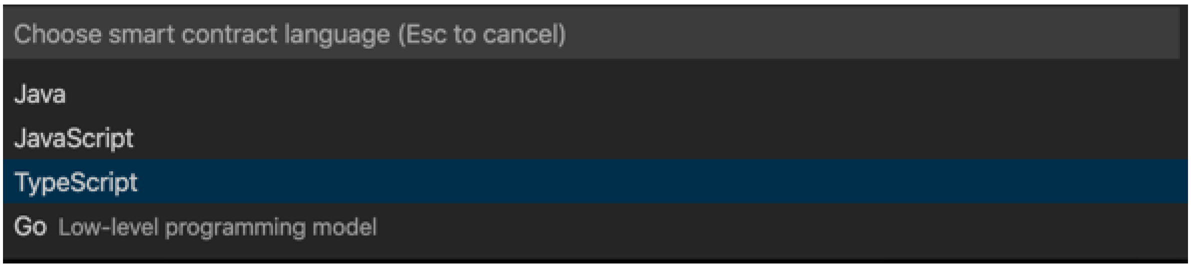


□ A2.4: Press Enter to accept the Default Contract type.


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.

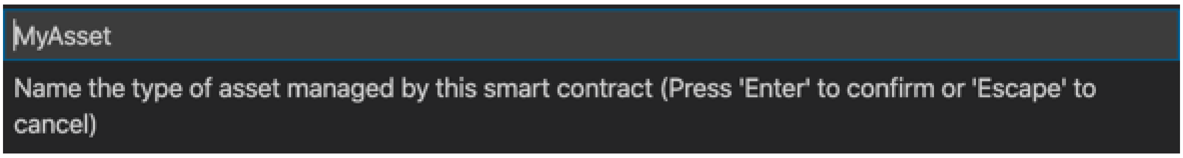
□ A2.5: Click 'Typescript'.


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. Artwork), 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.

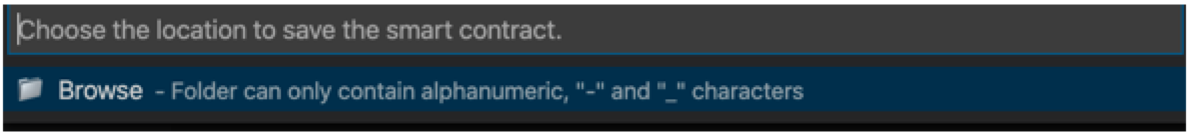
□ A2.6: Press Enter to accept the default asset type ("MyAsset").



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.

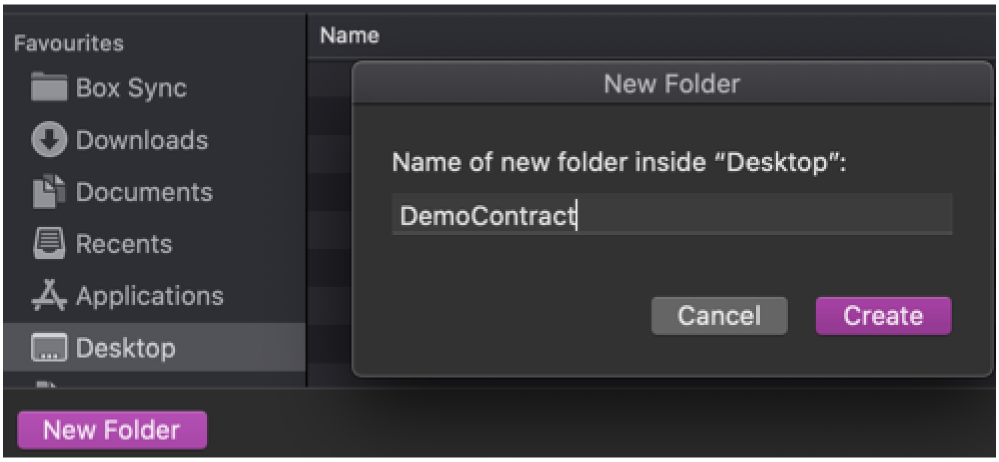


Choose the location to save the smart contract.

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

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

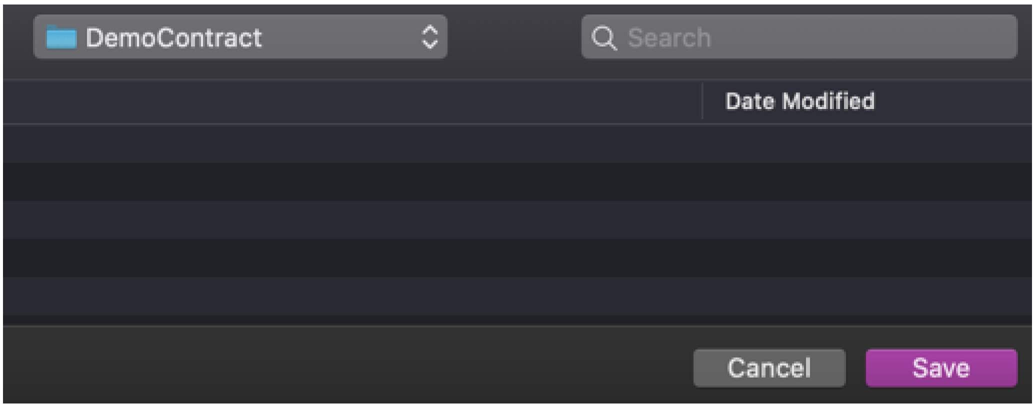
□ A2.8: Click "New folder" to create a new folder to store the smart contract project, and name it "DemoContract".



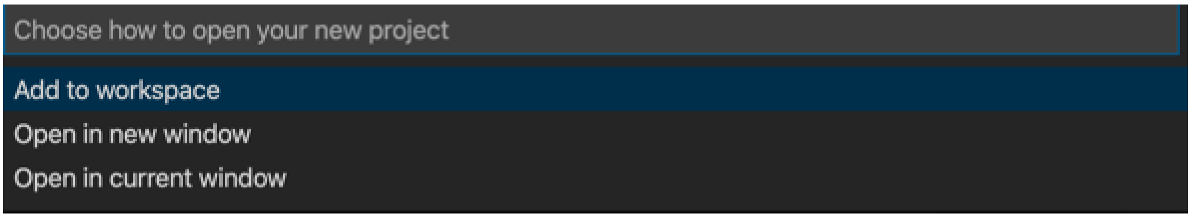
Name of new folder inside "Desktop":

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.



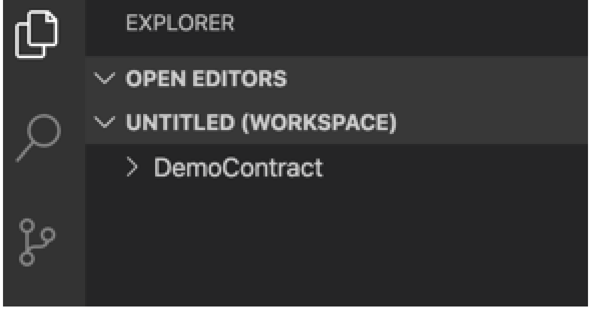
Choose how to open your new project

Add to workspace

Open in new window

Open in current window

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.

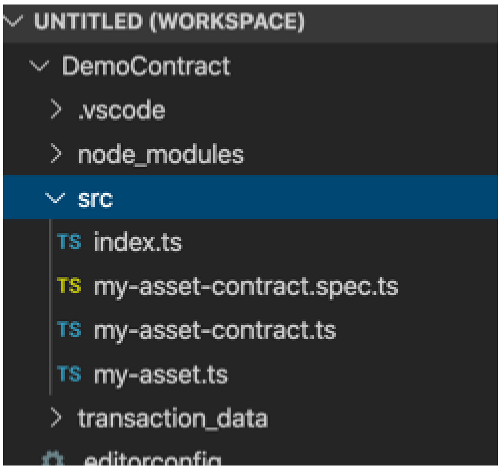


□ A2.11: Expand the next section of the tutorial to continue.

▶ Learn about the smart contract

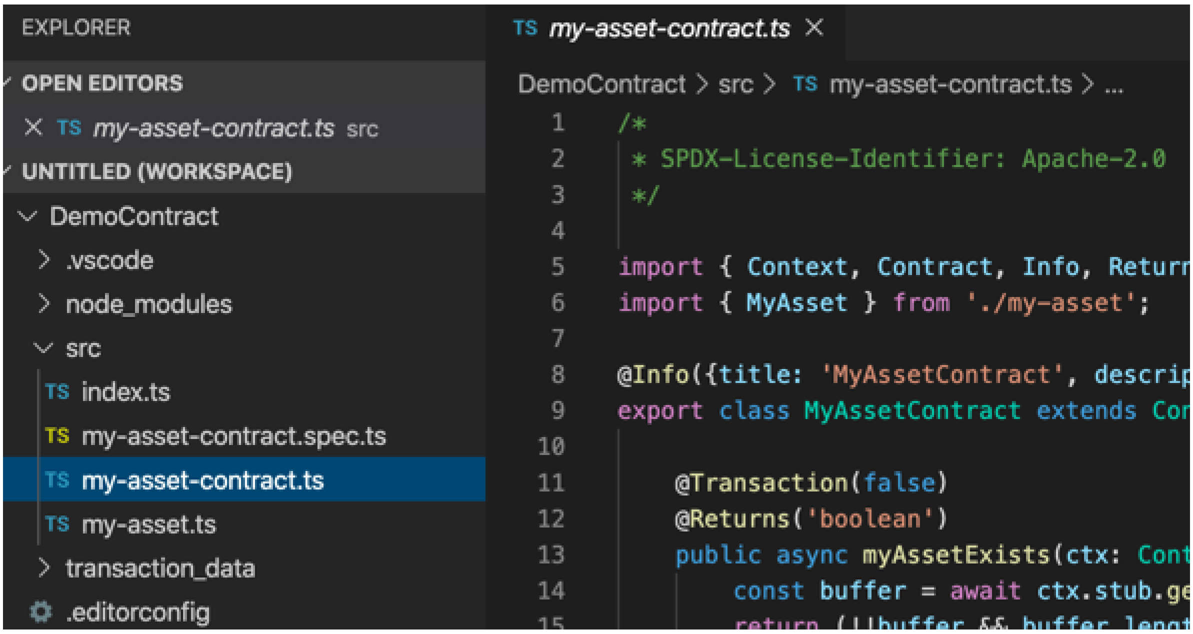
We will now look at the files that have been created to see what they do.

□ 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, Retur
import { MyAsset } from './my-asset';

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

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

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 contains the implementation for each transaction type generated by the smart contract.

Transactions

Look at the first method signature in the MyAsset contract:

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

Any method that is prefixed with the ***@Transaction()*** decorator indicates that this method can be called to generate a transaction response for this smart contract.

We can see that myAssetExists() is prefixed with a slightly more descriptive decorator - @Transaction(***false***) - indicating that it only reads from the ledger. We'll see an example of a @Transaction(***true***) method that also writes to the ledger later in this tutorial.

The remainder of the myAssetExists() signature mostly shows the required inputs provided by an application when it wishes to generate a transaction response. The Context object is a special parameter; it is used by the smart contract to maintain user data between transaction invocations, and we'll see later how it simplifies the process of writing a smart contract.

Imagine we had a ledger of cars: myAssetExists('CAR001') would return true or false depending on whether CAR001 was in the ledger.

Working with the world state

Each transaction method uses the world state to read the current value of a set of business objects and generate the corresponding new values for those objects. The Context object (ctx) gives you direct access to the world state:

```
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 only reads the value of business object, it does not change it. That's why the myAssetExists() method was decorated @Transaction(false).

Now take a look at the second method:

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

This ends with the line:

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

The putState method changes the current value of a business object with the key myAssetId to the value buffer. That's why the createMyAsset method was decorated with @transaction(true).

The transaction response for is generated automatically by Hyperledger Fabric when the smart contract execution completes. It comprises the states that have been read and those that are to be written.

It's important not to confuse these transaction responses with the return value from the method; they are not the same thing.

Smart contract determinism

While it may look like it, the effect of ***putState*** is not immediate. The business object identified by myAssetId will only be updated when every organization in the network agrees with the generated transaction response. This requires the consensus process to complete across the network.

As developers we generally don't need to worry about consensus. However, we do need to ensure that our smart contract is ***deterministic***; that is, it must always generate the same transaction response for a given set of transaction inputs. That's because our smart contract will be run by multiple organizations, each of whom must generate the ***same*** transaction response. If not, the resulting generated transaction will be captured in the ledger as ***invalid***, and the world state will not be updated. The transaction will not have had an effect.

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 our first smart contract.

We saw how a smart contract contains different methods, each of which can generate a transaction response for a given set of inputs. Transactions are generated using the ***getState*** and ***putState*** methods, which provide keyed access to get and set the current value of a business object in the ledger.

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**