Let us consider a hypothetical scenario where there are two organizations for a Mango Business. The first one is the producer’s organization and the second one is the seller’s organization. Producer’s organization consists of a set of authorized farmers who produce Mangoes every year. They produce and attach a unique code to Mangoes, and make an entry onto the Hyperledger Fabric Network.

The Seller organization sells the Mangoes using the unique code and records it onto the Network. Don’t worry about the use case at this point. You will learn more about this use case later in the Chaincode section. For now, let us understand that we have two organizations in the network design and we need to bootstrap the network for these organizations.

……………………………….

**Installing the Prerequisites**

In this section, you will learn how to install the Visual Studio Code IDE, the basic dependencies and finally the Fabric VS Code Extension for our development.

**Installing Visual studio code**

Download [Link](https://code.visualstudio.com/download) for Visual Studio Code

**Installing the Dependencies**

Please note that the below tutorials and commands are for **Ubuntu** machines.

You need to install the following dependencies on your machine before starting the development.

* **Curl**

Install curl on your ubuntu machine using the command

 sudo apt install curl -y

* **Docker** – Version 19.03 or above

Download the docker script using the command and provide execute permission.

curl -fsSL https://get.docker.com -o get-docker.sh

chmod +x get-docker.sh

Execute the script using the command

./get-docker.sh

Remove the docker script

rm get-docker.sh

Enable your user to properly use the docker commands without using sudo for every command. Execute the following command

sudo usermod -aG docker $USER

Now, **reboot** your machine

Check the installed docker version using the command

docker version

* **Docker-compose** – version 1.2.5

Install docker-compose using apt

sudo apt install docker-compose

Check the docker-compose version using the command

docker-compose version

* **Build Essentials**

Install build-essentials using the apt

sudo apt install build-essential

* **Nodejs** – 16.x

Get the node install script using curl

curl -fsSL https://deb.nodesource.com/setup\_16.x | sudo -E bash -

Install Node.js using apt

sudo apt install -y nodejs

Check the nodeJS version

node -v

* **npm** – 8.x

npm will be installed while installing nodeJS.

Check the npm version

npm -v

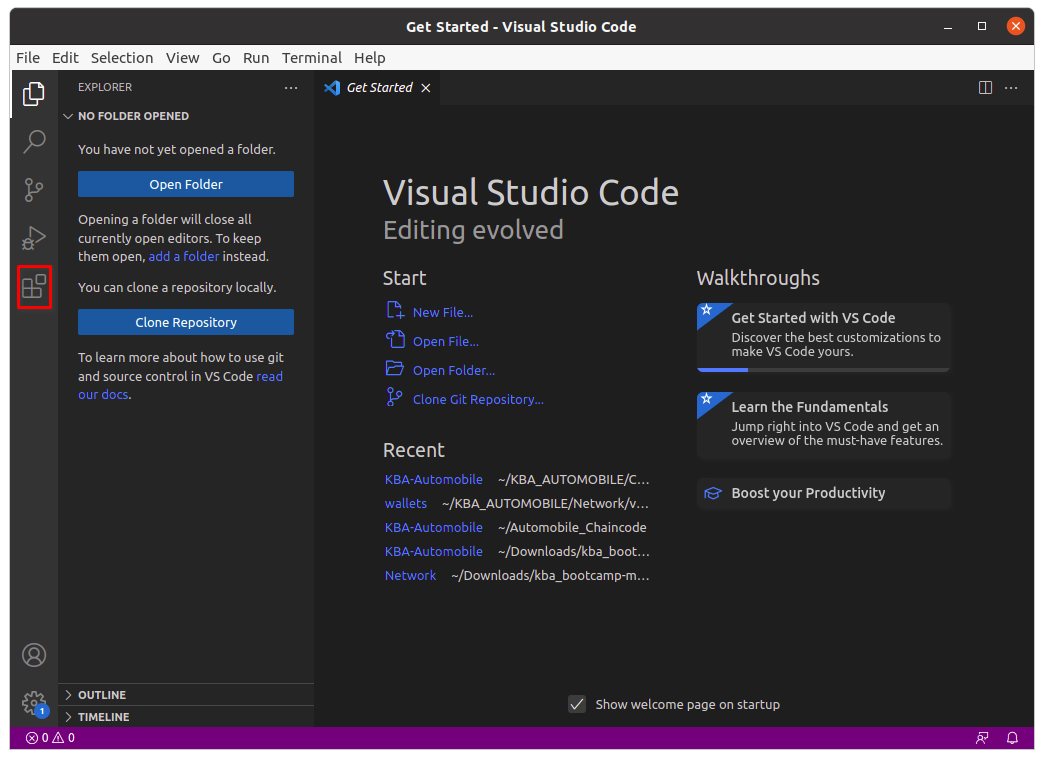
**IBM Blockchain Platform extension**

IBM blockchain platform extension in Visual Studio code helps with the seamless deployment and management of Hyperledger Fabric networks. This extension can be used to set up a local Fabric network, develop a new smart contract, package and deploy the newly created smart contract onto the local network and use it for development and testing purposes.

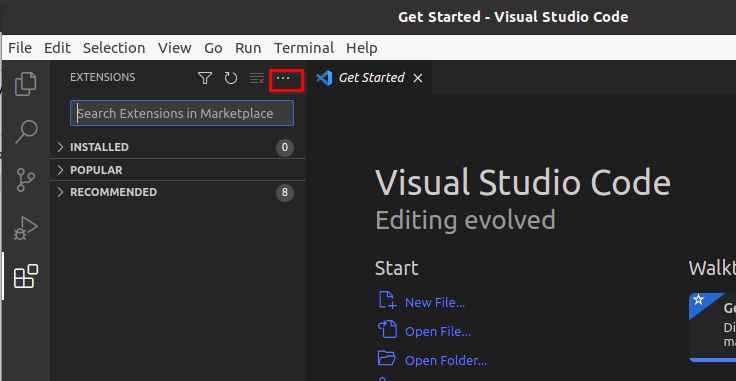
**Installing  IBM Blockchain Platform extension  in VS code**

Please download the IBM Blockchain Platform Extension for VS Code from [here](https://gitlab.com/CHF_KBA/kba_chf_ibmblockchainplatformextension_vscode/-/raw/main/ibm-blockchain-platform-2.0.8.vsix?inline=false). Follow the steps given below:

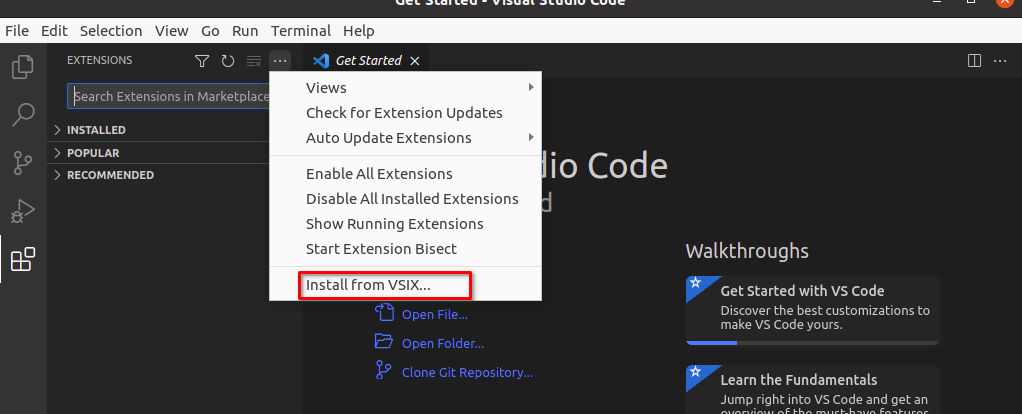
1. Open **VScode** and click on **Extensions** tab.



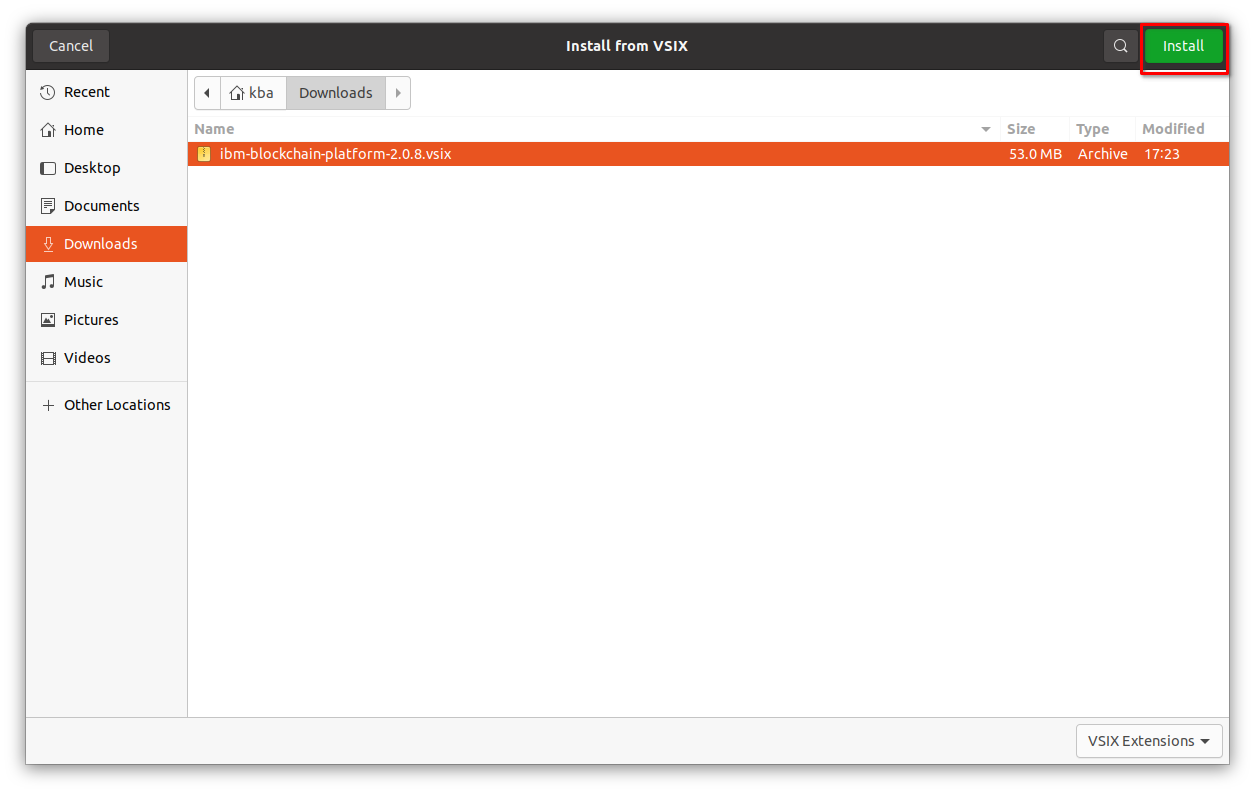
2. Click on **Views and More Actions**(three dots at the top).



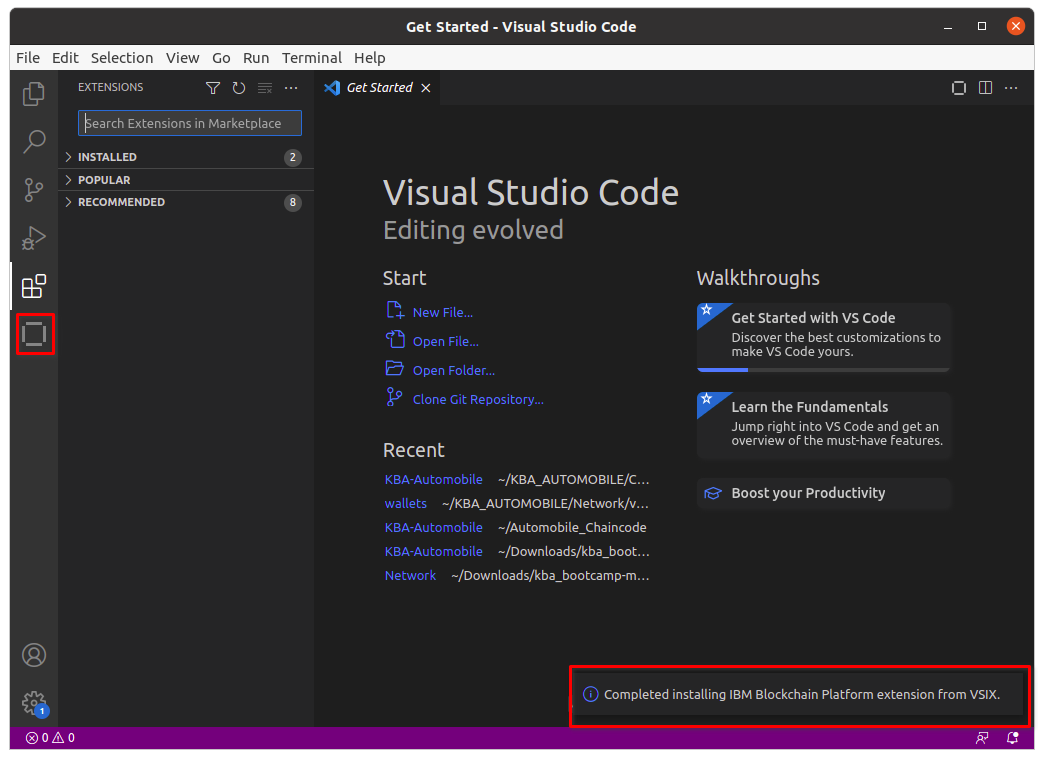
3. Select **Install from VSIX**.



4. Browse to the location that contains the downloaded file, **ibm-blockchain-platform-2.0.8.vsix**, select it and click **Install**.



5. After successful installation, you will get an alert and also the icon will be available in VScode.



………………………………..

**Prepare the configuration**

For bootstrapping the Hyperledger Fabric, we have different approaches available. We have chosen to use Microfab which makes it easy to understand and operate on.

Microfab, provided as part of the IBM Blockchain Platform, is a containerized Hyperledger Fabric runtime for use in development environments.

To explore more on Microfab [click here](https://github.com/IBM-Blockchain/microfab).

We are just one command away from bootstrapping the network. But, before we run that command we need to export the configuration as an environment variable.

Let us have a look at the configuration in our use case:

MICROFAB\_CONFIG='{

“port”: 8080,

“endorsing\_organizations”:[

 {

“name”: “ProducersOrg”

 },

 {

“name”: “SellersOrg”

 }

],

“channels”:[

 {

“name”: “mango-channel”,

“endorsing\_organizations”:[

“ProducersOrg”,

“SellersOrg”

 ]

 }

 ]

}’

***port:****Port on which Microfab container is mapped to run*

***endorsing\_organisations:****Organisations present in a business network.*

***channels:****Name of the channel on which the network is operated.*

Top of Form

COMPLETE

Bottom of Form

………………………………………………

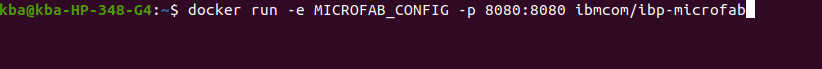
# Running the Network

After you are done with the configuration,  export the configuration like below:  
export MICROFAB\_CONFIG='{  
"port": 8080,  
"endorsing\_organizations":[  
{  
"name": "ProducersOrg"  
},  
{  
"name": "SellersOrg"  
}  
],  
"channels":[  
{  
"name": "mango-channel",  
"endorsing\_organizations":[  
"ProducersOrg",  
"SellersOrg"  
]  
}  
]  
}'

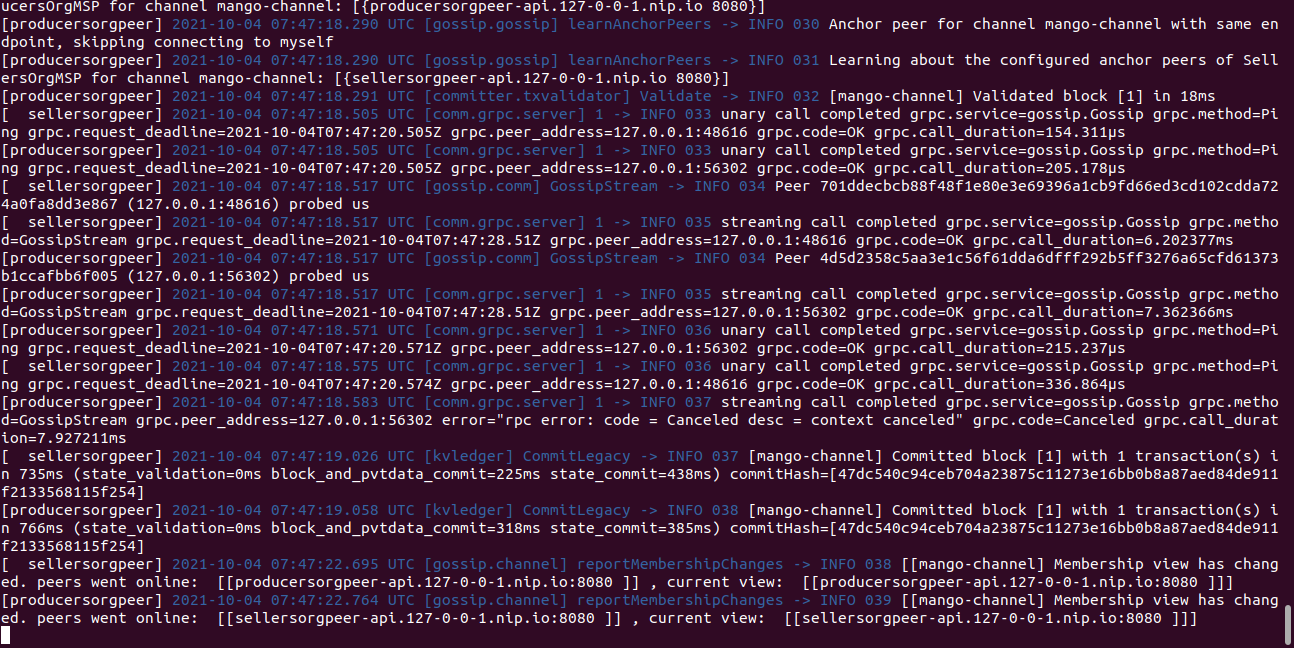


After you are done with exporting configuration, here comes the final command:

$ docker run -e MICROFAB\_CONFIG -p 8080:8080 ibmcom/ibp-microfab



Now, a runtime docker environment will be running in your terminal.



Hurray!! Your network is bootstrapped successfully and is ready to use. Please leave the terminal window. You can minimize it.

………………………………….

# Integrate with VSCode

Now, you have your network ready with your Hyperledger Fabric Network. But, if you want to interact with the Network, you need to have a client. For that, we will use the IBM Blockchain extension installed in the VSCode editor.

The below video shows detailed steps to integrate MicroFabric Network with IBM Blockchain extension.

# <https://youtu.be/6Hu4ARDXUSk>Teardown the Network

After you are done with the network operations, you also need to know how to teardown the MicroFab network.

The below video has detailed steps to teardown the network.

<https://youtu.be/Qi9CPRrrUaY>

…………………………………………….

Write a smart contract:

# Scaffold Project

Until now, you have learned about the architecture basics of Hyperledger Fabric, and this is the time to build your business logic and place it on the network that we launched.

We have also learned that the business logic is implemented as smart contracts and that is deployed on the network to perform the business actions as transactions.

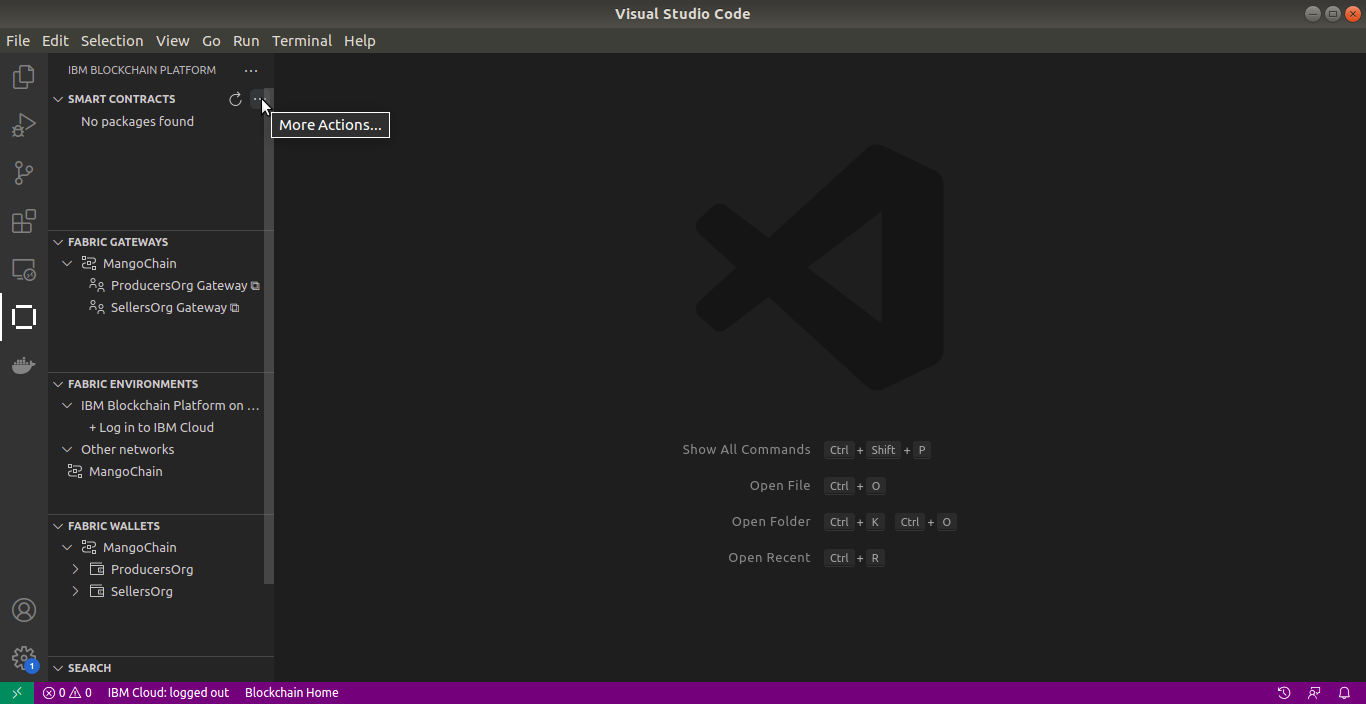
In this topic, we are going to learn how to write smart contracts using NodeJS from scratch. IBM Blockchain extension provides a way to create a boilerplate for smart contracts with bare minimum functionalities.

Please follow the below steps to generate the smart contract project and let’s start to write the code.

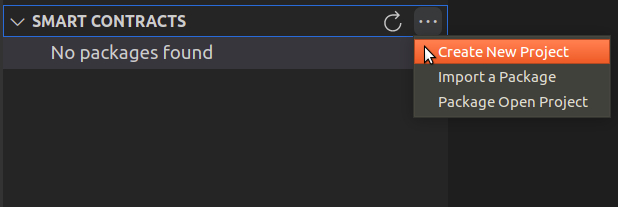
#### 1. Create New Smart contract Project

Let’s build a new smart contract project with a bare minimum code with the help of **IBM Blockchain extension**.

Go to **IBM Blockchain extension** and select the **more options** button (…) from the **Smart Contracts** window.

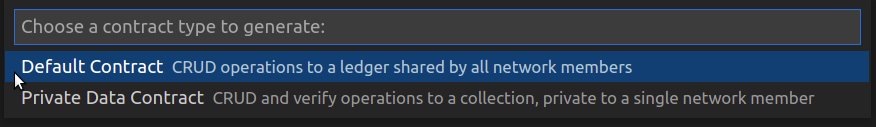


Choose the **“Create New Project”** menu for generating a new smart contract project.



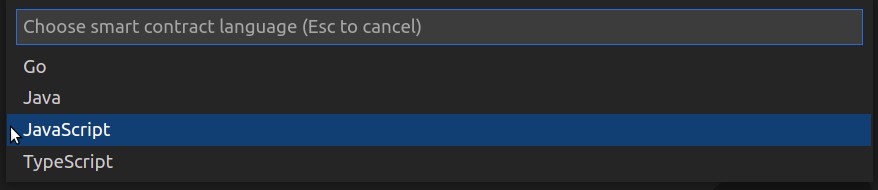
#### 2. Choose Contract Type

Now you can see a new prompt asking for choosing the contract type. Let’s pick the**“Default Contract”** and proceed.



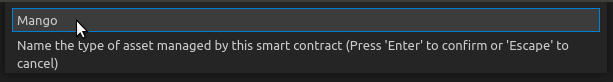
#### 3. Choose language

Next, you have to decide the language for the smart contract, you can develop smart contracts in four different languages, which are Go, Java, JavaScript, and Typescript. Since this course is designed for NodeJS smart contracts, please select **“JavaScript”** as the smart contract language.



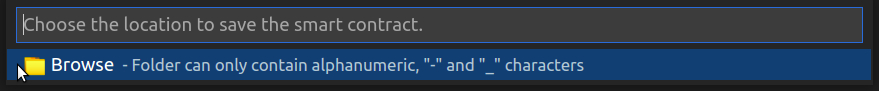
#### 4. Name the Asset

Each smart contract has its assets, or the business logic in a smart contract is built around an asset. So the next step is to name your asset. Since we are building a smart contract for the mango supply chain use case, please give the asset name **“Mango”** and press Enter.



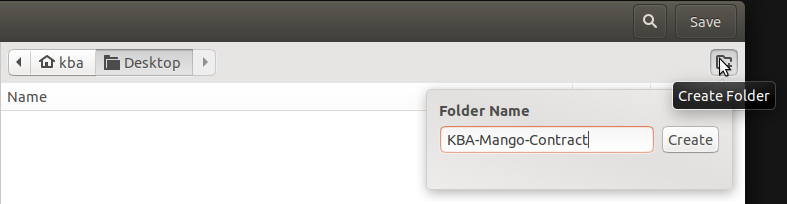
#### 5. Browse Location

Let’s select a place for keeping the Smart contract project. So please click the **“Browse”**.



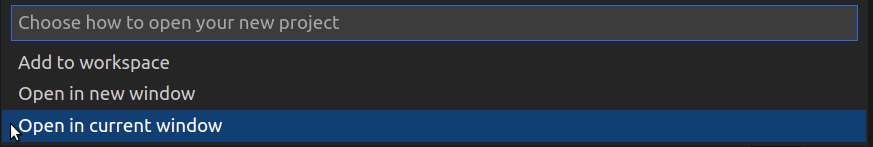
#### 6. Create Folder

Next, please go to your **Project Location** and create a **New folder** with the **Project Name**(KBA-Mango-Contract) and click the **SAVE** button.



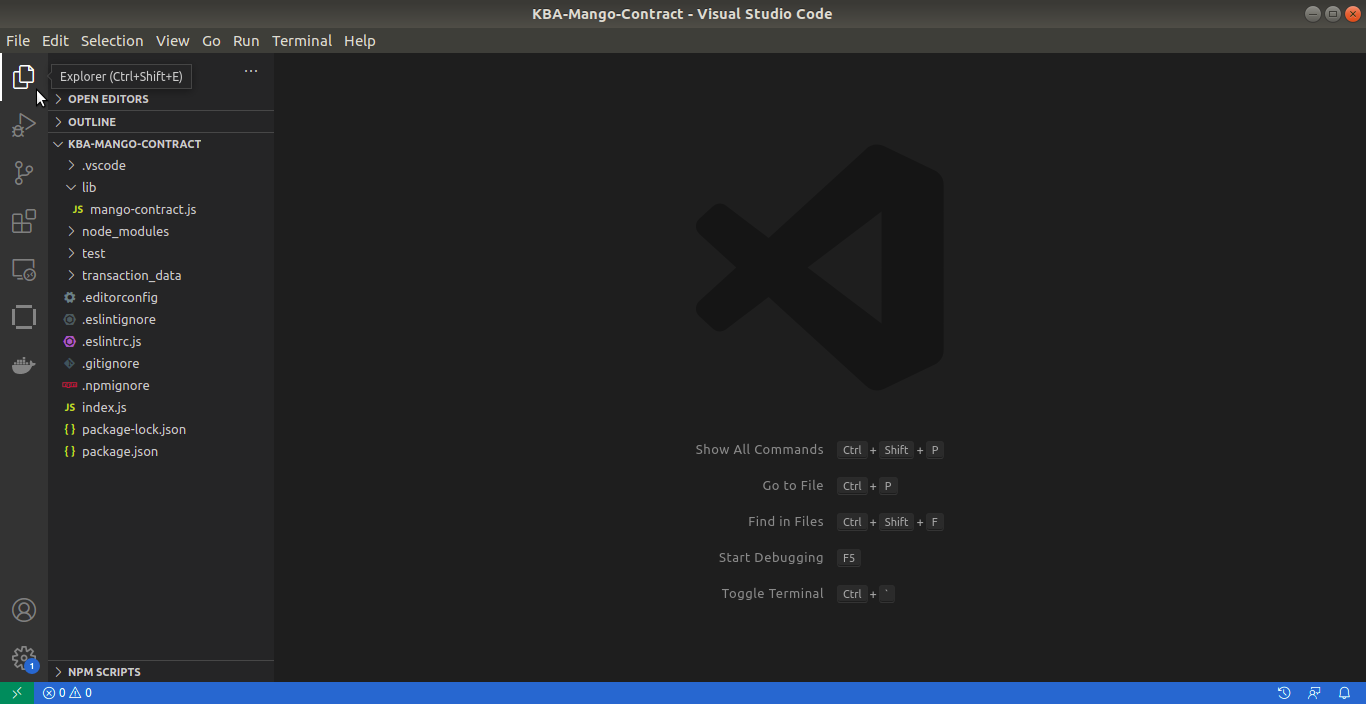
#### 7. Open in Current Window

Let’s open the newly created smart contract project in the current window in VS code.



#### Project folder

It takes a few seconds to generate everything for the new smart contract project in NodeJS. After a few seconds, you can see a notification message in the bottom-right corner of VS code saying that “Successfully created smart contract project”. And the VS code will be reloaded with the new project and files like the below diagram.



…………………….

**Project Structure**

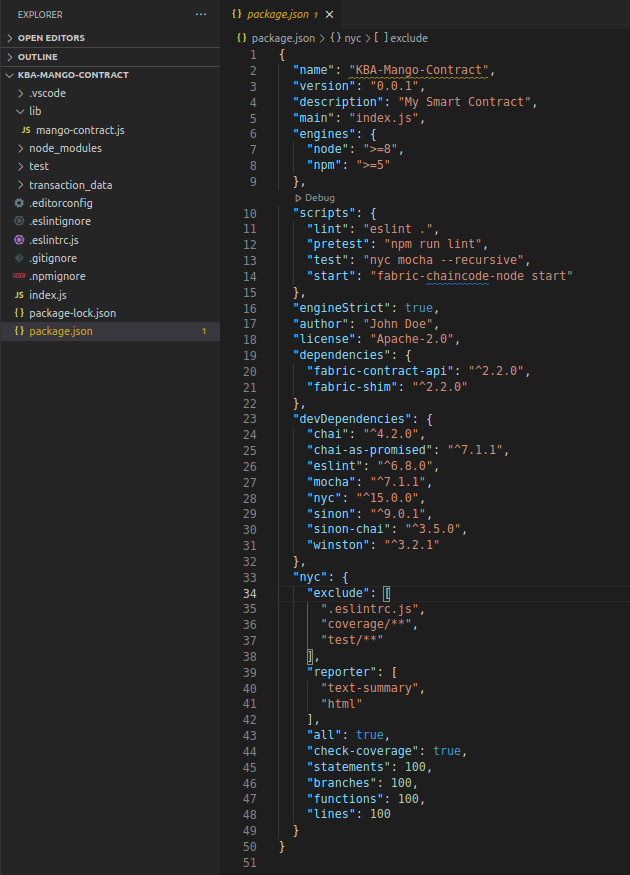
Now you have successfully generated a smart contract project in NodeJS with minimum functionalities. Let’s explore the very important files in this project.

**package.json**

This is the basic configuration file in a NodeJS project. It contains the basic configurations and dependencies.

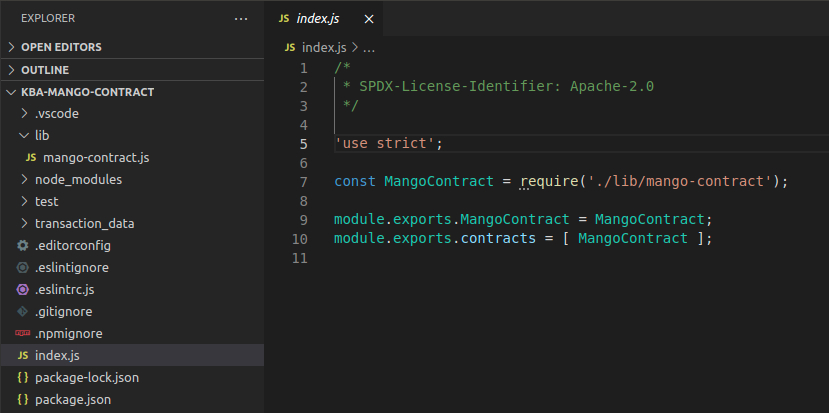
Let’s check the most important configurations,

1. **name**: This is the name of the smart contract,
2. **version**: This is the version of the smart contract. NodeJS uses the name and version of the smart contract to generate a unique namespace for the chaincode. And the developer should bump(increment) the version of the smart contract after making any changes in the code right before deploying it.
3. **description**: A Place for giving a description for the smart contract project.
4. **main**: This is the entry point to the NodeJS project. In this project index.js is the entry point.



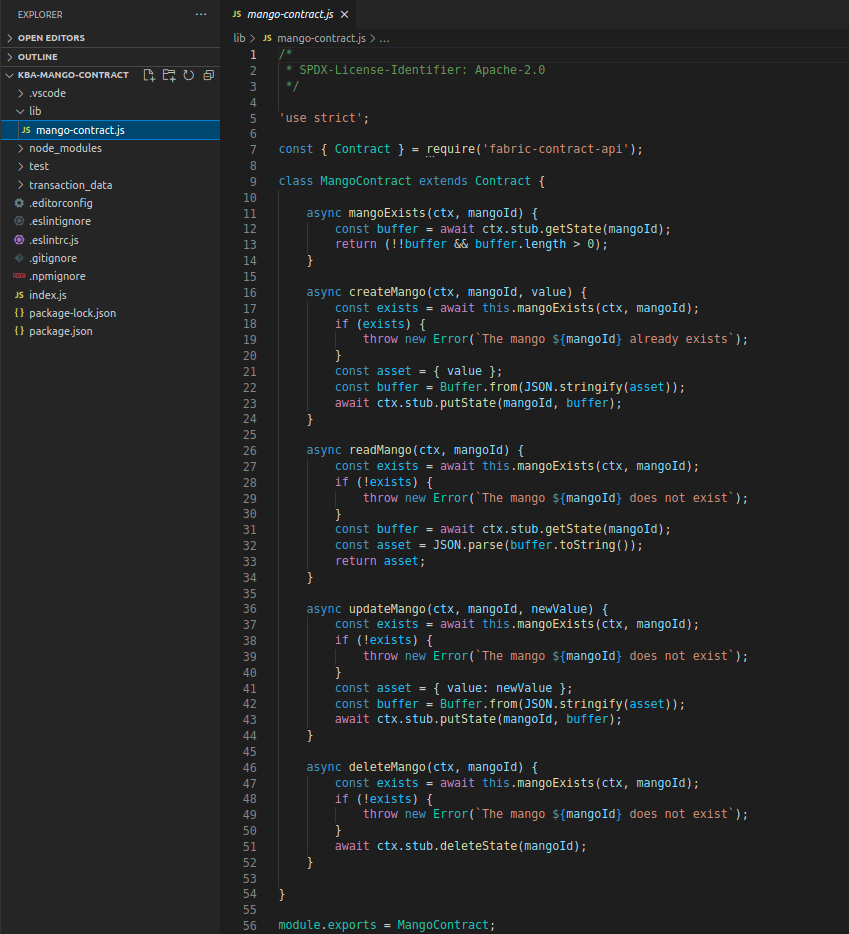
**index.js**

We already discussed that index.js is the main file. So let’s check the code, Here you can see at line number 7, a file called **mango-contract**is loaded from the **lib** folder(that is the actual smart contract file), and storing it in a constant variable called **MangoContract**. Subsequently, you can see that the MangoContract is exported as a list to the **contracts** module.



**mango-contract.js**

Let’s explore the smart contract, please go to the **lib** folder and open the file mango-contract. So this is the place that we need to add our business logic. We will discuss more on the contract file in the coming chapters.



……………………………………..

**Discussing the default smart contract**

Now we have a simple smart contract in our hands. We are going to apply our business logic(code for driving the mango’s transactions).

So before starting the real code, let’s try to understand this auto-generated basic contract. As a chaincode developer, it will definitely give a clear idea about how smart contracts look like.

To explore the smart contract code, please go to the **lib** folder and open the **mango-contract.js** file.

**Overview**

Let’s skim through the available contract. The smart contract code is started with a user directive ‘use strict’ to force the code execution in strict mode.

Please refer to this link for more information <https://www.w3schools.com/js/js_strict.asp>

**Loading Contract Class**

The first step to writing a new smart contract is to load the appropriate NodeJS SDK library. Here we are using the fabric-contract-api.

See the below code, which is used for loading the Contract class from fabric-contract-api.

const { Contract } = require(‘fabric-contract-api’);

**Create a user-defined Contract Class**

The next step is to create a new contract class by inheriting the Contract class. In this way, you can implement your business logic inside the inherited class by using functionalities of the Contract class.

class MangoContract extends Contract {}

**Export**

Finally, in the end, let’s do a module export with the newly created class. Then only it’s available in the index.js file for exporting the contract.

module.exports = MangoContract;

**Contract Class**

The business logic will place under the Contract Class. We already discussed that the code for the business logic is added inside the user-defined contract class and it is inherited from the Contract class from fabric-contract-api.

Here we can see that a contract class, **MangoContract** is automatically generated with a few functions and code.

We can explore it one by one. And also you can see that the contact class is exported as a module in the last line.

**Default Functionalities**

You can see that the smart contract is generated with a bare minimum code. So let’s list the functionalities available in these smart contracts.

1. **mangoExists**
2. **createMango**
3. **readMango**
4. **updateMango**
5. **deleteMango**

In this smart contract, you can see that all the functions are interacting with the ledger, so that’s why these functions used transaction context (ctx) as a parameter.

**Transaction Context**

Generally, a context holds all the relevant information that a programmer needs to complete a task. Similarly, in a Transaction Context, which holds necessary information for performing transactions. By default, a context contains two elements, stub, and clientIdentity.

**Stub** enables accessibility to a wide range of methods used for processing transactions.

**clientIdentity** is used for getting the user-related identity pieces of information.

If you are implementing a smart contract function by inheriting the Contract class, then you should pass context as the first parameter to your function to use the stub methods for processing your transactions.

**Important Stub Methods**

Let’s list the stub methods used in this smart contract for performing transactions or related activities.

1. **getState(assetId)**
2. **putState(assetId, buffer)**
3. **deletState(assetId)**

All these above methods are provided by the stub interface.

Let’s get to their details.

**getState**

This method is used for reading information about the current state of an asset from the ledger. This method fetches the data from the state database for an asset Id.

**putState**

putState is used to create a new asset in the ledger by adding an entry to the world state. It is also used for updating an existing asset. The putState accepts two parameters as input. While one is the key(asset Id), the other is the value in buffer format.

The putState method will write something to the actual blockchain in the ledger too in the form of transactions.

**deleteState**

deleteState is the method used for removing an existing asset from the world state. Please note that the deleteState will not remove anything from the blockchain because it is immutable.

**Mango Contract**

Now we are going to travel through all the functions deeply and will try to explore its working. Before doing it, please see all these functions are using asynchronous programming.

Why do these functions use asynchronous programming?

The execution of the code needs to get a confirmation from the Fabric network before moving to the other parts. So, execution needs to wait until getting transaction confirmation. That is the use of **async** and **await** keywords in smart contracts.

**mangoExists**

This function is used to check whether an asset exists or not. The parameters are ctx and mangoId. Here is used the stub method getState to get the current state of the asset from the ledger. And this function will return either true or false based on the existence of the mango asset.

**!!buffer** is always true, but the combination of **!!buffer** and **buffer.length>0** will give the actual information about the existence of the mango asset.

async mangoExists(ctx, mangoId) {

const buffer = await ctx.stub.getState(mangoId);

return !!buffer && buffer.length > 0;

}

**createMango**

This function is for creating a new mango asset. So the parameters are ctx, mangoId, and value. mangoId is the asset id, which is used for referencing the asset in the state database. value, is the information about the mango asset, that we need to store in the blockchain.

First of all, the function is checking the existence of the mango asset by calling the mangoExists function. If the mango exists,Â it throws an error denoting the asset already exists. Otherwise, continue the execution.

The next step is to model your asset with the value you got as the parameter. Here it’s just storing the value as a JSON object. Sometimes there may have multiple properties for the asset. So you need to carefully model your asset here.

The data format, which is used to write or read from the ledger should be in the format of a buffer object. So we need to convert the JSON to Buffer by stringifying the asset.

Finally, call the putState method to create the new mango asset and pass the asset Id and buffer objects as parameters.

async createMango(ctx, mangoId, batchNumber, producer, quantity, price) {

const exists = await this.mangoExists(ctx, mangoId);

if (exists) {

throw new Error(`The mango ${mangoId} already exists`);

}

const asset = {value};

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

**readMango**

This function is for reading the current state of a mango asset. So the parameters are ctx, mangoId. mangoId is the asset Id, which is used for referencing the asset in the state database.

First of all, the function is checking the existence of the mango asset by calling the mangoExists function. If it does not exist there throw an error, saying that the asset does not exist. Otherwise, continue the execution.

The next step is to call the getState method and pass mangoId as the parameter. So you can get the result in a constant variable called buffer. And the result will be in a buffer format.

Finally, return the result as JSON by parsing the buffer to JSON.

async readMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const buffer = await ctx.stub.getState(mangoId);

const asset = JSON.parse(buffer.toString());

return asset;

}

**updateMango**

This function is almost the same as createMango function. The only difference is assigning a new value to the asset. And write the value to the ledger, by calling putState method.

async updateMango(ctx, mnewValue) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const asset = { value: newValue };

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

**deleteMango**

This function is used for removing an existing asset from the world state. So the parameters are ctx and mangoId as asset Id. Here also, it checks the existence of the asset first, and if it exists remove the asset by calling the deleteState method.

async deleteMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

await ctx.stub.deleteState(mangoId);

}

……………………………………………

**Adding More Functionalities**

**Coding begins here!!**

Before we start the actual coding, let’s discuss the implementation methodology that we’re going to practise. Here we are not removing anything from the code, but altering the code with our business logic. Identifying the asset and transactions prior to writing the business logic is very important here.

Let’s identify and model the **asset** first.

In the Mango Chain use case, we are doing transactions of mangoes. So we can name our asset as mango asset. A mango asset can have different properties like ID, BatchNumber, Producer, OwnedBy, Quantity, Price and so on.

Let’s identify the **transactions.**Here we can have all the basic CRUD transactions. So we can use the existing functions in the smart contract and we need to add one more function to change the asset owner name. We can use this function for selling the mangoes to another party(either customer or seller).

So the functions are,

1. **mangoExists**
2. **createMango**
3. **readMango**
4. **updateMango**
5. **deleteMango**
6. **sellMangoes**

**Mango Exists**

Firstly, we need to define a function that checks whether an asset Id exists or not. Let’s name the function as mangoExists and pass the asset Id(mangoId) as the function parameter.

The next step is to read the asset information from the ledger by using the stub method getState. Pass the mangoId as a parameter. This function will return either true or false values, based on the existence of the asset.

async mangoExists(ctx, mangoId) {

const buffer = await ctx.stub.getState(mangoId);

return !!buffer && buffer.length > 0;

}

**Create Mango**

This function is used for creating new mango assets. Here we are going to code the asset model that we discussed. So here we need to pass the asset properties as parameters like mangoId, batchNumber, producer, quantity, and price.

Let’s implement the code by defining the function name as **createMango** and passing the parameters as **mangoId**, **batchNumber**, **producer**, **quantity**, and **price**.

Firstly, call the prior implemented **mangoExists** function to check the newly created asset exists or not. If it’s already created then throw an error message saying it already exists.

Otherwise, define the asset as a JSON object.

ie,

{

ID: mangoId,

BatchNumber: batchNumber,

Producer: producer,

OwnedBy: producer,

Quantity: quantity,

Price: price,

};

Let’s convert the asset to a buffer object after converting it to a JSON string. And execute the **stub.putState** method by passing **mangoId** and buffer as parameters to create a new asset.

async createMango(ctx, mangoId, batchNumber, producer, quantity, price) {

const exists = await this.mangoExists(ctx, mangoId);

if (exists) {

throw new Error(`The mango ${mangoId} already exists`);

}

const asset = {

ID: mangoId,

BatchNumber: batchNumber,

Producer: producer,

OwnedBy: producer,

Quantity: quantity,

Price: price,

};

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

**Read Mango**

**readMango** receives the **mangoId** as parameter. This function is used for reading the information(current state) about an asset from the ledger. Here you can see that the data is retrieved to a constant variable called a **buffer**, the data will be in buffer format, with the help of the **getState** method. And in the next line, the buffer object is converted to a JSON object for making it human-readable.

async readMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const buffer = await ctx.stub.getState(mangoId);

const asset = JSON.parse(buffer.toString());

return asset;

}

**Update Mango**

This function is used for updating the properties of an already created asset. So let’s change the parameters of the function based on our business model. Here we pass **mangoId, batch number, producer, owner, quantity, and price** as parameters and set these values to the asset model, after checking the existence of the asset. Finally, convert the asset to buffer and update the ledger with the help of putState method.

Note: putState method will create a new asset if it is not already created. And putState is, also used for updating an existing asset.

async updateMango(ctx, mangoId, batchNumber, producer, owner, quantity, price) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const asset = {

BatchNumber: batchNumber,

Producer: producer,

OwnedBy: owner,

Quantity: quantity,

Price: price,

};

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

**Delete Mango**

This is the function used for removing an asset from the world state. The deleteState method is used for removing an existing asset from the world state.

async deleteMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

await ctx.stub.deleteState(mangoId);

}

**Sell Mangoes**

Let’s add the final function which is used for changing the state of an asset. While selling the mango, change the ownership of the asset to the buyer’s name. Let’s pass the buyer’s name as the owner’s name and update the current owner of the asset.

async sellMangos(ctx, mangoId, ownerName) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The apple ${mangoId} does not exist`);

}

const asset = { currentOwner: ownerName };

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

**Final Code**

Hurray! Now we have created a smart contract project with our own business logic. So the next step is to deploy it on the network and test the functionalities that we added.

/\*

\* SPDX-License-Identifier: Apache-2.0

\*/

‘use strict’;

const { Contract } = require(‘fabric-contract-api’);

class MangoContract extends Contract {

async mangoExists(ctx, mangoId) {

const buffer = await ctx.stub.getState(mangoId);

return !!buffer && buffer.length > 0;

}

async createMango(ctx, mangoId, batchNumber, producer, quantity, price) {

const exists = await this.mangoExists(ctx, mangoId);

if (exists) {

throw new Error(`The mango ${mangoId} already exists`);

}

const asset = {

ID: mangoId,

BatchNumber: batchNumber,

Producer: producer,

OwnedBy: producer,

Quantity: quantity,

Price: price,

};

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

async readMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const buffer = await ctx.stub.getState(mangoId);

const asset = JSON.parse(buffer.toString());

return asset;

}

async updateMango(ctx, mangoId, batchNumber, producer, owner, quantity, price) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

const asset = {

BatchNumber: batchNumber,

Producer: producer,

OwnedBy: owner,

Quantity: quantity,

Price: price,

};

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

async deleteMango(ctx, mangoId) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The mango ${mangoId} does not exist`);

}

await ctx.stub.deleteState(mangoId);

}

async sellMangos(ctx, mangoId, ownerName) {

const exists = await this.mangoExists(ctx, mangoId);

if (!exists) {

throw new Error(`The apple ${mangoId} does not exist`);

}

const asset = { currentOwner: ownerName };

const buffer = Buffer.from(JSON.stringify(asset));

await ctx.stub.putState(mangoId, buffer);

}

}

module.exports = MangoContract;

Top of Form

COMPLETE

Bottom of Form

……………………………………………………………………..

# Additional Resources

You can download the chaincode from this link: [KBA-Mango-Contract](https://gitlab.com/CHF_KBA/kba-mango-contract/-/archive/main/kba-mango-contract-main.zip)

Before packaging the chaincode, please install the node modules by running **“npm install”**command in the project directory.

……………………………………………………….

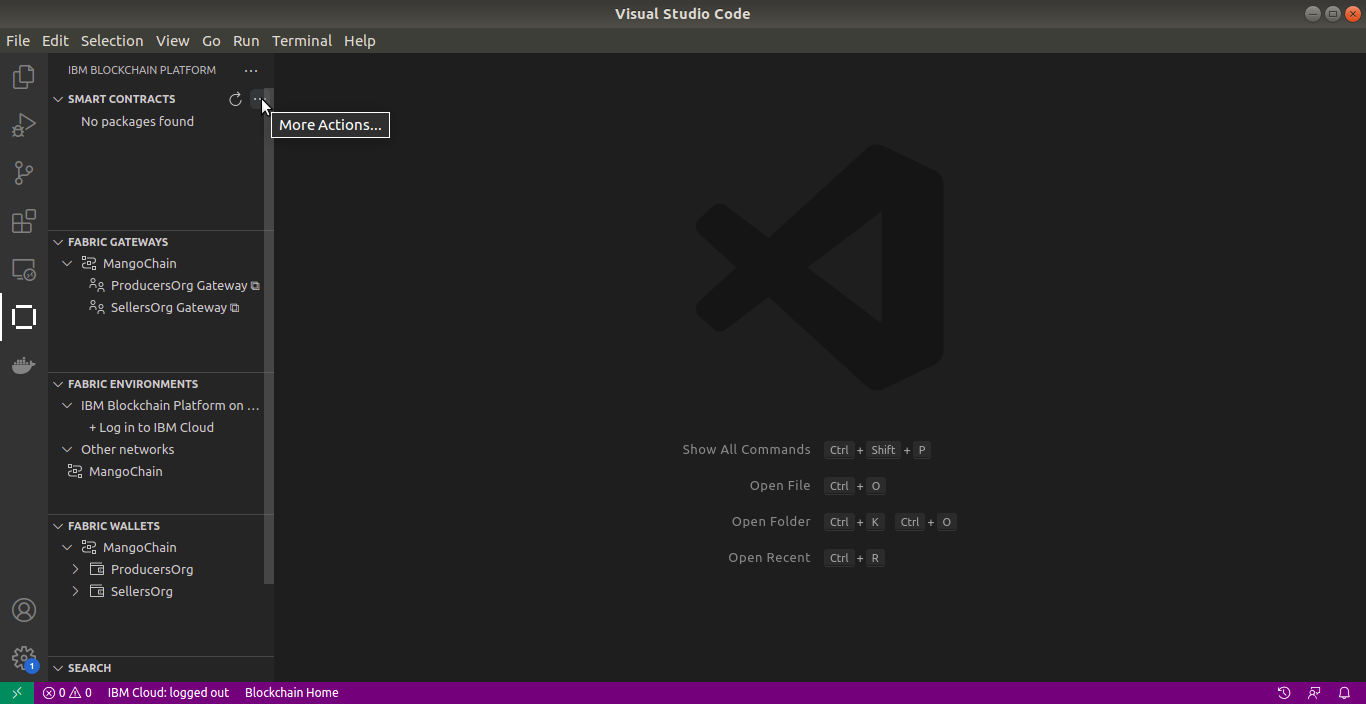
deploying chaincode:

# Packaging Smartcontract

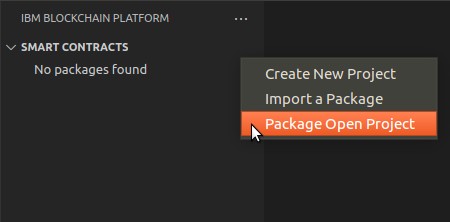
Packaging is the process of wrapping up everything in the smart contract project and creating a packaged chaincode. A package will be the actual chaincode, which can contain multiple smart contract files. The functions of a smart contract are only accessible after deploying the package on a network.

In Hyperledger Fabric 2.0, the chaincode package will be in the format of tar.gz. In the earliest version, it was CDS(Chaincode Deployment Spec).

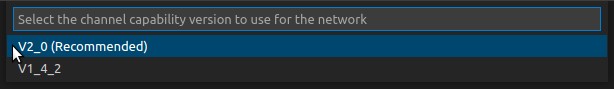
To create a new package, please go to the **More Actions** button in the **Smart contracts**window.



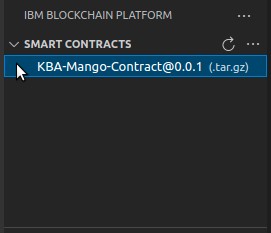
Select the **Package Open Project**.



Select the channel capability version as **V2\_0**. After a few seconds you can see a notification saying that the Smart Contract is successfully packaged.



Now you can see a new chaincode package under the **Smart Contracts** window with the name **KBA-Mango-Contract@0.0.1**



………………………………………………………..

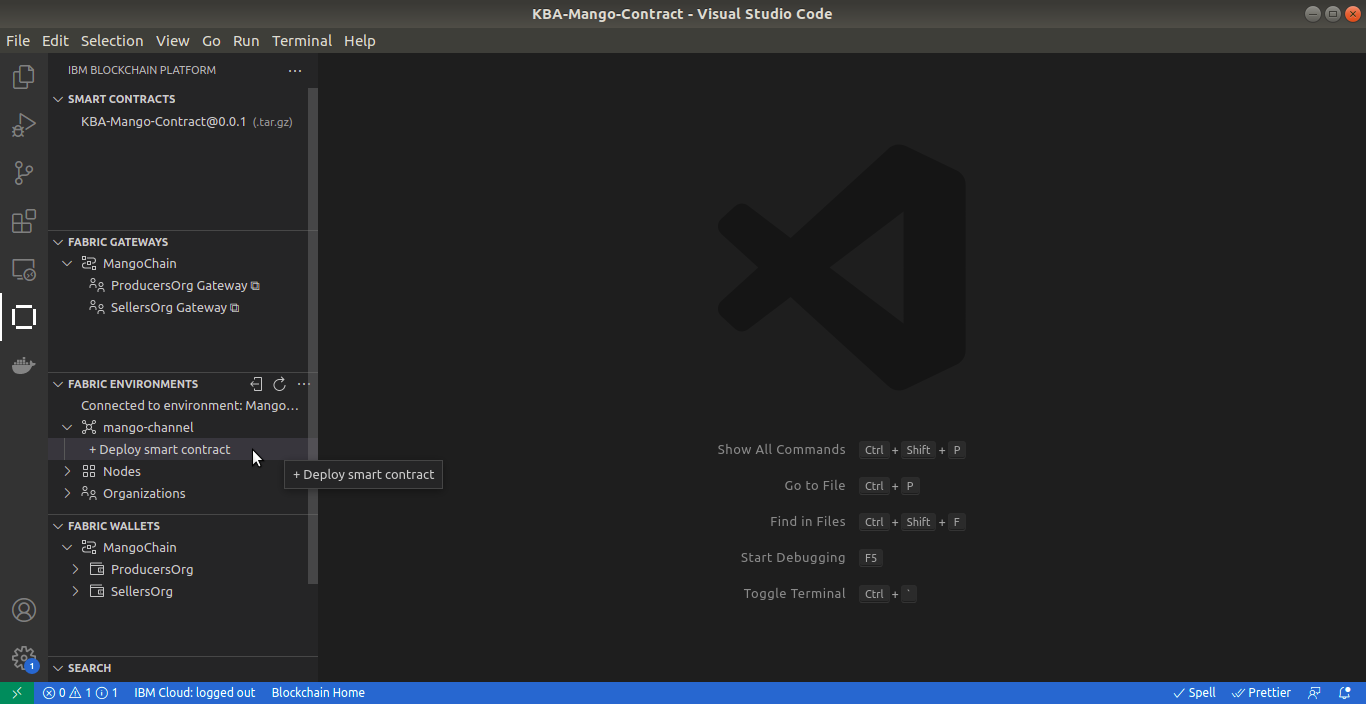
# Deploying Chaincode

In the previous chapters, we learned to implement the smart contract and created the chaincode package. So the next step is to deploy the chaincode on the network.

#### Deploy Smart Contract

To Deploy the smart contract, go to your network in the **Fabric Environments** window, and connect your network by clicking the network you have already created. Now you can see components in the network. Please explore the mango-channel and click the **Deploy smart contract** button.

(Note: If the network is not bootstrapped then refer to the previous sections – [Running the Network](https://learn.kba.ai/course/hyperledger-fabric-fundamentals/lessons/running-the-network/) and [Integrate with VSCode](https://learn.kba.ai/course/hyperledger-fabric-fundamentals/lessons/integrate-with-vscode/) )

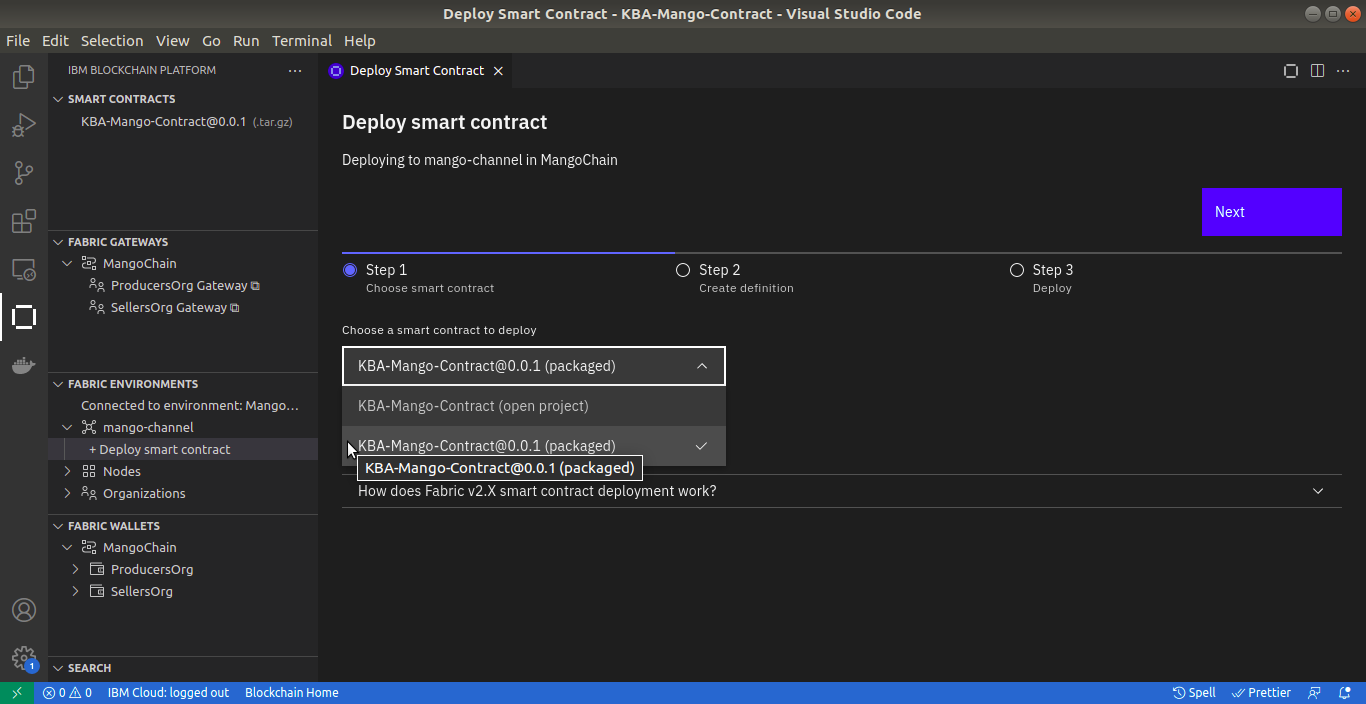


#### Choose Smart Contract

Now you can see a new tab that is opened the VScode with the name Deploy Smart Contract.

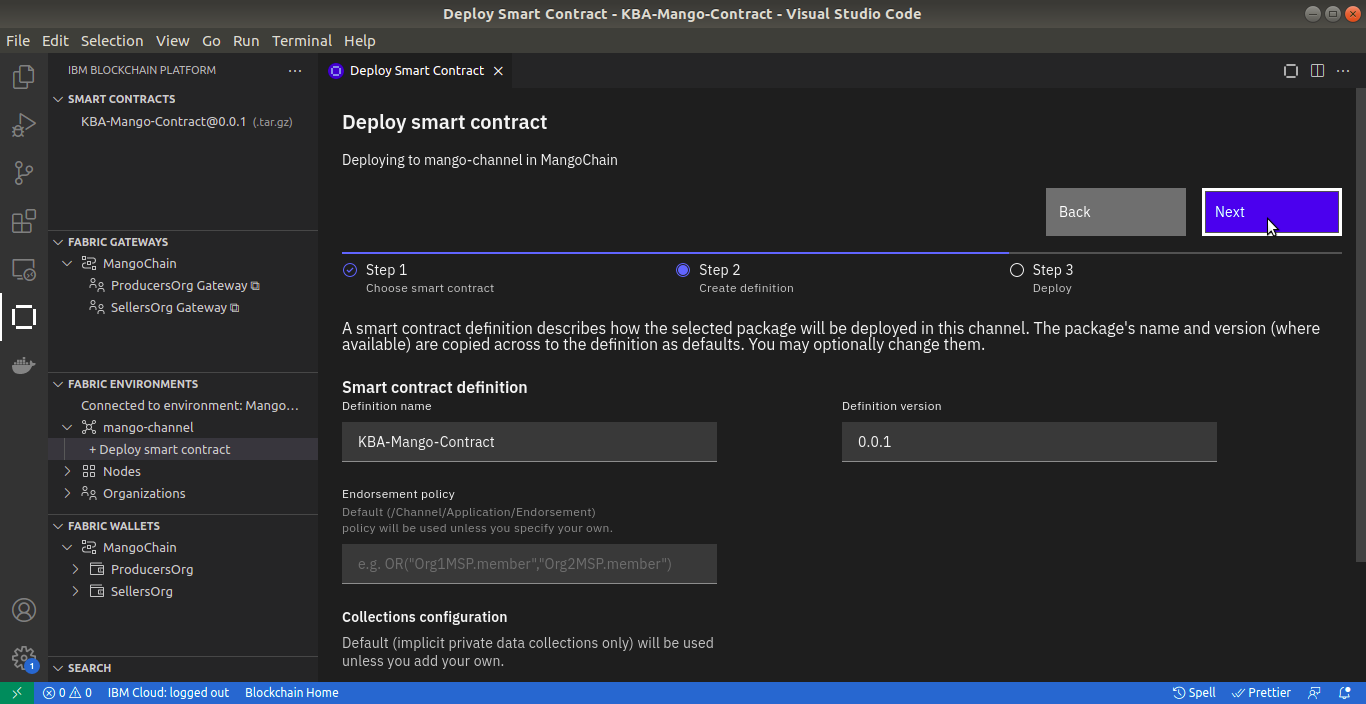
Let’s finish all three steps to deploy the smart contract on the network.

The first step is, choose the smart contract to deploy. Please select the KBA-Mango-Contract from the list. And click the **Next** button.



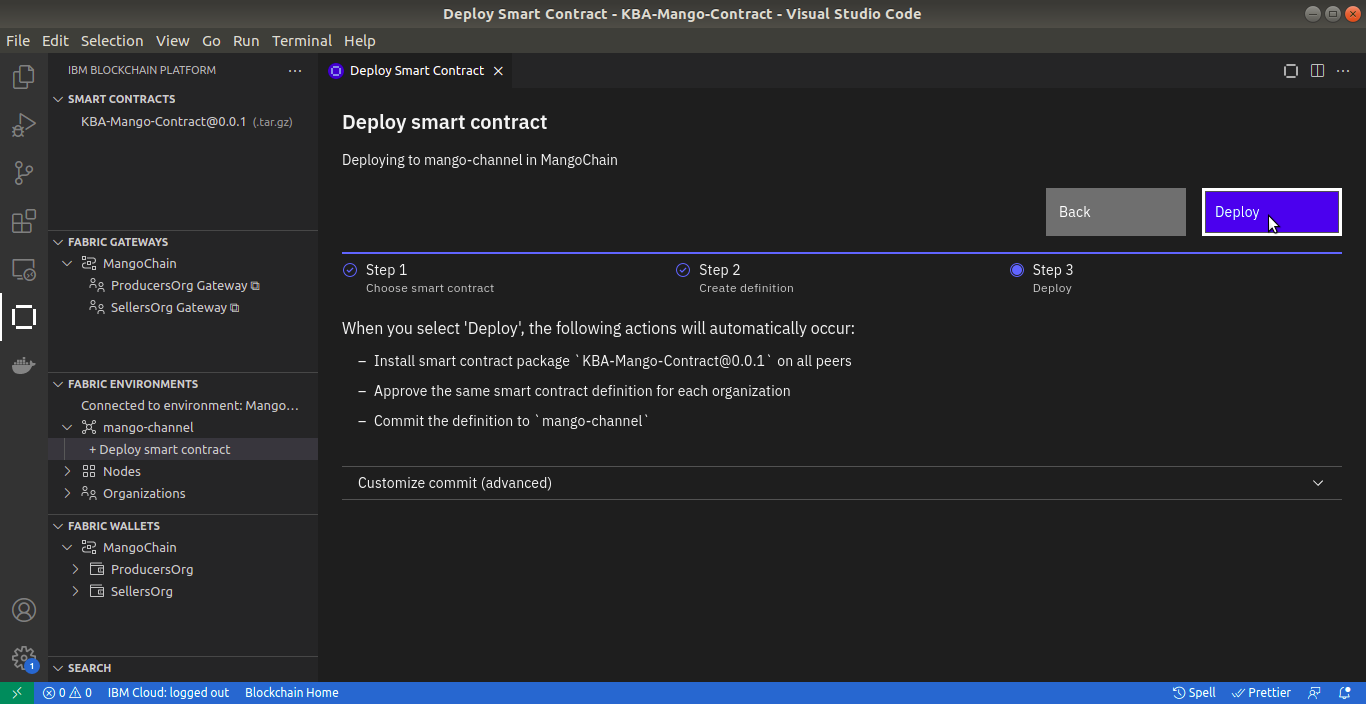
#### Create Smart Contract Definition

The second step is, give the smart contract definition and its version. You can see that it’s already populated there, if you want to change it you can do that. So remember one thing, all the peers must approve this smart contract definition. And click the **Next** button.

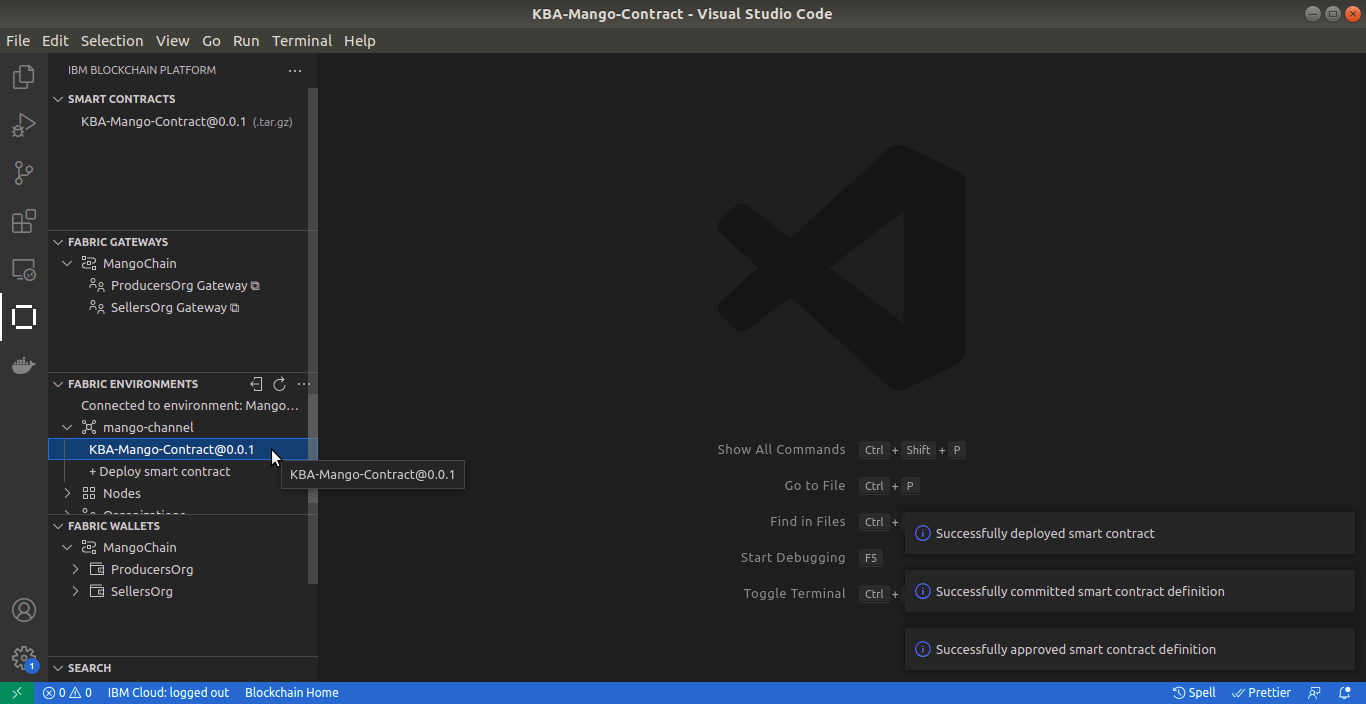


#### Deploy!!

Now click the **Deploy** button.



You can get the status of deployment in the Output window. And you can get notifications in the bottom right corner of VS code. Once everything finishes without errors, you can see the installed smart contract under the channel in the network window.



…………………………………………………………

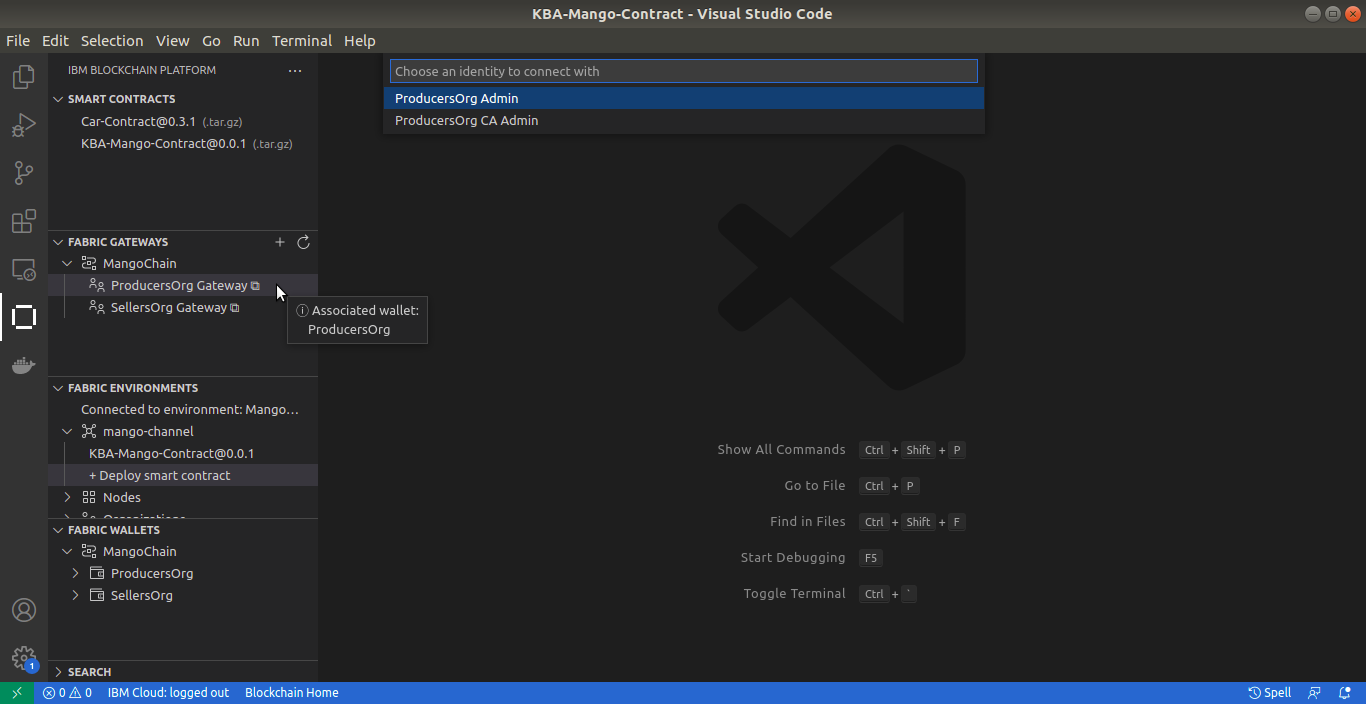
# Making Transactions

Remember the transaction flow diagrams from the previous chapter. You can see that all the transactions are done through a Client App. Don’t worry if Client App sounds unheard. IBM Blockchain Platform gives a feature to perform transactions without having a Client App. IBM Platform Extensions have implemented a feature known as Fabric Gateways that enables you to conduct transactions via appropriate identities.

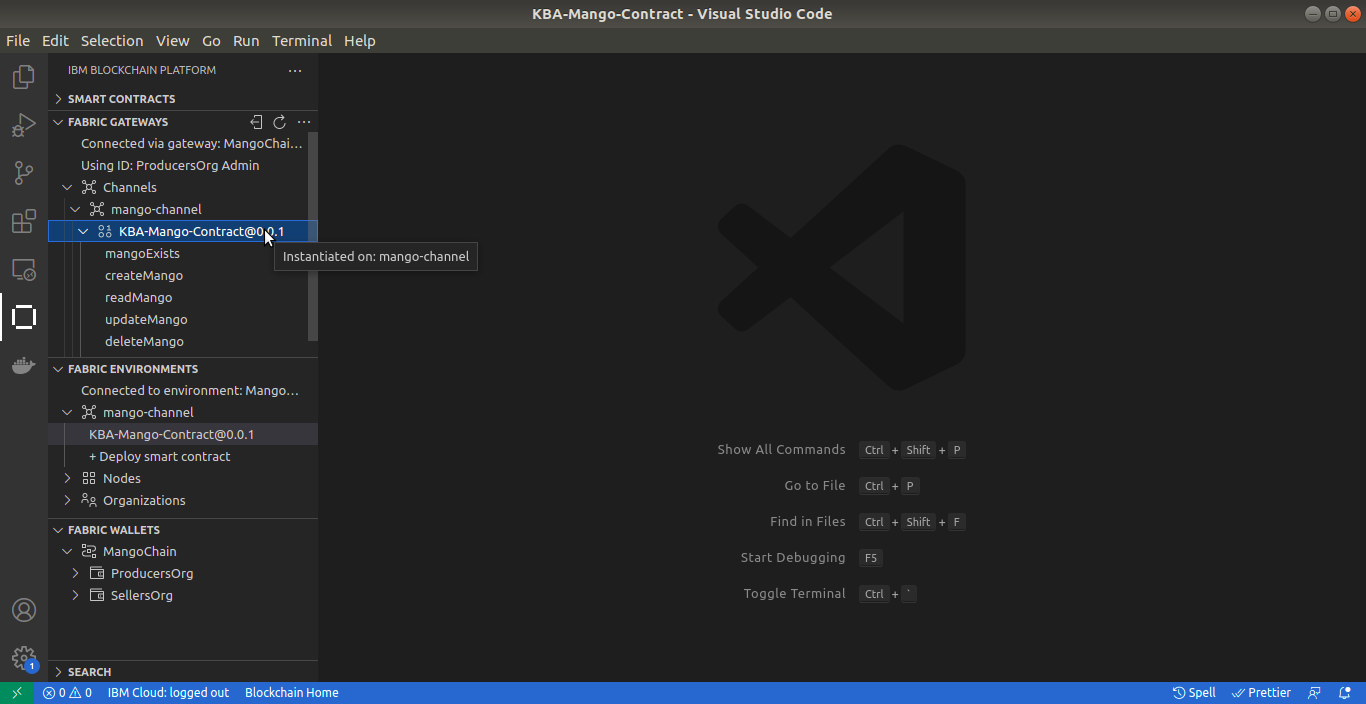
So now we are going to test the business logic that we implemented in the smart contract.

#### Fabric Gateway

To create transactions, please connect the network in the **Fabric Gateways** window. And click the Organisation which you want to do the transactions. For now, let’s select Producers Organisation and choose an identity from the subsequent list to connect.

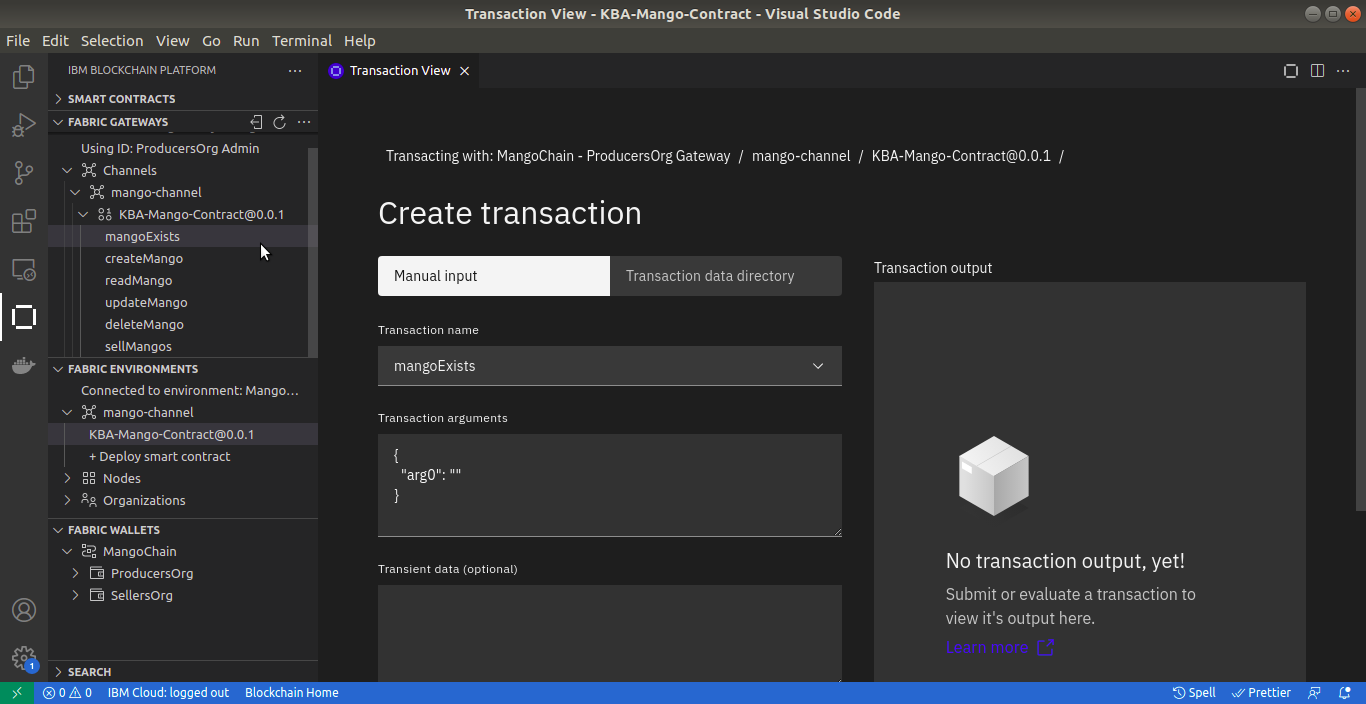


Now you can see the smart contract functions under, **Channels > mango-channel > KBA-Mango-Contract@0.0.1**

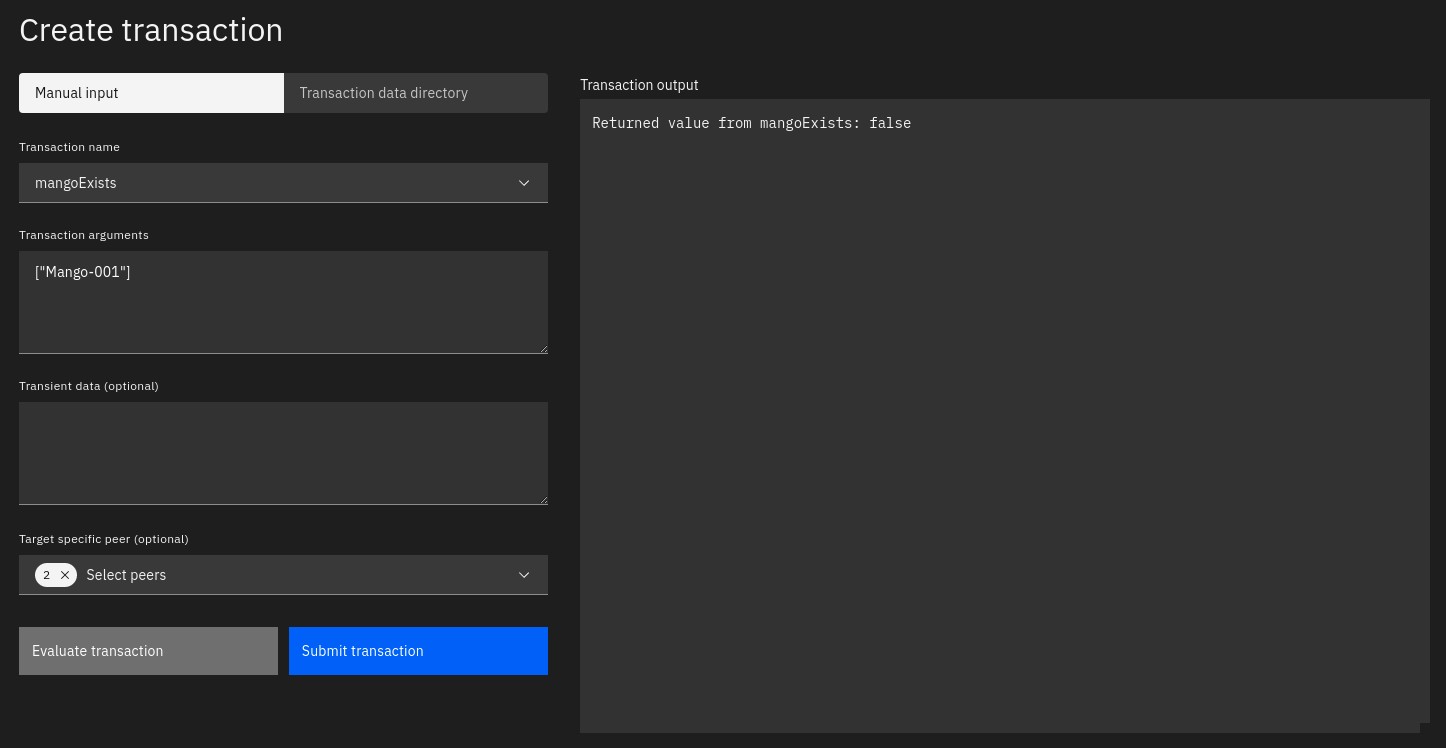


#### mangoExists

You can perform transactions by clicking on the smart contract functions.

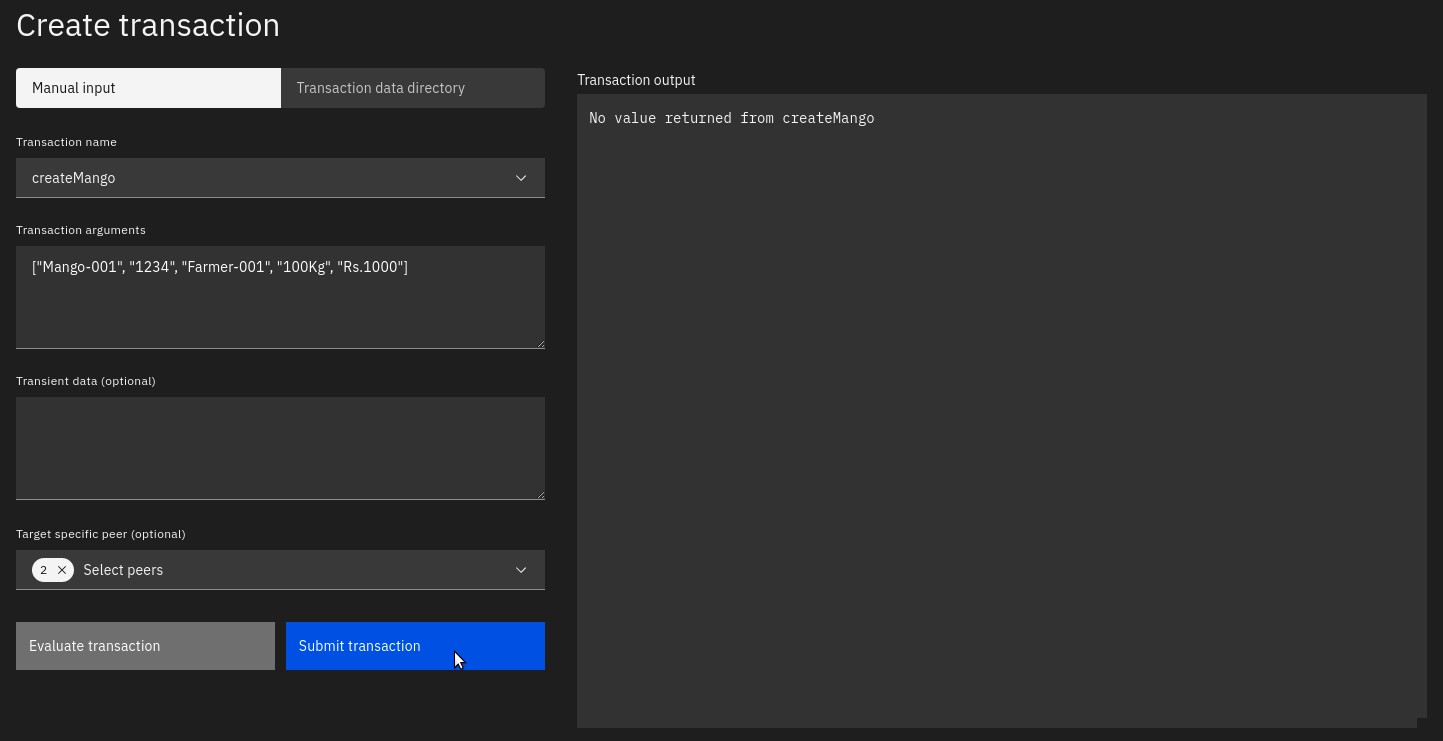


For example, let’s click on mangoExists, now you can see that a new tab is opened with the name Transaction View. Select the Manual Input tab under Create transaction, and give the transaction arguments as a list. Refer to the below image. And submit the transaction by clicking the **Submit transaction** button. So you can get a result from the Transaction output window. Now we can get a false value because there is no mango asset in the ledger as the supplied asset Id.



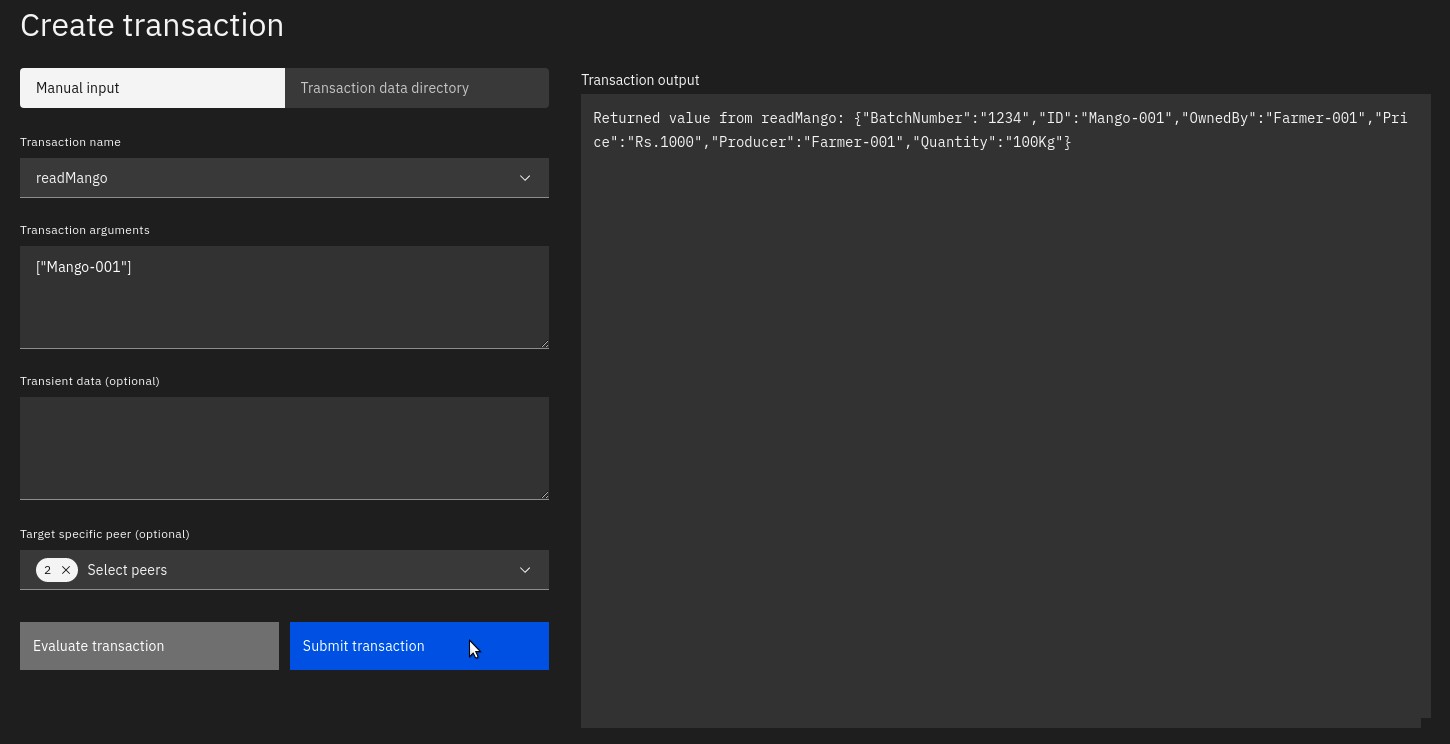
#### createMango

Let’s create a mango asset, by invoking createMango function. Select the createMango function from the transaction name, drop-down or click the createMango function from fabric gateways. And pass the values shown in the below image and submit the transaction. Now you can get a Successful message and you can see the result saying that no value returned from createMango.



#### readMango

Now we have successfully created an asset in the ledger. Next is to retrieve the information about the asset from the ledger. Choose readMango function from the transaction list and pass [“Mango-001”] as the argument. Now you can see the result showing the current state of the mango asset.



#### Teardown the Network

To teardown the network, refer to the previous section – [Teardown the Network](https://learn.kba.ai/course/hyperledger-fabric-fundamentals/lessons/teardown-the-network-2/).

………………………………………………