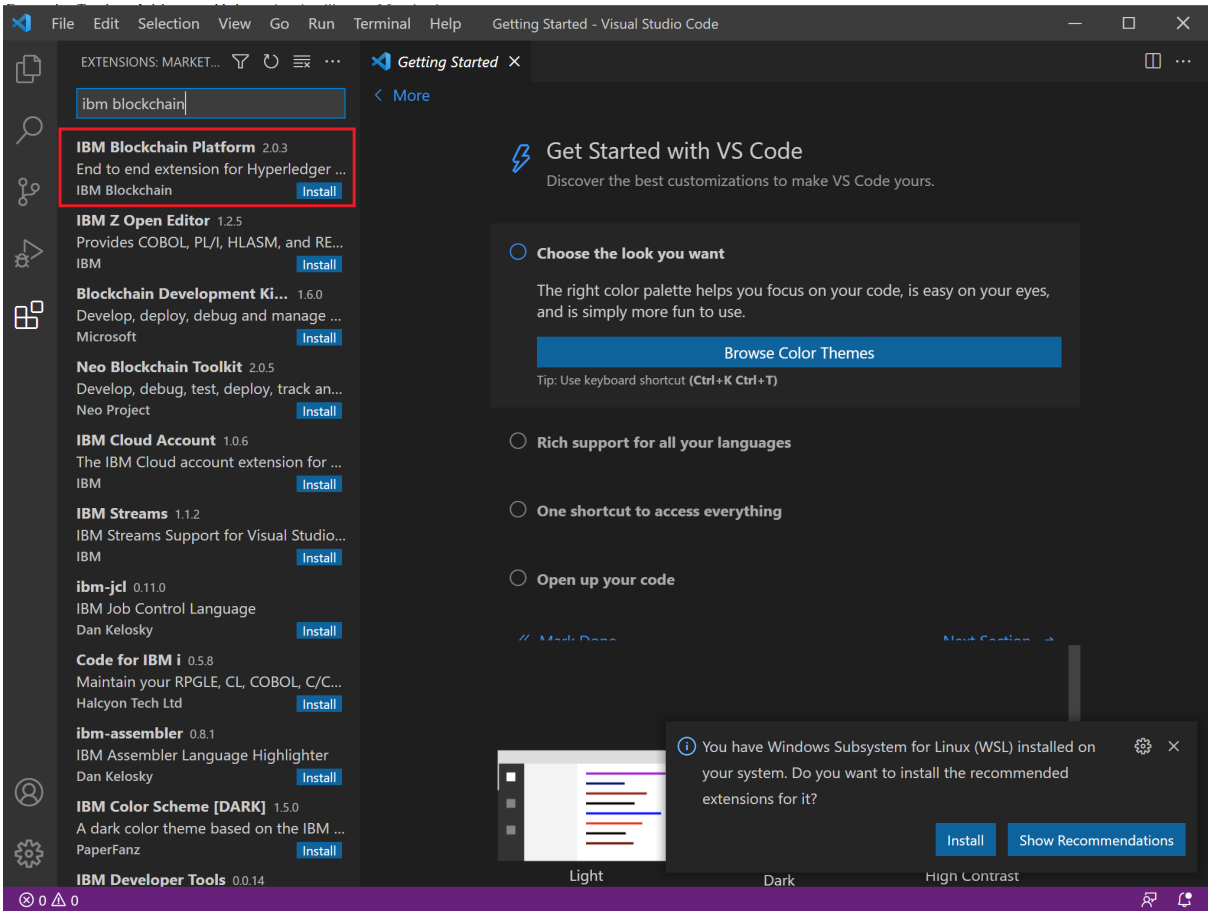
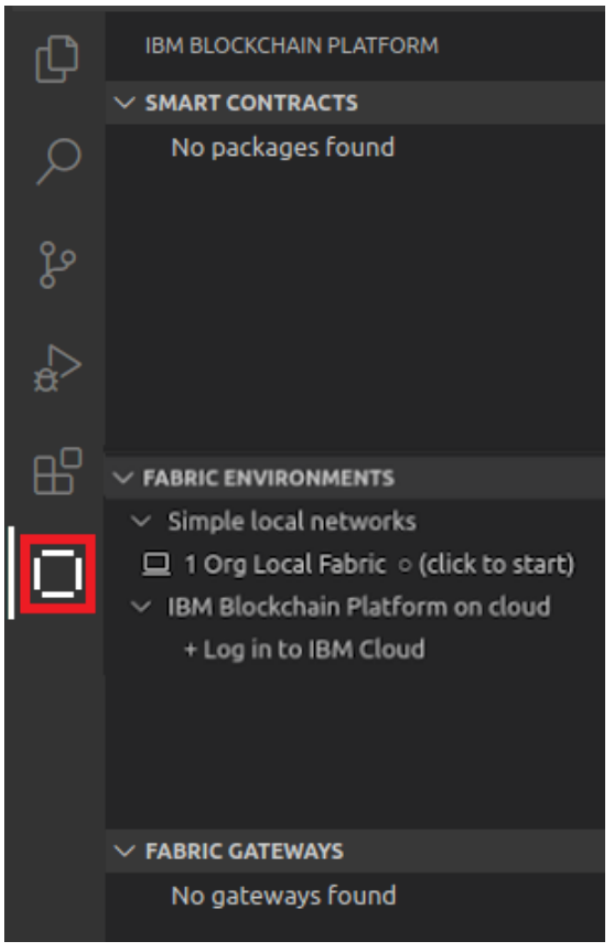
Install IBM Blockchain



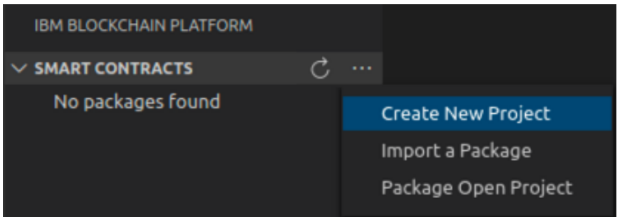
Create a smart contract project

We will use Hyperledger Fabric components and files in the IBM Blockchain Platform VS Code extension.

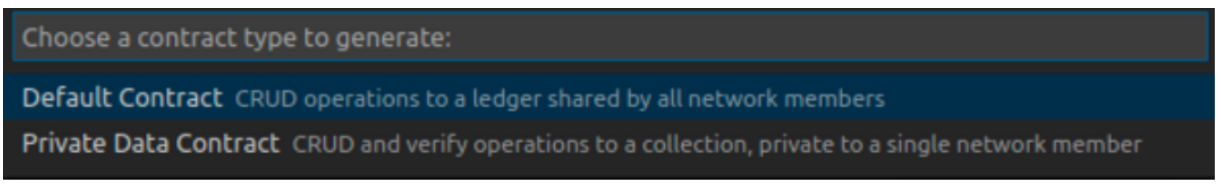


We will now create a smart contract project that will contain the files we need for our smart contract.

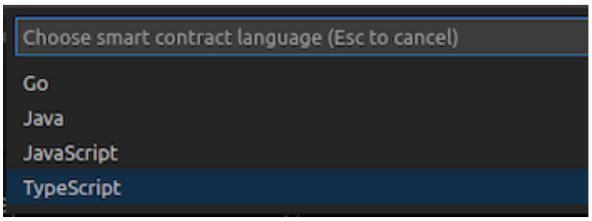
Move the mouse over the title bar of the Smart Contracts view, click the "..." that appears and select "Create New Project".



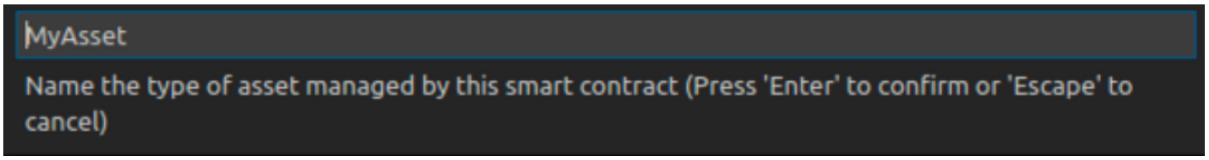
Press Enter to accept the Default Contract type.



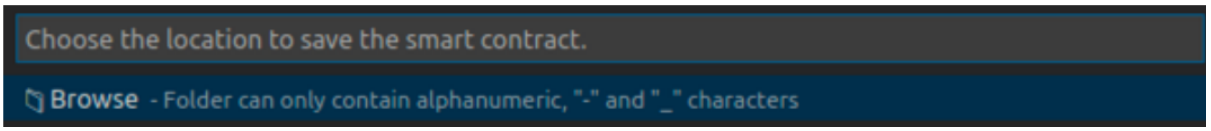
Click 'TypeScript'



Press Enter to accept the default asset type ("MyAsset").

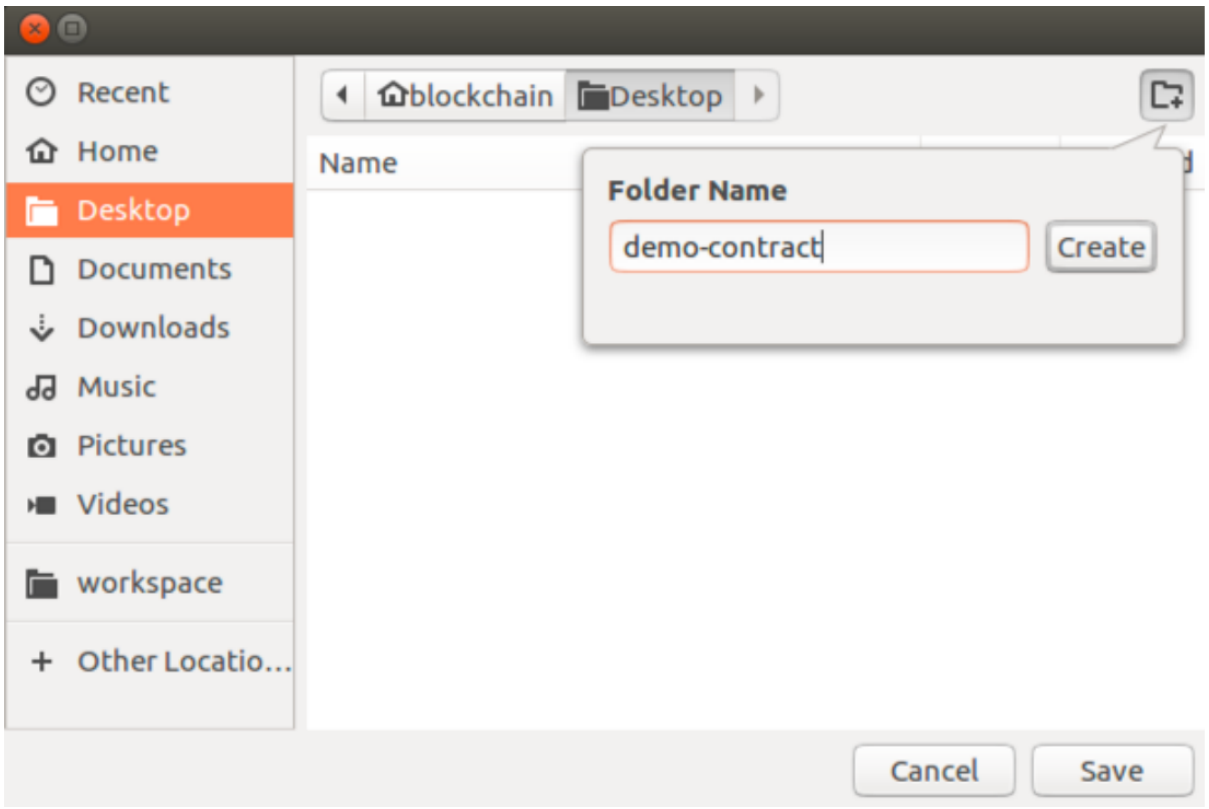


Click Browse to choose a target location of the project on the file system.

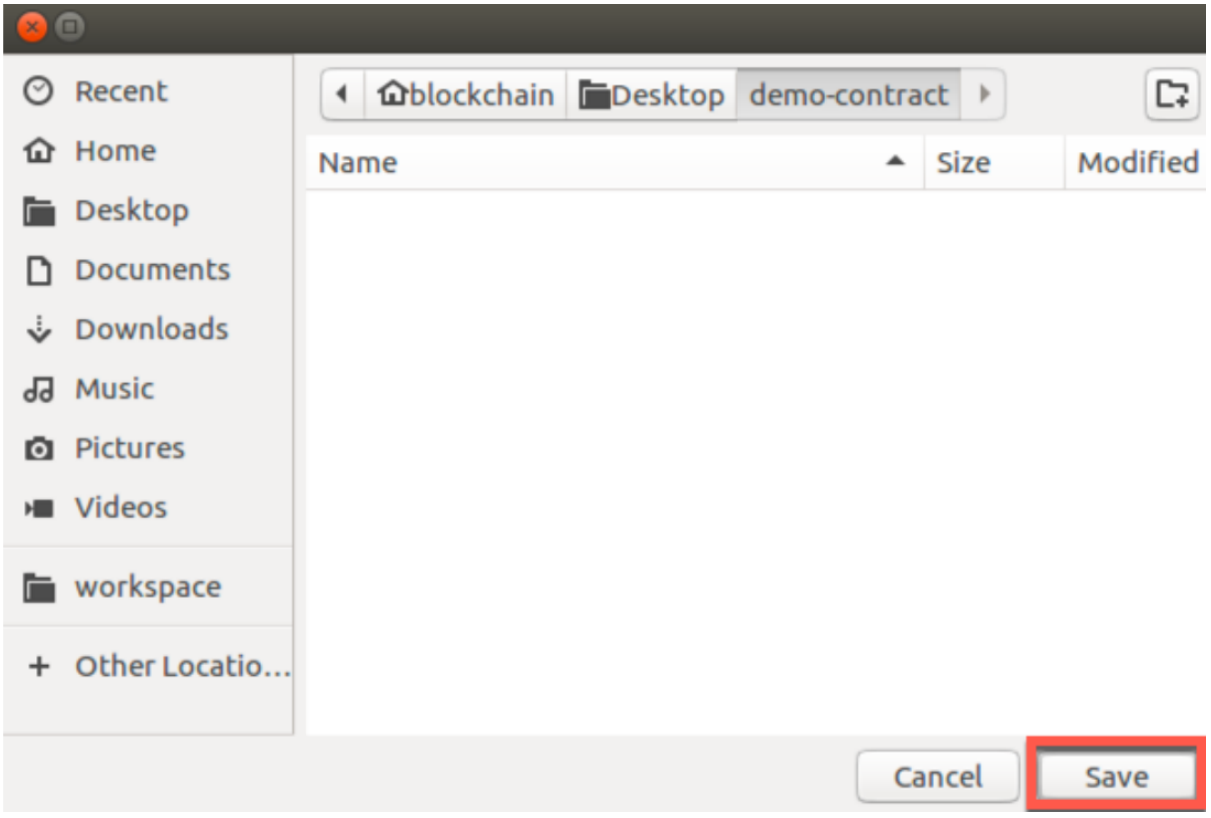


Navigate to the folder on your file system that you wish to save your files. (In the screenshots we will use the Desktop folder, for convenience.)

Click "New folder" to create a new folder to store the smart contract project, and name it "demo-contract"

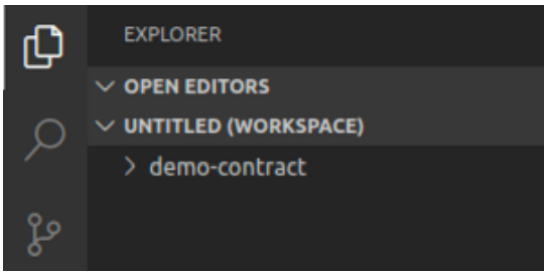


Click Save to select the new folder as the project root.

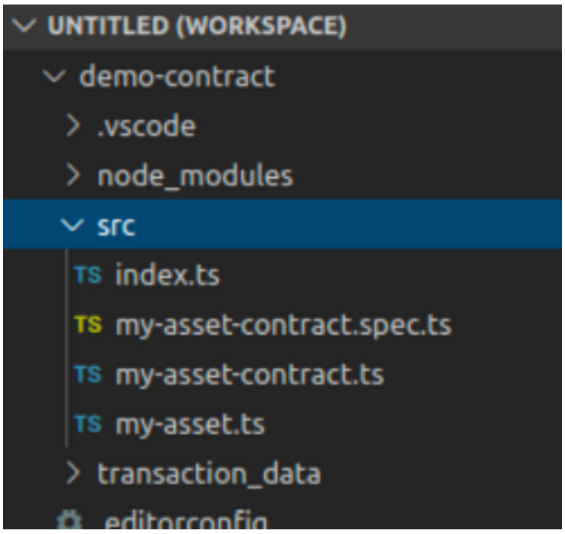


Select "Add to workspace" to tell IBM Blockchain Platform to add the project to your workspace

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



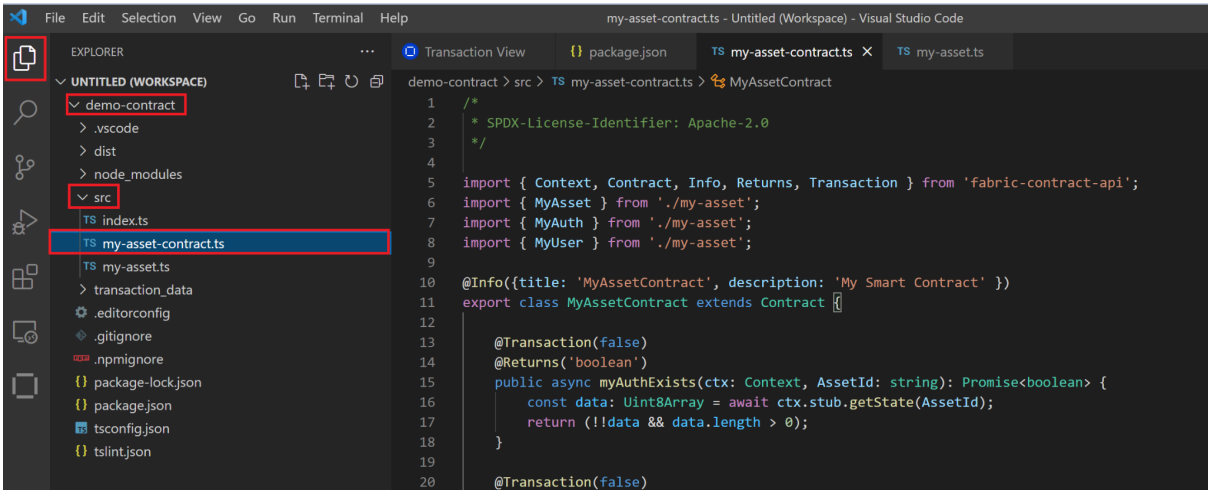
In the Explorer side bar, expand "demo-contract" -> "src".



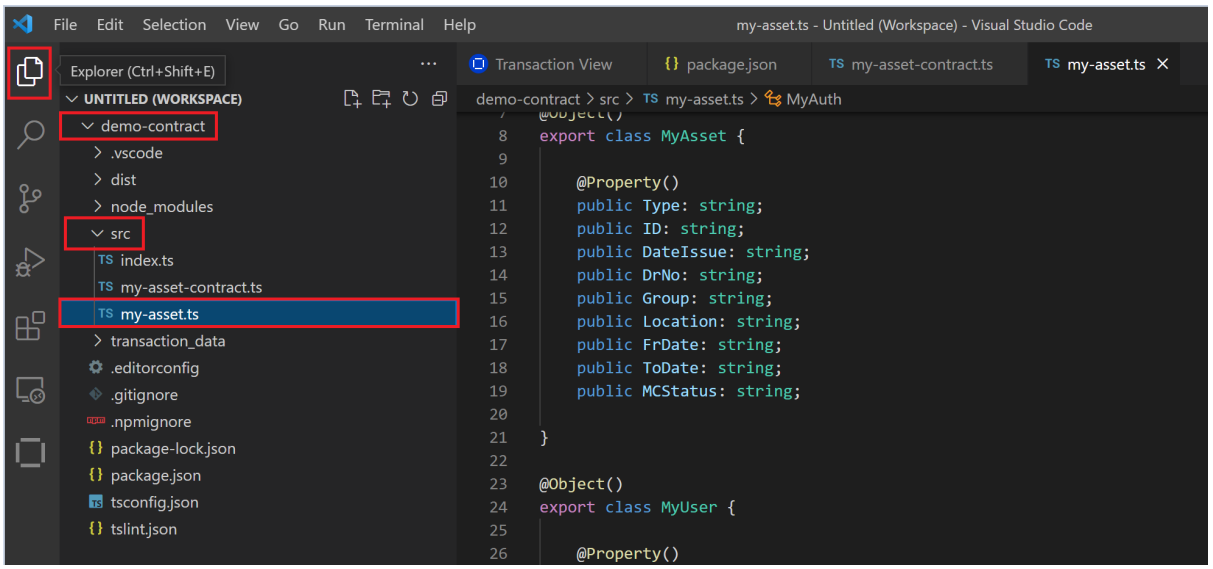
The smart contract is contained within the 'my-asset-contract.ts' file. The file name has been generated from the asset type you gave earlier.

Click on 'my-asset-contract.ts' to load it in the VS Code editor

Copy and paste to overwrite the content of my-asset-contract.ts from the same file in Github - https://github.com/bpteo2006/bc-project/tree/main/demo-contract/src folder.



Do the same for the file my-asset.ts

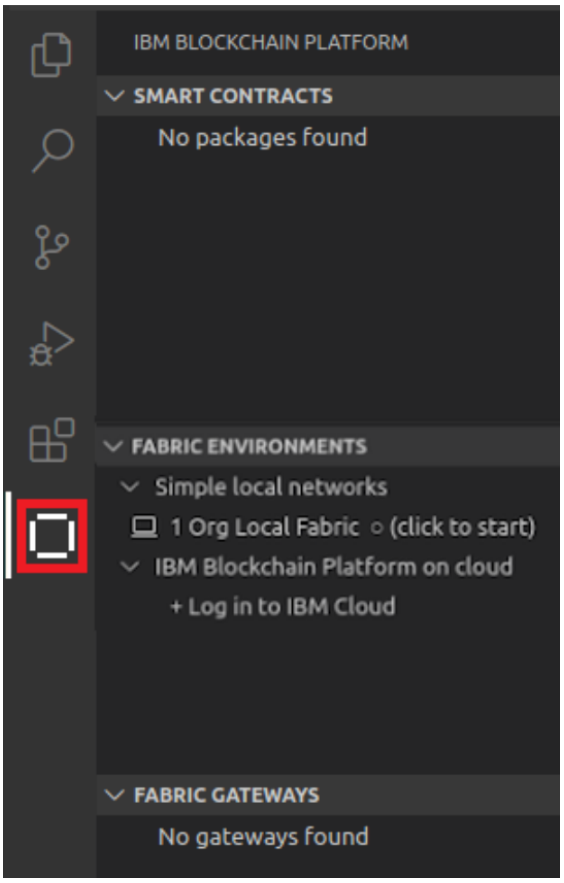


Deploying a smart contract

Start an instance of a Hyperledger Fabric network in the local workspace
Package the smart contract we previously created
Deploy the smart contract to the running Hyperledger Fabric network

Start the VS Code Hyperledger Fabric environment

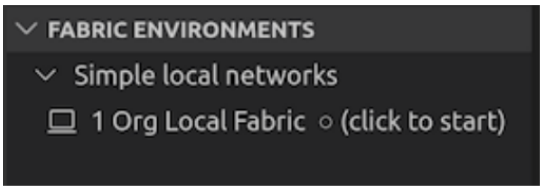
Click on the IBM Blockchain Platform icon in the activity bar to show the blockchain side bar



The Fabric Environments view

We will use the IBM Blockchain Platform VS Code Extension to test our smart contracts in a Hyperledger Fabric network. The extension comes with a pre-configured one organization network that runs on your local machine ("1 Org Local Fabric").

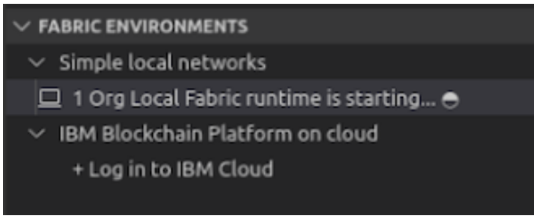
The available networks are shown in the Fabric Environments view.



If you do not have a "1 Org Local Fabric" environment, then click + Add local or remote environment to create the environment. If creating the environment, click Create new from template and click 1 Org template. Enter "1 Org local Fabric" as the name, select "V2_0 (Recommended)" for the channel capabilities and when you press enter the environment will start which may take up to 5 minutes.

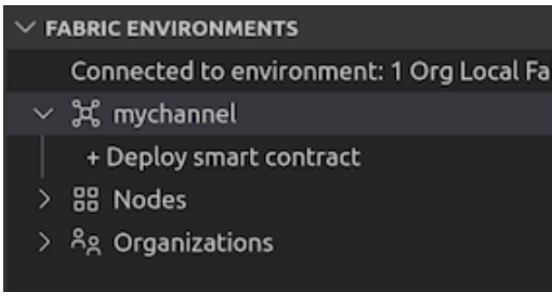
The required Hyperledger Fabric components are automatically downloaded and started when you select it.

In the Fabric Environments view, click "1 Org Local Fabric O (click to start)" This will download and start the embedded instance of Hyperledger Fabric, and may take up to five minutes to complete.

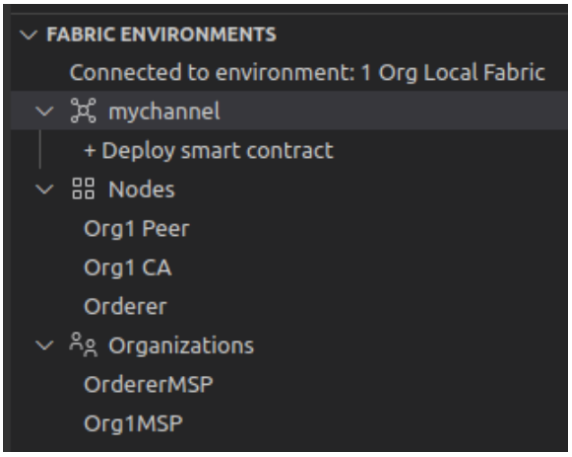


When Hyperledger Fabric has fully initialized, the view will change to show the channels, nodes and organizations in the local environment.

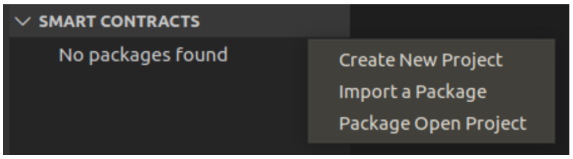
Click on the environment to see the details.



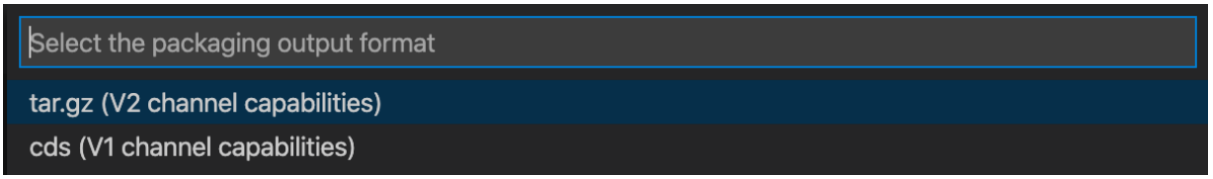
- Channels define the scope of each network, and form one method of choosing how organizations share data. Deployed smart contracts that are available to the network will show under channels.
- Nodes are the Hyperledger Fabric components that make the system work. There are three types of nodes:
- Peers which host ledgers and execute smart contracts
- Orderers which assert transaction order and distribute blocks to peers
- Certificate authorities which provide the means of identifying users and organizations on the network
- Organizations are the members of the blockchain network. Each organization will consist of many different users and types of users.
- For more about the components that make up a Hyperledger Fabric network, see the Hyperledger Fabric documentation.
- If you expand the various sections you'll see the various defaults for each of these elements:
- Three nodes: a single peer called Org1 Peer, an ordering node called Orderer and a certificate authority called Org1 CA.
- Two organizations, with identifiers of 'OrdererMSP' and 'Org1MSP'. The former will own the orderer and the latter the peer; it is good practice to use separate organizations for orderer nodes and peers.
- There is a single default network channel called mychannel.
- By default there are no smart contracts deployed to the channel.



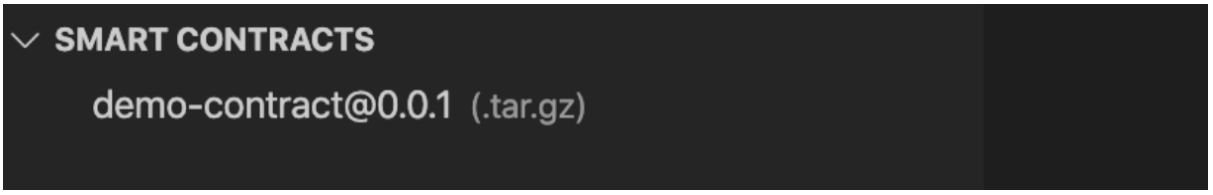
Move the mouse over the title bar of the Smart Contracts view, click the "..." that appears and select "Package Open Project".



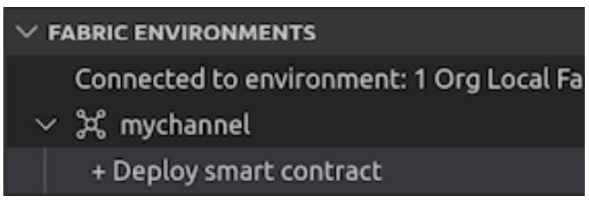
Select "tar.gz (V2 channel capabilities)" to deploy this contract in future to a channel with V2 channel capabilities.



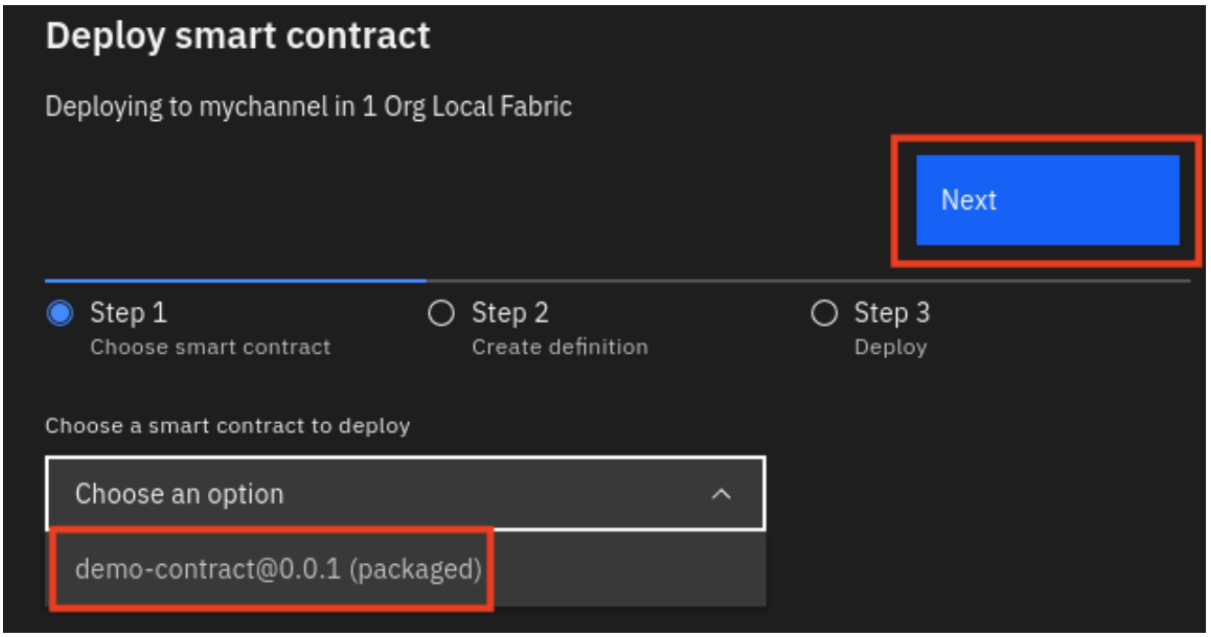
The Smart Contracts view will be updated to show the new package.



In the Fabric Environments view, click "mychannel" -> "+ Deploy smart contract".

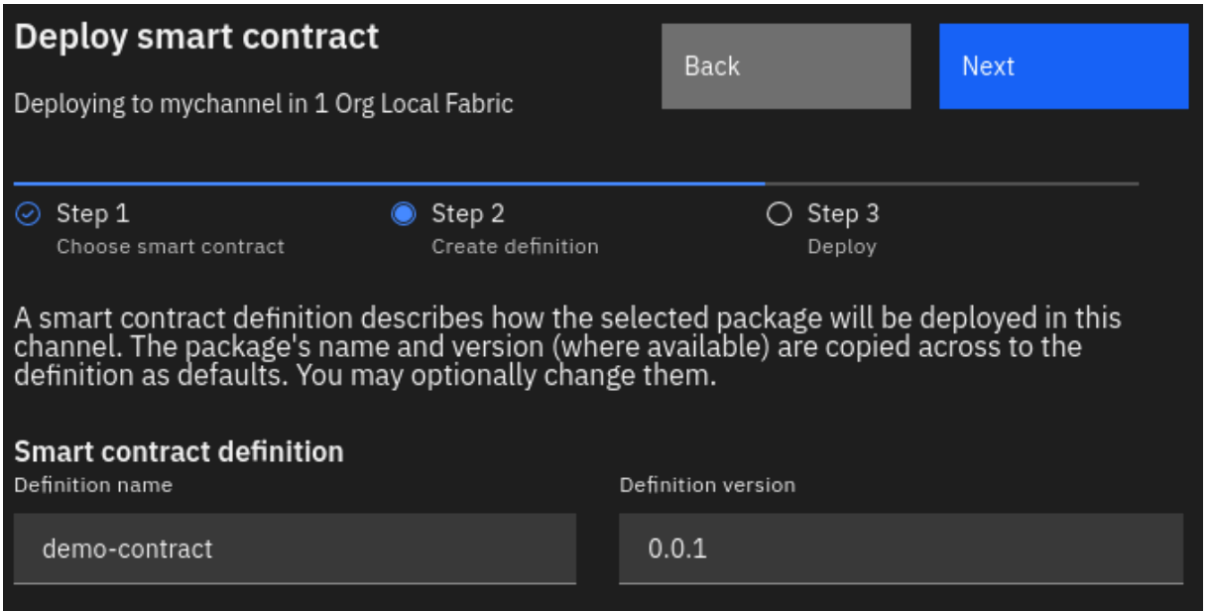


In the Deploy Smart Contract form, select "demo-contract@0.0.1" from the drop down list, and click 'Next'

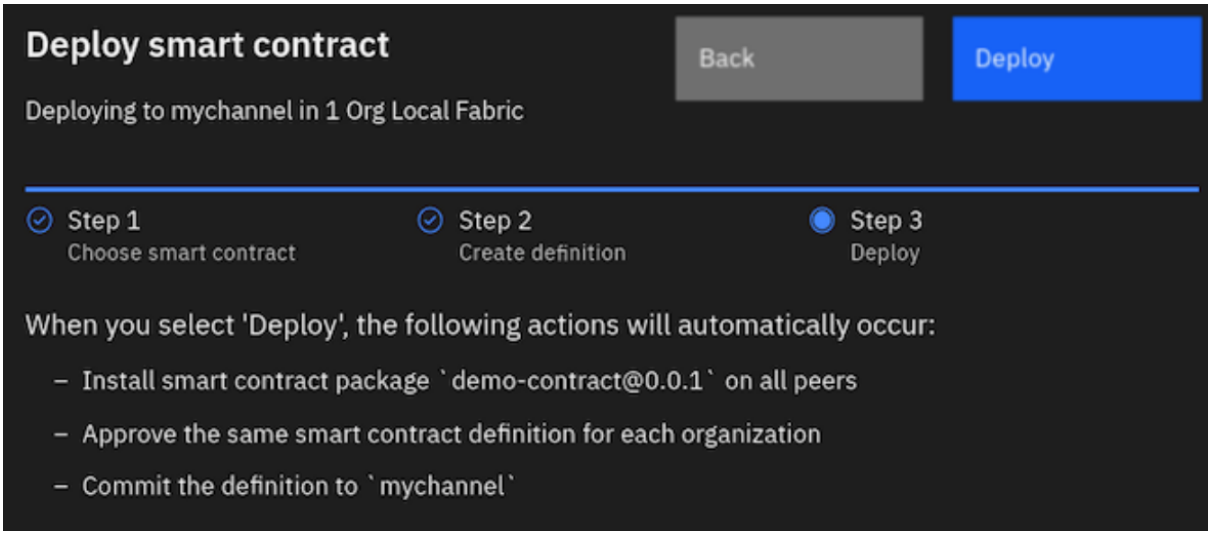


Click 'Next' to move to Step 3 of the deploy.

For TypeScript smart contracts, both the name and version are taken from the package.json file in the root of the smart contract project.

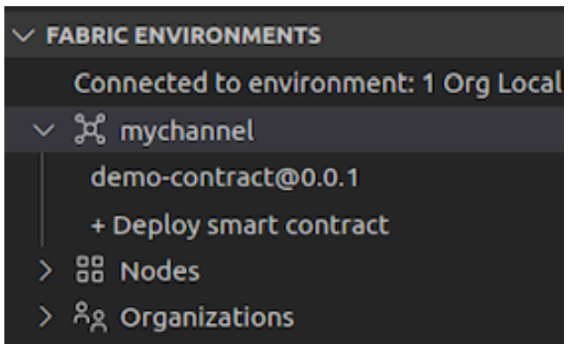


Click 'Deploy' to start the deployment



Deployment may take a few minutes to complete.

When deployment has completed you will see the new smart contract listed under "mychannel" in the Fabric Environments view.



Invoking a smart contract from VS Code

Connect to the Hyperledger Fabric gateway

In order to submit transactions in Hyperledger Fabric you will need an identity, a wallet and a gateway.

Identities, wallets and gateways

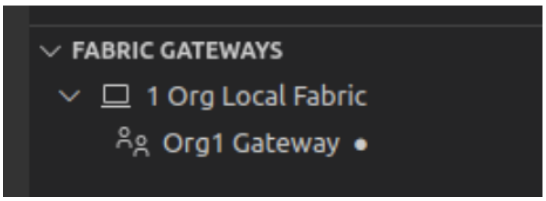
The resources that you can access in a Hyperledger Fabric network are determined according to your identity. Your identity is typically represented by an X.509 certificate issued by your organization, and stored in your wallet. Once you have an identity and a wallet, you can create a gateway that allows you to submit transactions to a network.

A gateway represents a connection to a set of Hyperledger Fabric networks. If you want to submit a transaction, whether using VS Code or your own application, a gateway makes it easy to interact with a network: you configure a gateway, connect to it with an identity from

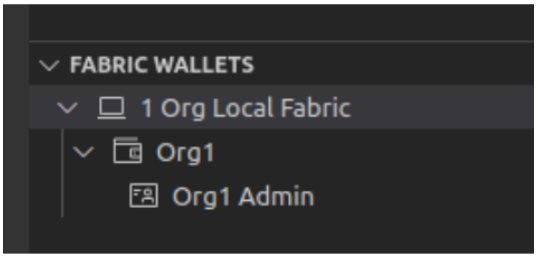
your wallet, choose a particular network, and start submitting transactions using a smart contract that has been deployed in that network.

Gateways and Wallets in VS Code

When the one organization network was created, a gateway was created at the same time; this is now shown in the Fabric Gateways view. This view allows you to add new gateways to submit transactions to both local and remote Hyperledger Fabric networks.



Furthermore, an Org1 wallet with an identity Org1 Admin has been created in the Fabric Wallets view.

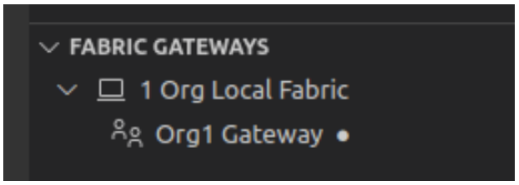


The Org1 Admin identity will be used to submit and evaluate transactions through the smart contract.

Connecting to the Fabric Gateway in VS Code

We will now connect to a gateway using the Org1 Admin identity.

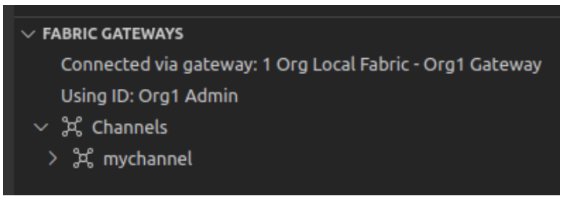
In the Fabric Gateways view, click "Org1 Gateway".



There is only 1 identity in the Org1 wallet, so the IBM Blockchain Platform VS Code extension will use that identity automatically and connect to the local Hyperledger Fabric gateway.

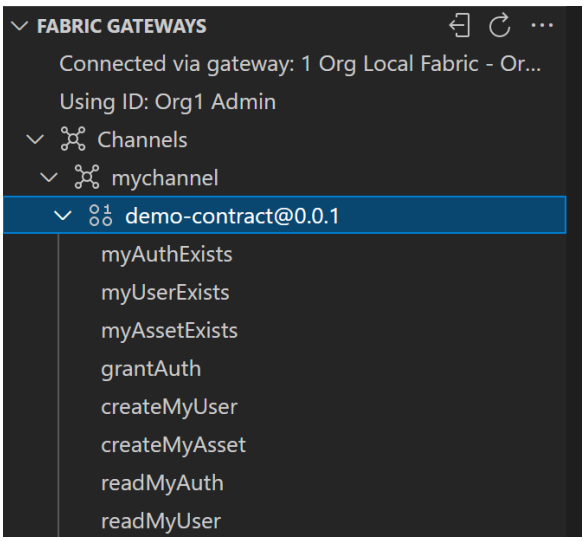
Once connected, notice that the view changes to reflect the channels available to the connected gateway.

Review the channels.



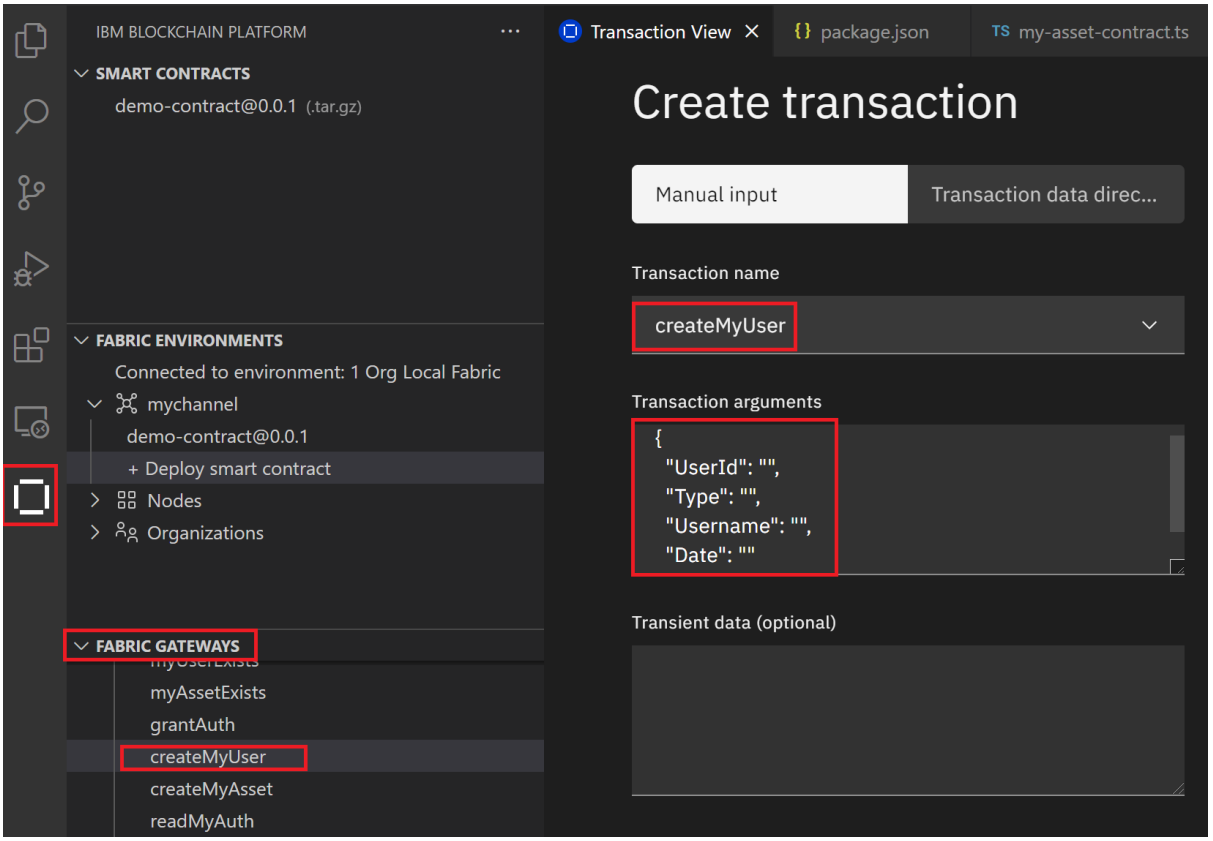
The connected gateway shows the list of channels on the network, in this case just the single mychannel.

Fully expand the Channels tree in the Fabric Gateways view to show the available transactions.



Register user

On the Transaction view click on the dropdown for Transaction Name and select createMyUser



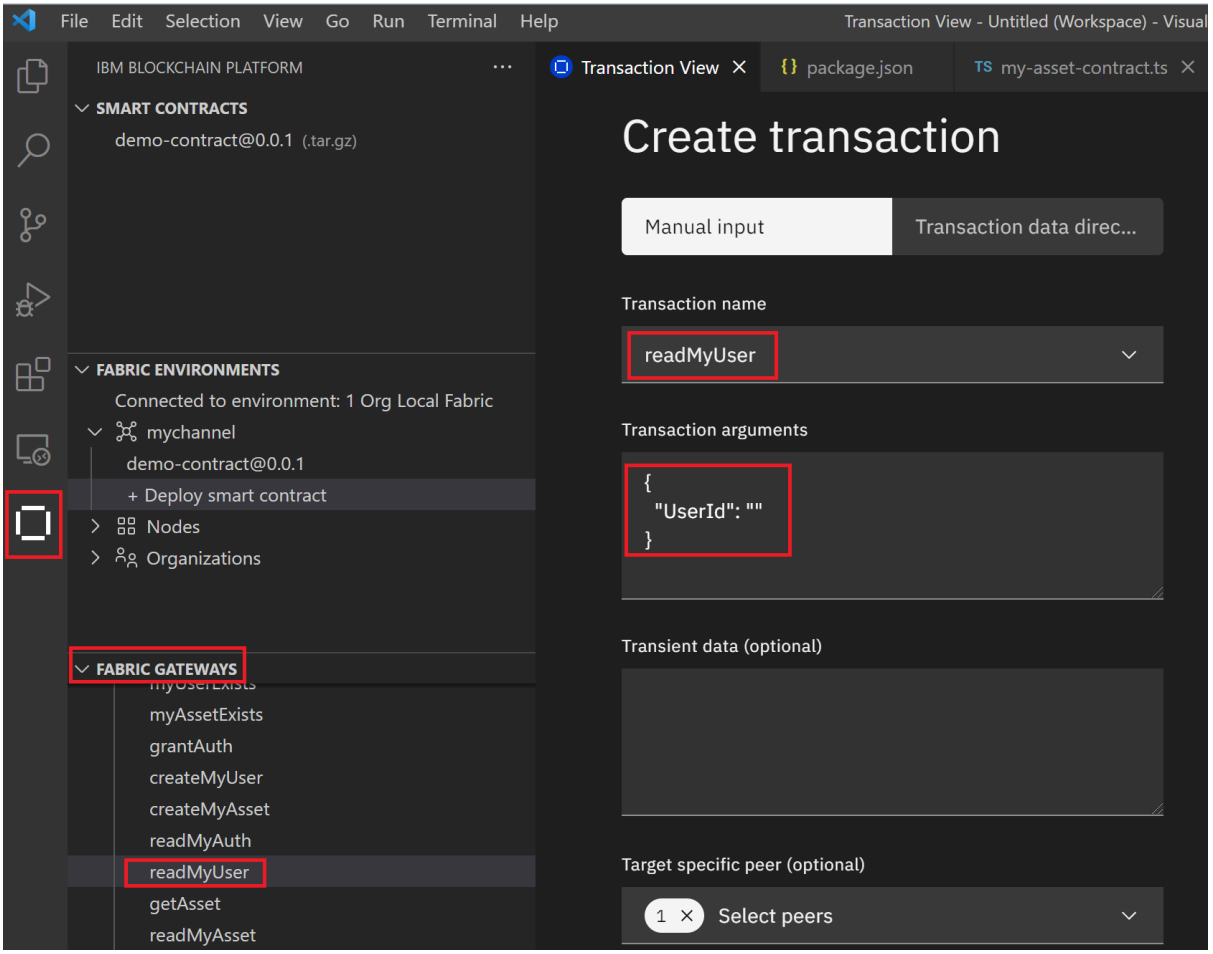
Replace the Transaction arguments:

```
{
  "UserId": "",
  "Type": "",
  "Username": "",
  "Date": ""
}
```

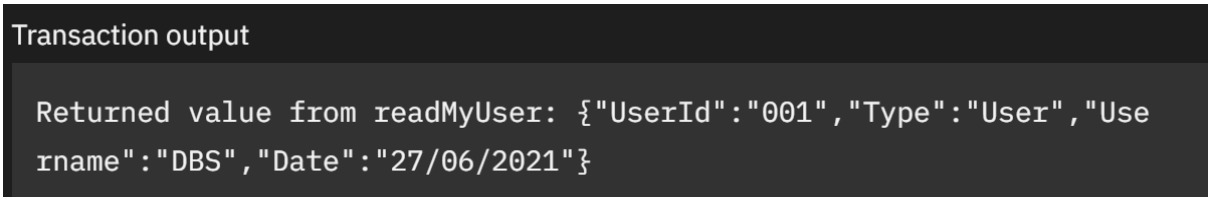
Let “UserId” : <3 digits e.g “001”>, “Type”: “User”, “Username”: <Free text e.g “DBS”>, and “Date”: <is the date of creation e.g, “27/06/2021”>

Verify User record is created successfully,

On the Transaction view click on the dropdown for Transaction Name and select readMyUser



Evaluate the "readMyUser" transaction to return the asset. Specify "001" for UserId.



Invoking a smart contract from an external application

- Build a new TypeScript application that interacts with Hyperledger Fabric
- Run the application to submit a new transaction

Export the network details

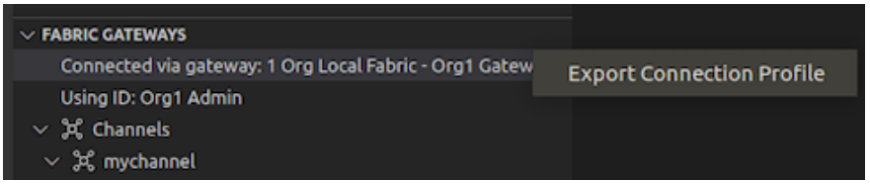
As we have seen, to interact with a Hyperledger Fabric network it is necessary to have:

- a connection profile
- a wallet containing one or more identities

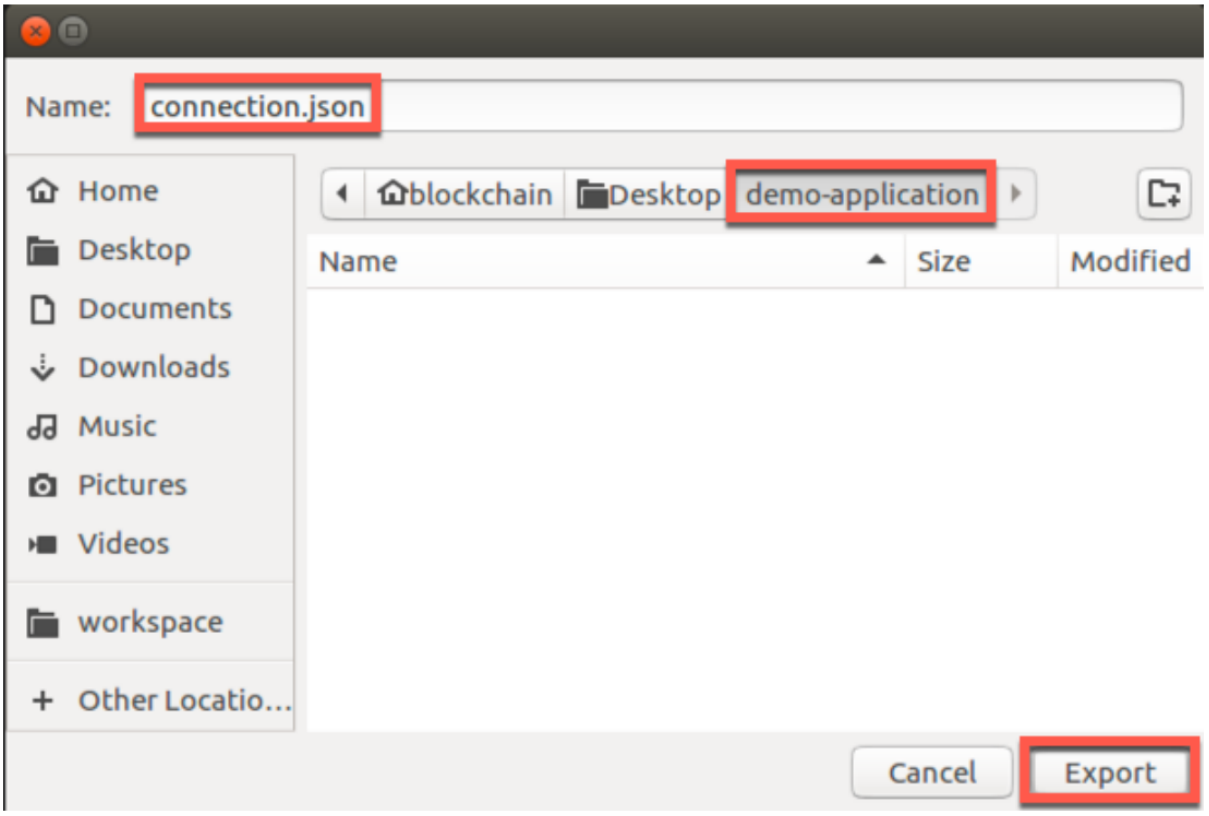
Our application will use the same identity and connection profile used by VS Code to interact with the sample network.

We will start by exporting a connection profile.

With the gateway connected, move the mouse over the Fabric Gateways view, click the ellipsis that appears and select "Export Connection Profile".



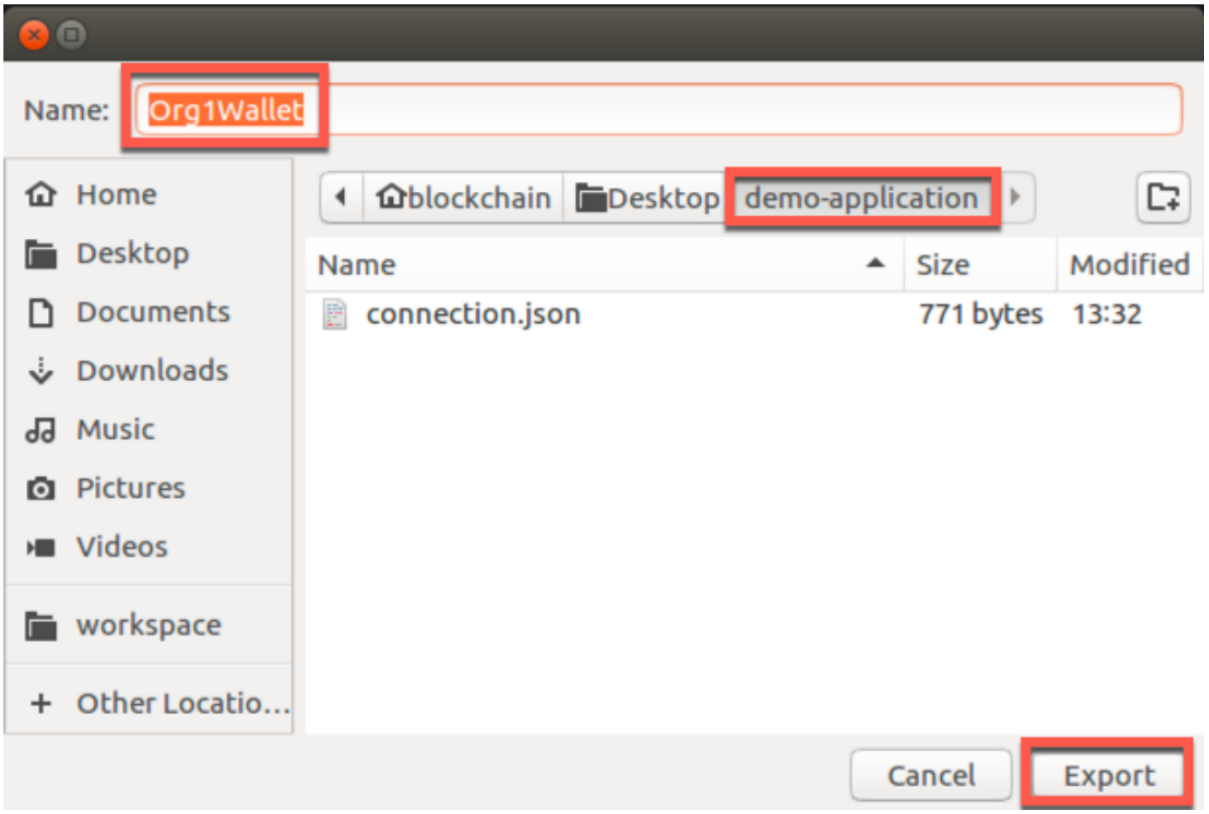
Create a new folder called 'demo-application' as a peer of the demo-contract project we created earlier. Give the connection profile a convenient name ('connection.json') and export it into the new folder



We will now export our wallet.

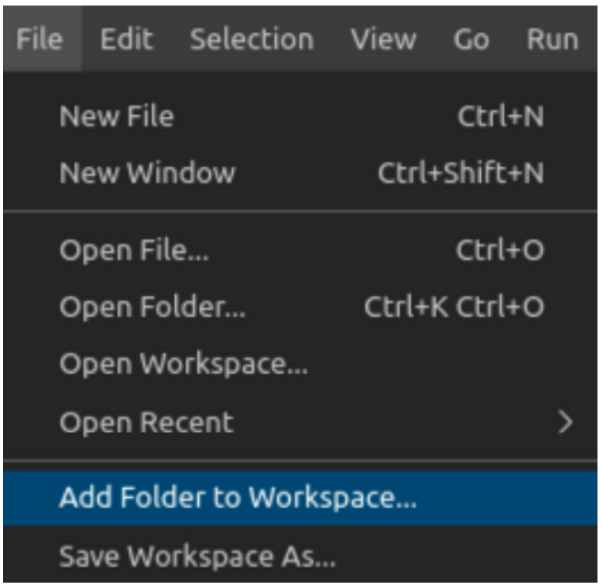
In the Fabric Wallets view, expand '1 Org Local Fabric', right click 'Org1' and select 'Export Wallet'.

Navigate into the 'demo-application' folder, change the name to 'Org1Wallet' and click Export to save the wallet

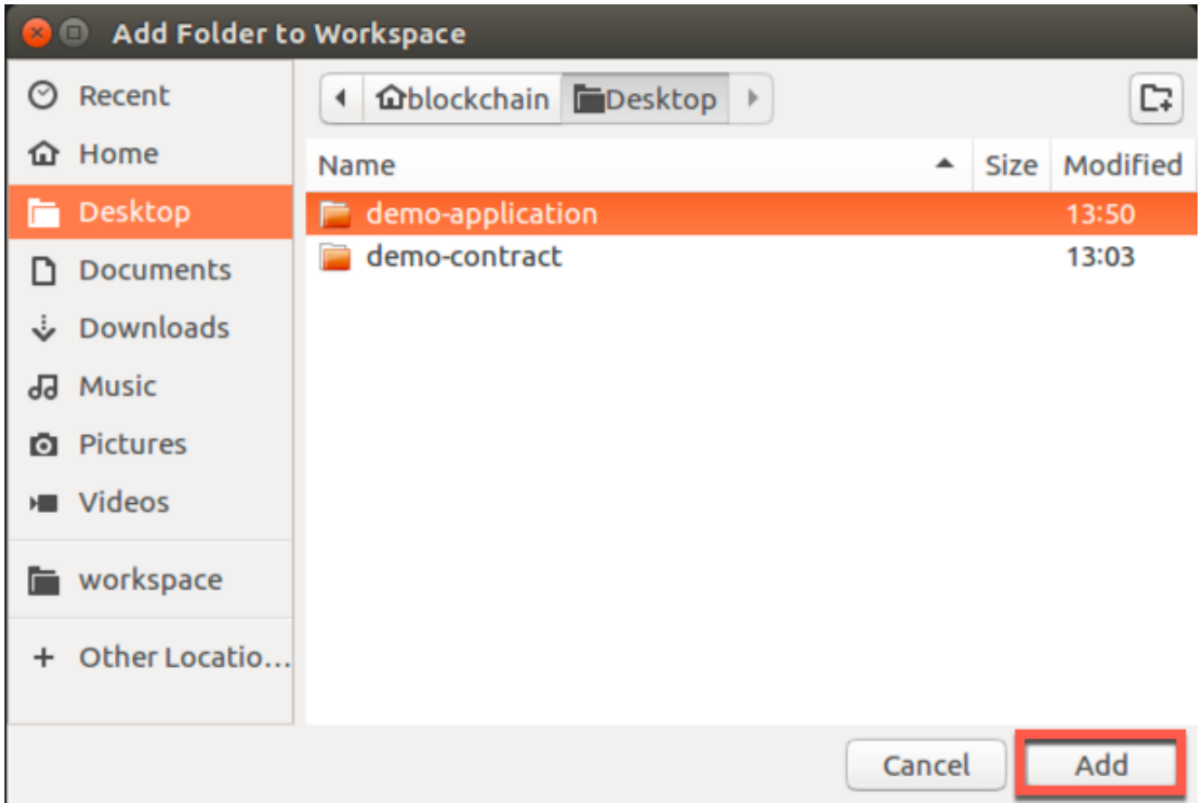


Create the external application Let's start by adding the demo-application folder to the VS Code workspace.

From the VS Code menu bar click "File" -> "Add Folder to Workspace..."

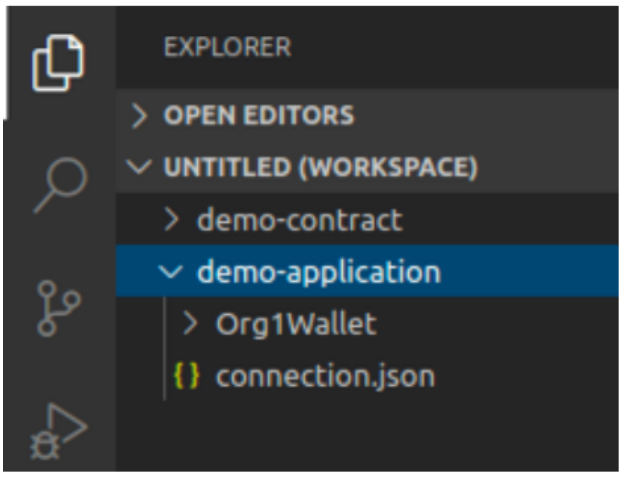


Highlight the 'demo-application' folder and click 'Add'



After adding the folder to the workspace, VS Code will show the Explorer side bar, with the new 'demo-application' folder underneath 'demo-contract'.

The demo-application folder should contain a subfolder called 'Org1Wallet' (the wallet) and a connection profile called 'connection.json'.

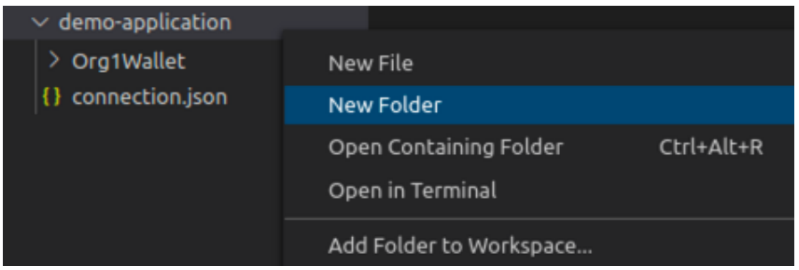


In order to build a working Typescript application we will now create three files in addition to the wallet and connection profile:

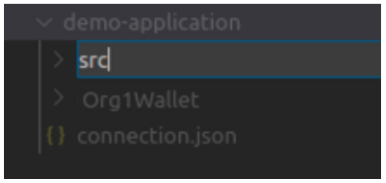
- create.ts: The TypeScript application containing the logic required to connect to the Hyperledger Fabric sample network and submit a new transaction.
- tsconfig.json: TypeScript compiler options, including source and destination locations
- package.json: Application metadata, including the Hyperledger Fabric client SDK dependencies, and commands to build and test the application.

We will create a file create.ts inside a src folder.

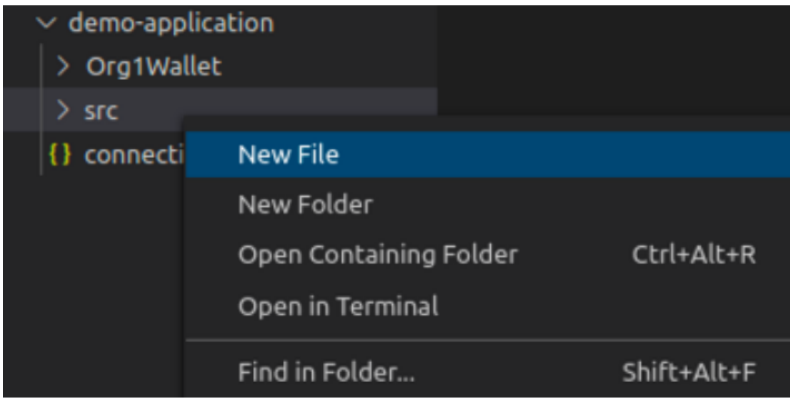
Right-click 'demo-application' and select 'New Folder'



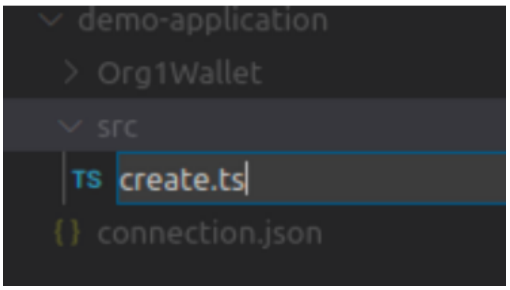
Name the folder 'src'.



Right-click 'src' and select 'New File'.



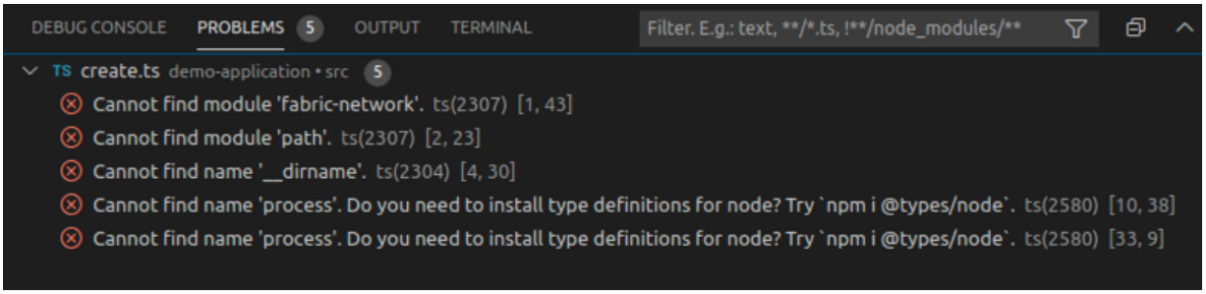
Name the file 'create.ts'.



In the editor view for the new create.ts file, copy and paste the following text. (Copy and paste the content of this file from the said file in Github - https://github.com/bpteo2006/bc-project/tree/main/demo-application/src).

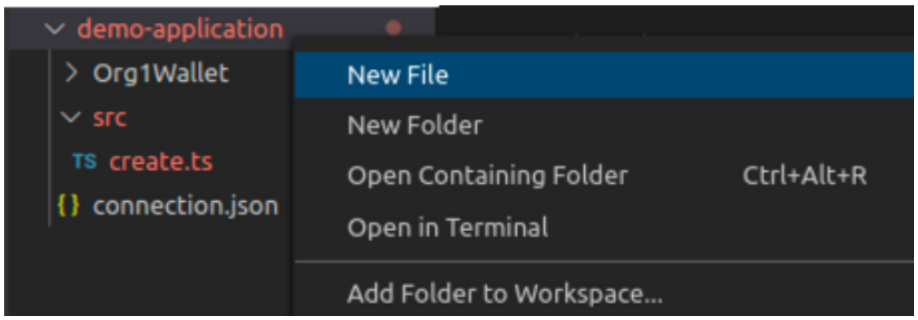
Save the file ('File' -> 'Save').

When you save, you will see various errors reported by VS Code. This is because we have not yet configured the set of external dependencies.

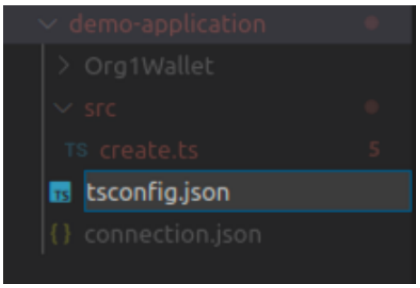


Next we will create the tsconfig.json file.

Right-click 'demo-application' (NOT 'src') and select 'New File'.



Name the file 'tsconfig.json'.



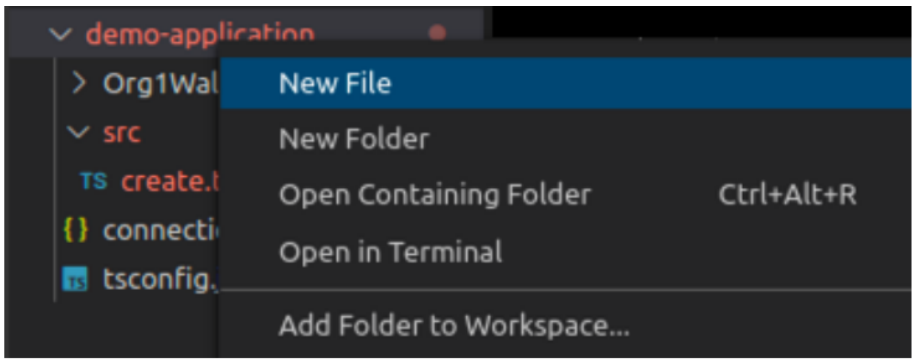
In the editor view for the new tsconfig.json file, copy and paste the following text. (Copy and paste the content of this file from the said file in Github - https://github.com/bpteo2006/bc-project/tree/main/demo-application).

Importantly, the tsconfig.json file specifies the source and output folders ('src' and 'dist' respectively), and enables compiler options for strict syntax checking of our Typescript.

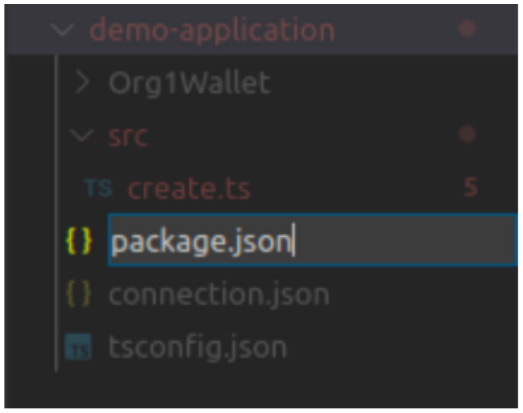
Save the file ('File' -> 'Save').

Next, we will create the package.json file.

Right-click 'demo-application' (NOT 'src') and select 'New File'.



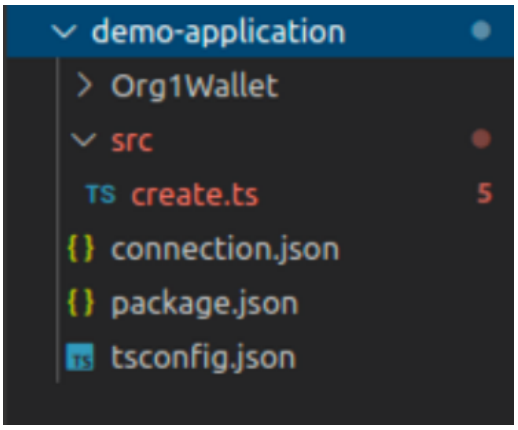
Name the file 'package.json



In the editor view for the new package.json file, copy and paste the following text. (Copy and paste the content of this file from the said file in Github - https://github.com/bpteo2006/bc-project/tree/main/demo-application)

Save the file ('File' -> 'Save').

At this stage, the application structure should contain a wallet folder ('Org1Wallet'), a source folder ('src') which contains a single file ('create.ts'), a connection profile ('connection.json'), package.json and tsconfig.json. If this is not the case, check the instructions and move and edit files as necessary.

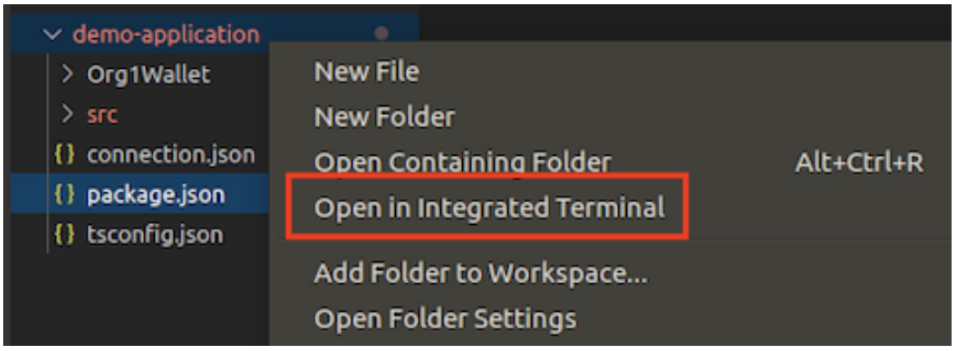


In the next section we will build the application.

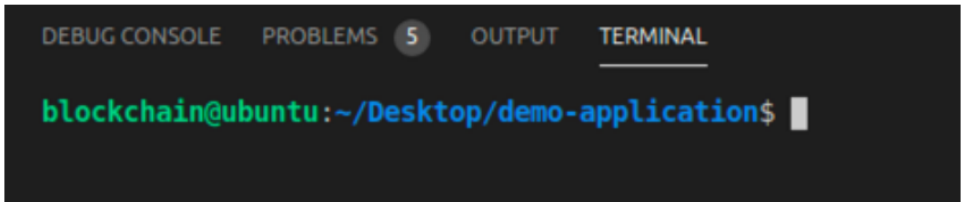
Build the external application

Even though we've specified our application's dependencies inside package.json, we haven't yet loaded the required modules into our workspace and so errors remain. The next step is to install these modules so that the errors disappear and allow our application to be built and run.

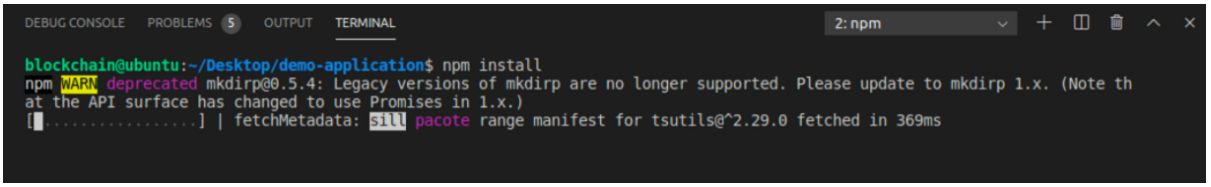
Right-click 'demo-application' and select 'Open in Terminal'



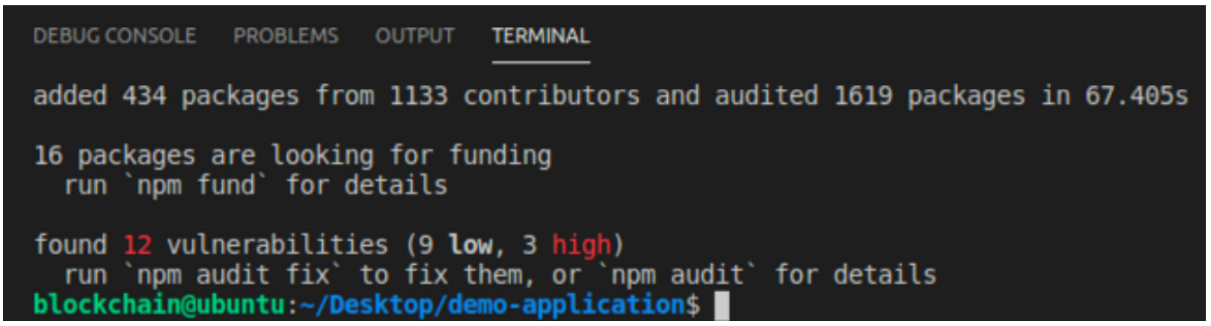
This will bring to focus a terminal prompt inside VS Code



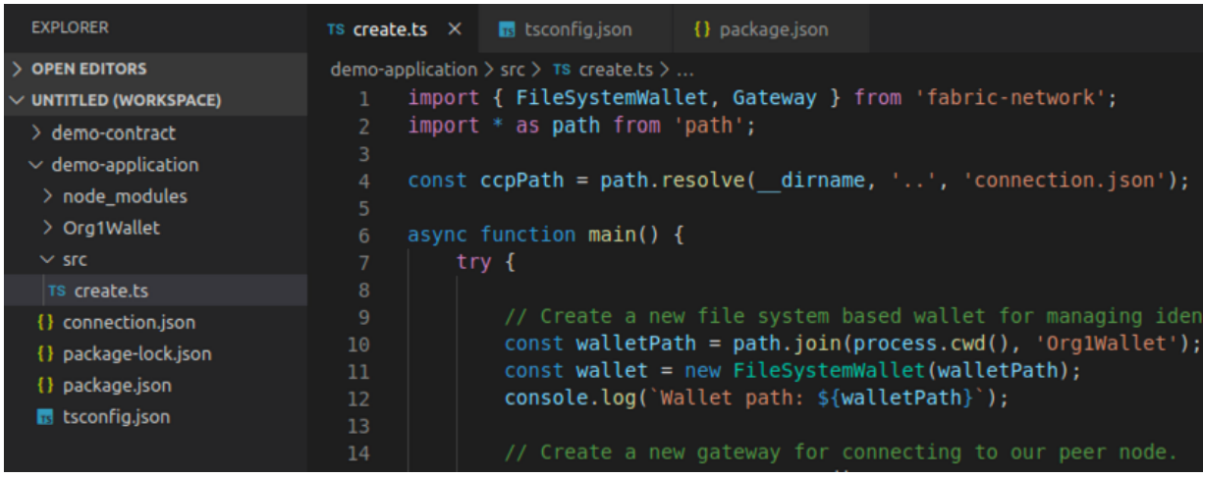
In the terminal window type npm install and press Enter.



This will download the module dependencies into our project folder and may take a minute or so to complete. When it has finished, the prompt will return.

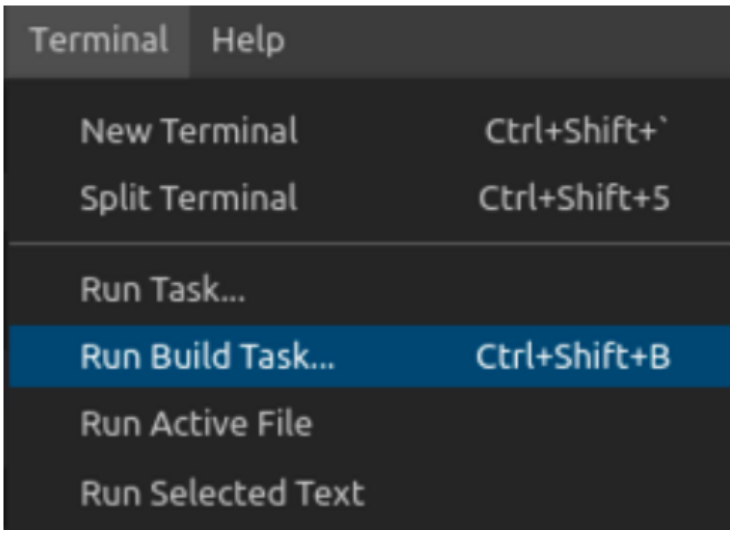


The errors that were previously reported will now all vanish, and the demo-application folder will now contain a new 'node_modules' folder that contains the imported dependencies.

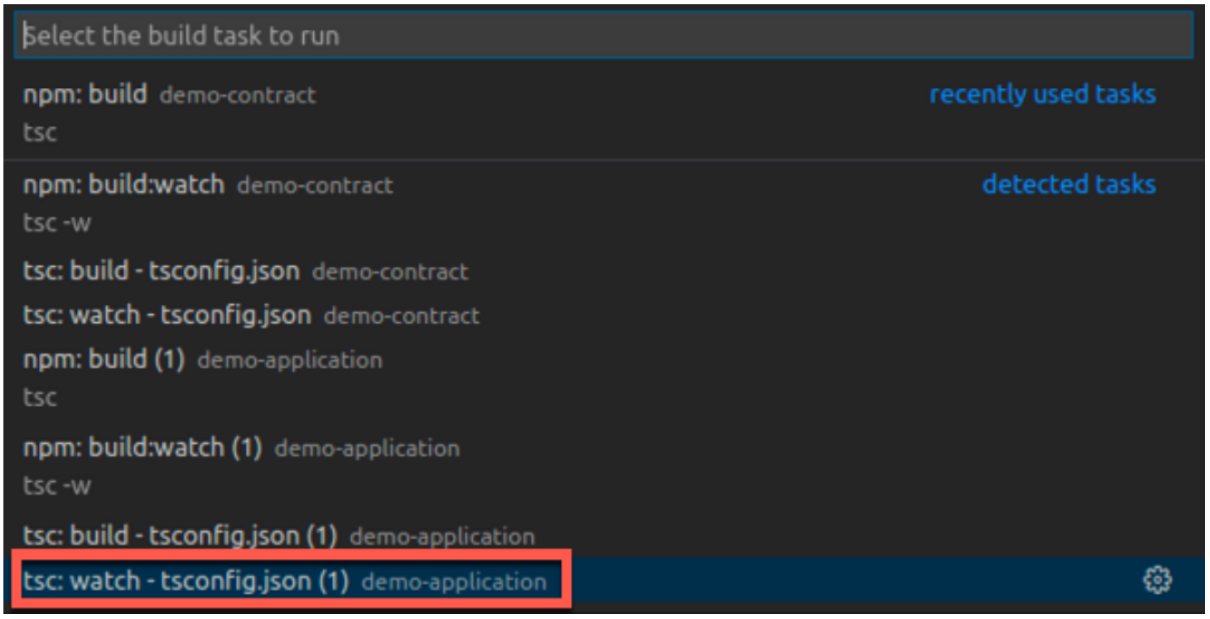


With our dependencies resolved we can now build our application.

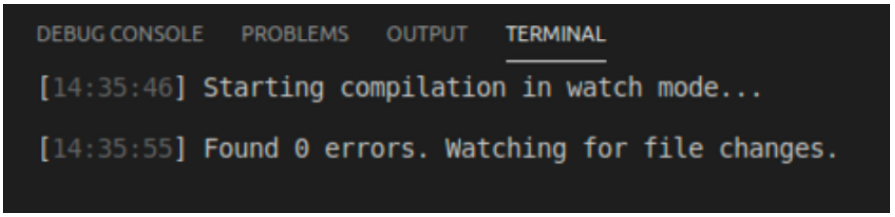
In the main VS Code menu, click 'Terminal' -> 'Run Build Task...'.



In the command palette, find and click 'tsc: watch - tsconfig.json demo-application'. Take care to select the correct option as there will be similar looking alternatives (build options for our smart contract project, for example). You might need to scroll the list to find the correct task



After a few seconds, the application will have been built and the compiler will enter 'watch' mode, which means that any changes to the source will cause an automatic recompilation. Using watch mode is useful as it means you do not have to force a rebuild each time you make a change.



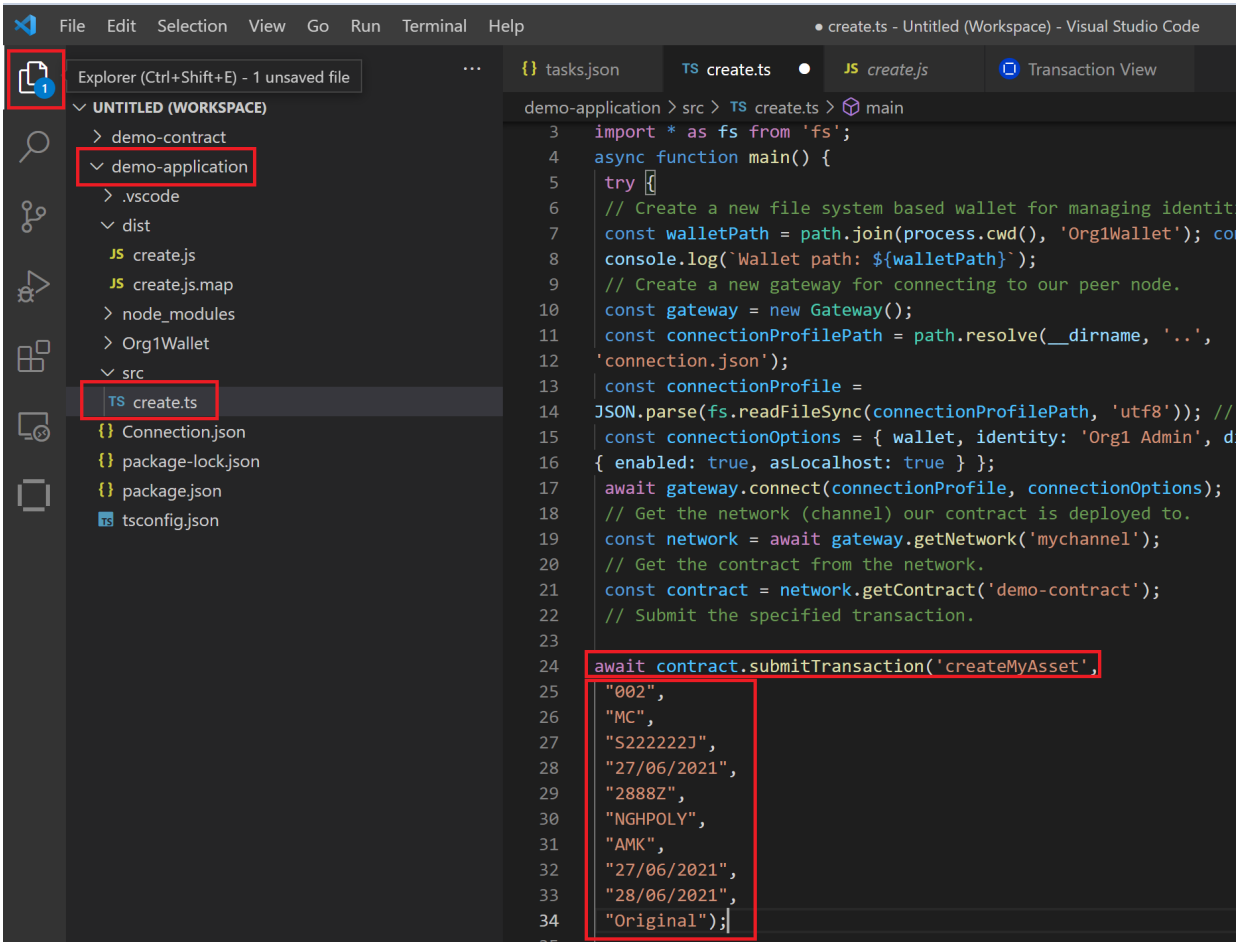
Doctor issue MC

Next we will simulate a doctor issuing an MC to his patient. Below are the sample data in JSON format:

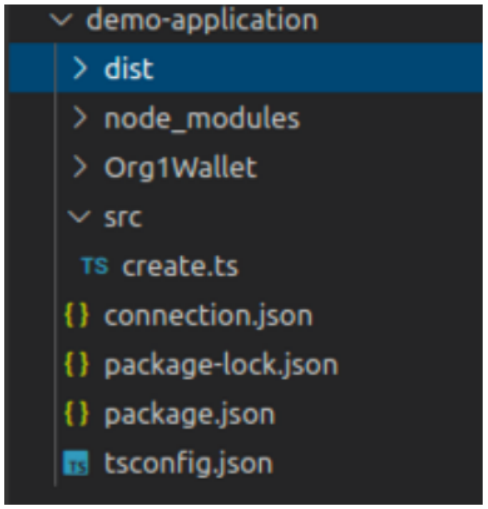
```
  "AssetId": "002",
  "Type": "MC",
  "ID": "S222222J",
  "DateIssue": "27/06/2021",
  "DrNo": "2888Z",
  "Group": "NGHPOLY",
  "Location": "AMK",
  "FrDate": "27/06/2021",
  "ToDate": "28/06/2021",
  "MCStatus": "Original"
```

However, in the external application we will only enter the value:

```
await contract.submitTransaction('createMyAsset',
 "002",
 "MC",
 "S222222J",
 "27/06/2021",
 "2888Z",
 "NGHPOLY",
 "AMK",
 "27/06/2021",
 "28/06/2021",
 "Original");
```

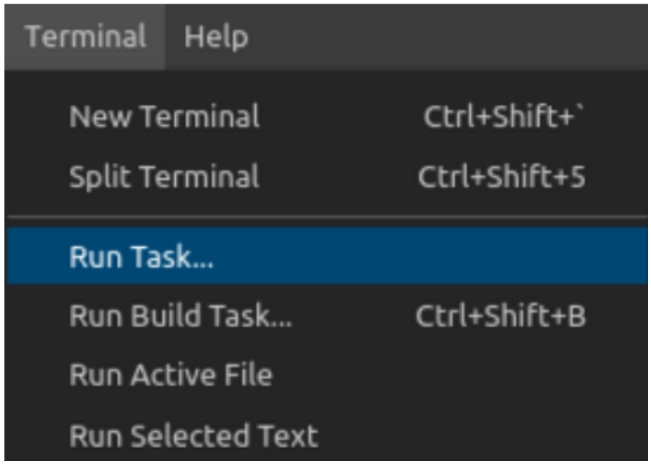


You will also see a new 'dist' folder underneath the demo-application project. This contains the built version of the application, which is what we will run in the next section.



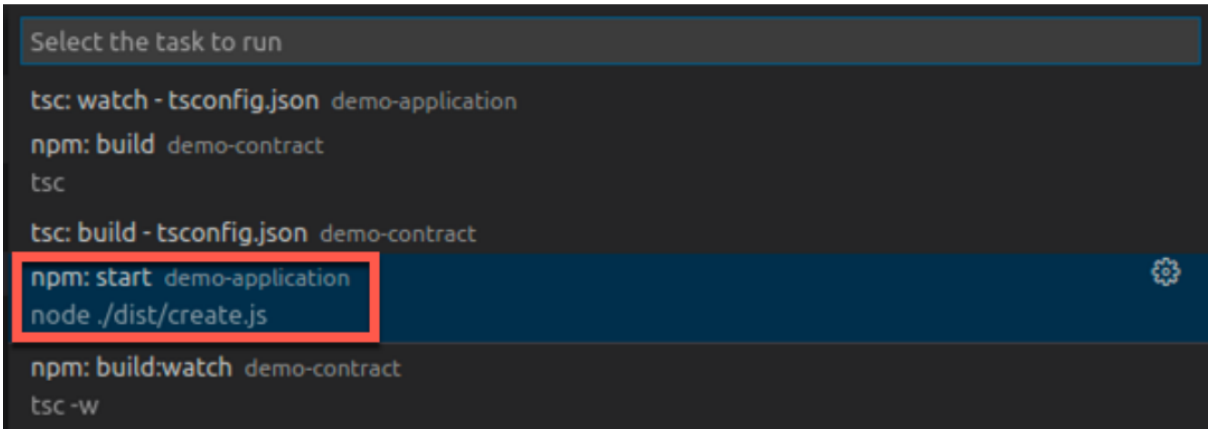
Run the external application We can run our application wherever we choose - it is just a standard Node.js application. However, VS Code additionally provides an integrated terminal facility whereby different tasks can be run in different terminals. We'll use that now to make all our outputs easily accessible within VS Code.

In the main VS Code menu, click 'Terminal' -> 'Run Task...'.



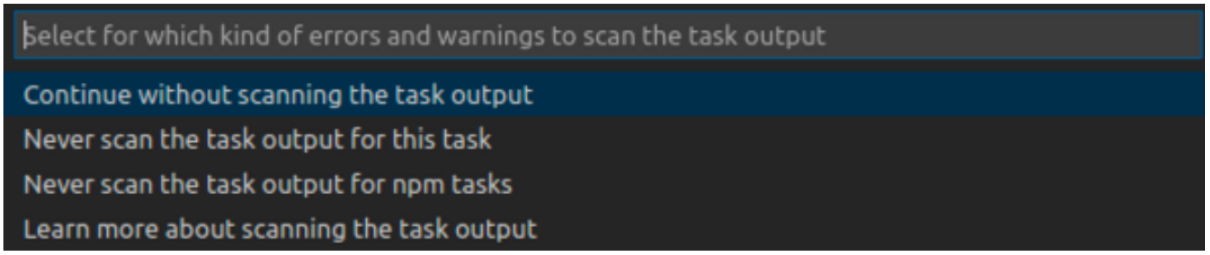
Find and select the task 'npm: start demo-application'.

Again, take care to select the correct task as there might be alternatives that look very similar. You might need to scroll the list to find the correct task. You may also need to click Show All Tasks.



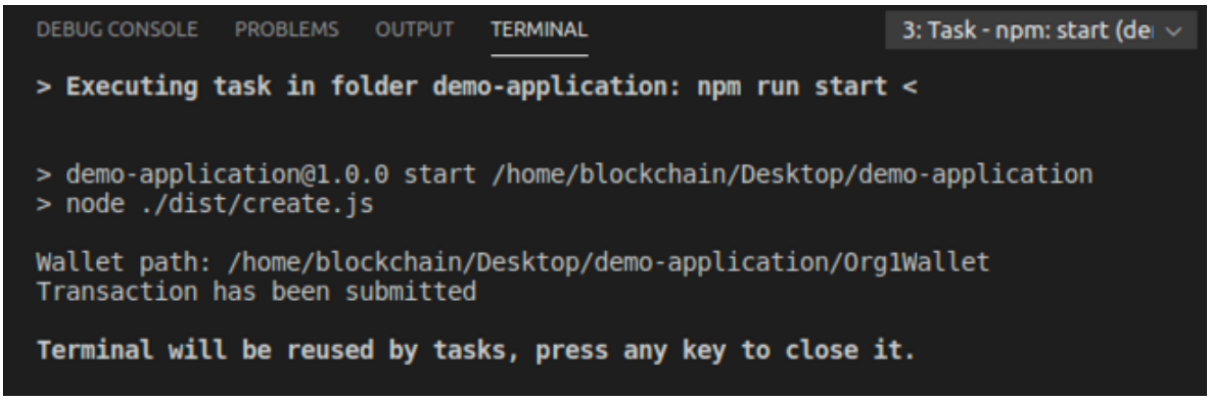
VS Code will offer to automatically scan the task output for us, but we will not do that here.

Click 'Continue without scanning the task output'.



The task will now run. What it will do is run the start script that is defined in demo-application's package.json, which is the command node ./dist/create.js. You could run the same node command in any appropriately configured environment and achieve the same output.

The task will be run inside a VS Code terminal and, after a brief pause, you will see it complete successfully.



Press any key in the terminal window to free it up for other tasks.
The Terminal window will switch back to what was there previously.

Patient/Employee granting authority

Next we will simulate the patient/employee grant authority to his employer to access his MC by submitting a grantAuth transaction.

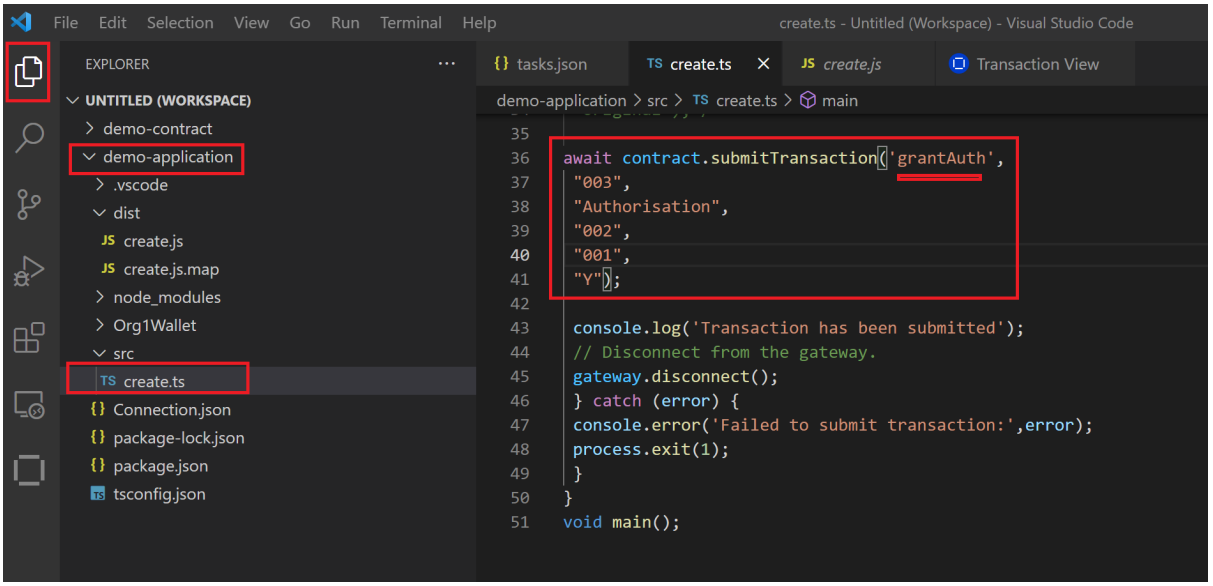
The following is the input for this transaction:

```
{
  "AuthCode": "003",
  "Type": "Authorisation",
  "forAssetId": "002",
  "forUserId": "001",
  "Approval": "Y"
}
```

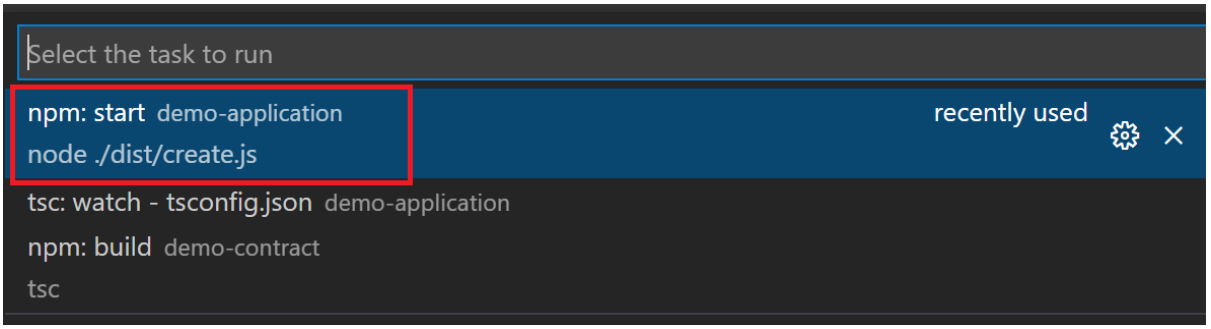
The “AuthCode” is the unique value assigned to every authority granted.
“Type” simply identifies the type of record.
“forAssetId” refers to the unique id assigned to the MC issued by the doctor.
“forUserId” refers to the user whom the MC (the asset) was issued to.
“Approval” is a “Yes” or “No” flag pertaining to this authorisation.

For the external application we will only enter the value:

```
await contract.submitTransaction('grantAuth',
 "003",
 "Authorisation",
 "002",
 "001",
 "Y");
```

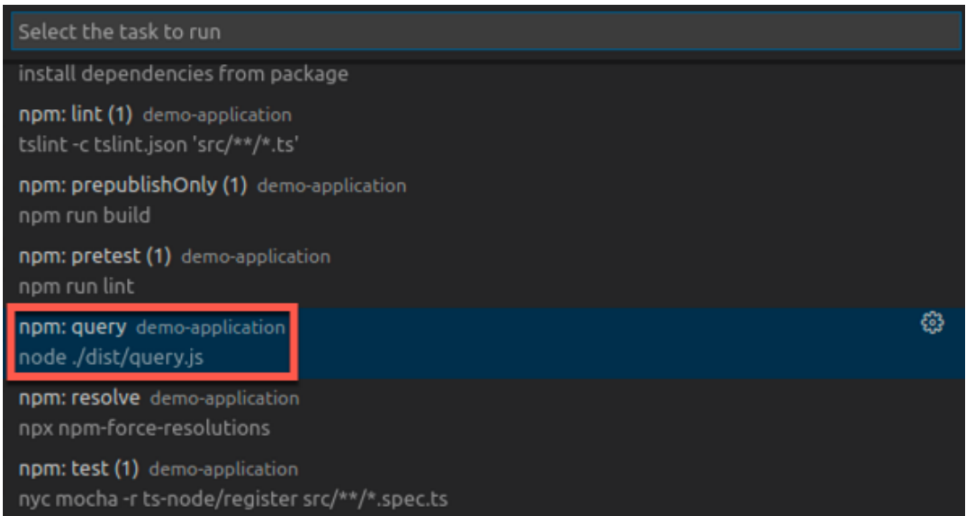


Submit the data by invoking the transaction grantAuth via the terminal option Run Task, “npm: start demo-application” will appear.

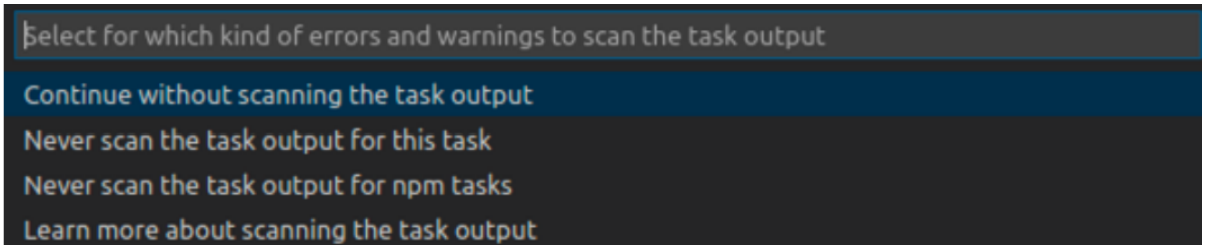


The following message below will appear.

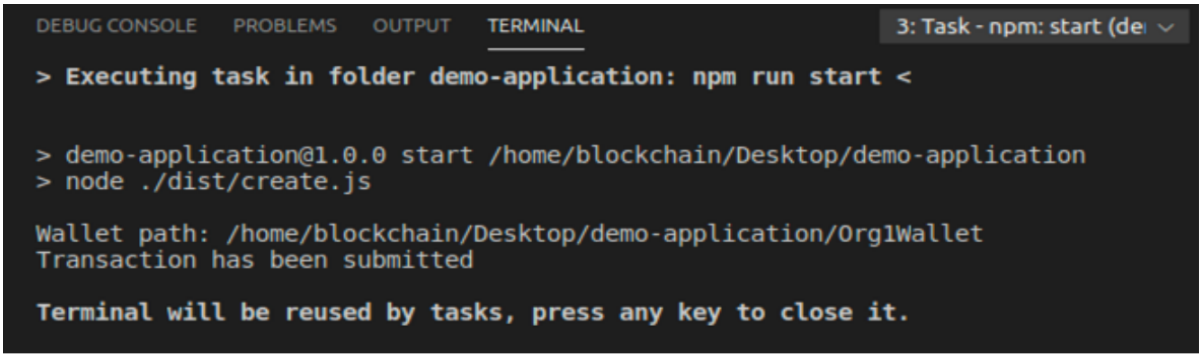
When the query.ts loaded in the editor, find and select the lines that submit the transaction:



Click 'Continue without scanning the task output'.



The task will be run and again, after a brief pause, you will see it complete successfully.



Requestor retrieves MC

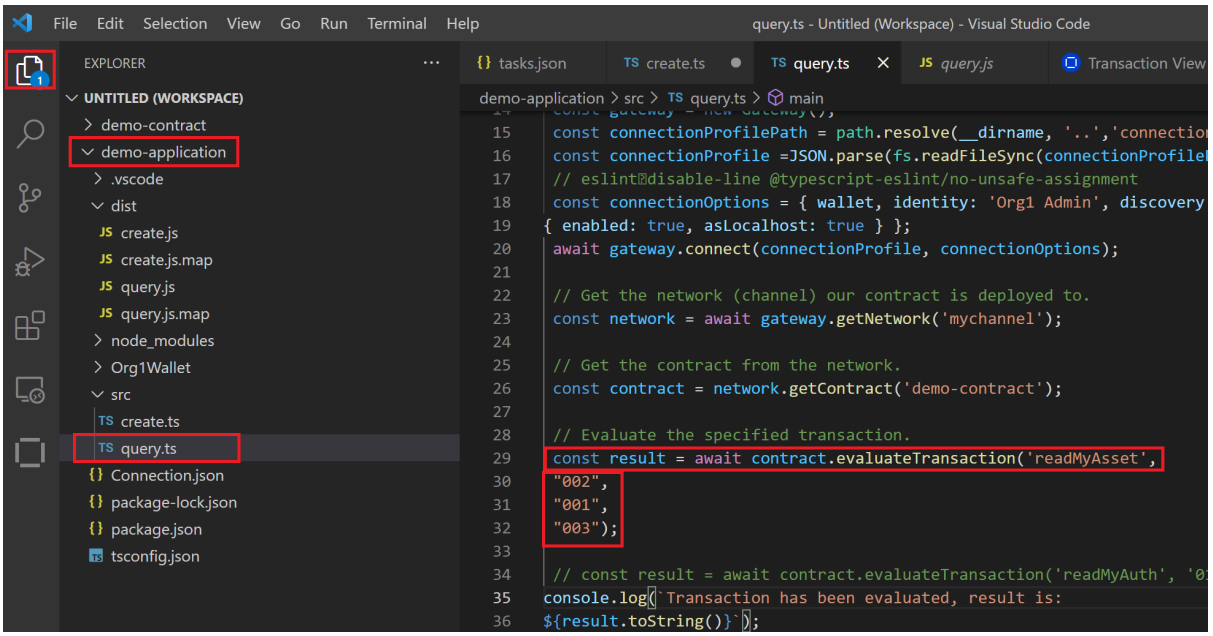
Next we will simulate the employer accessing the employee's MC by submitting a readMyAsset transaction.

```
{
 "AssetId": "002",
 "UserId": "001",
 "AuthCode": "003"
}
```

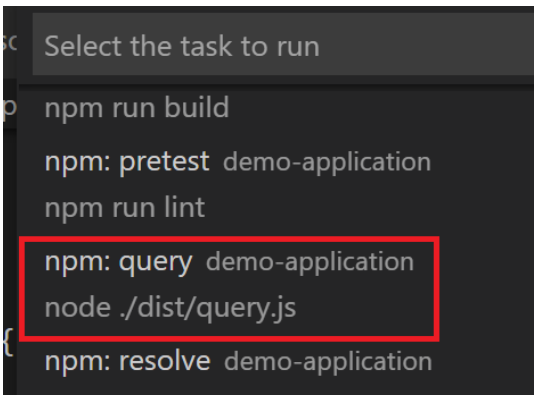
"AssetId" refers to the unique Id assigned to the MC in question.
"UserId" refers to the unique User Id assigned to the requester of the MC, in this, case the employer
"AuthCode" refers to the unique code assigned to authorisation granted by the patient/employee for the request.

However, in the external application we will only enter the value:

```
await contract.submitTransaction('readMyAsset',
 "002",
 "001",
 "003");
```



Run task



The terminal window displays the result of the query, which is the MC issue by the doctor.

```
Transaction has been evaluated, result is:
{"DateIssue":"27/06/2021","DrNo":"2888Z","FrDate":"27/06/2021","Group":"NGHPOLY","ID":"S2222222J","Location":"AMK","MCStatus":"Or
iginal","ToDate":"28/06/2021","Type":"MC"}
```