Tutorial A5: Invoking a smart contract from an external application

---

Estimated time: `45 minutes`

In the last tutorial we learned how an identity, wallet and gateway are used to access a Hyperledger Fabric network. We also used the IBM Blockchain Platform VS Code extension to connect to a sample network and call a smart contract to query the ledger and submit new transactions. In this tutorial we will:

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

In order to successfully complete this tutorial, you must have first completed tutorial A4: Invoking a smart contract from VS Code in the active workspace.

`A5.1`: Expand the first section below to get started.

▶ **Export the network details**

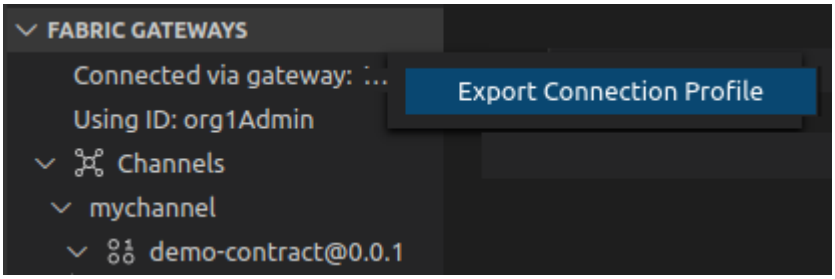
As we have seen, to interact with a Hyperledger Fabric network it is necessary to have:

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

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

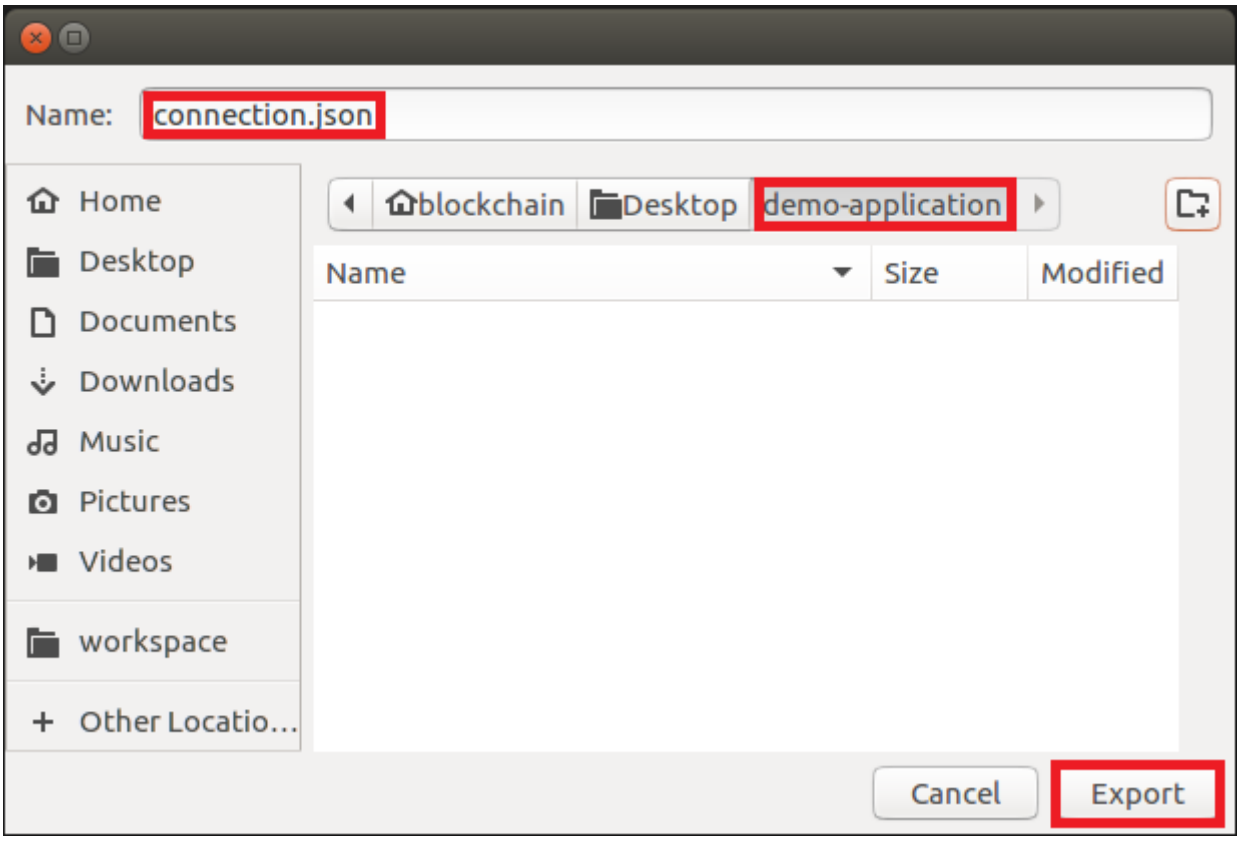
We will start by exporting a *connection profile*.

`A5.2`: With the gateway connected, move the mouse over the Fabric Gateways view, click the ellipsis that appears and select "Export Connection Profile".



`A5.3`: 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

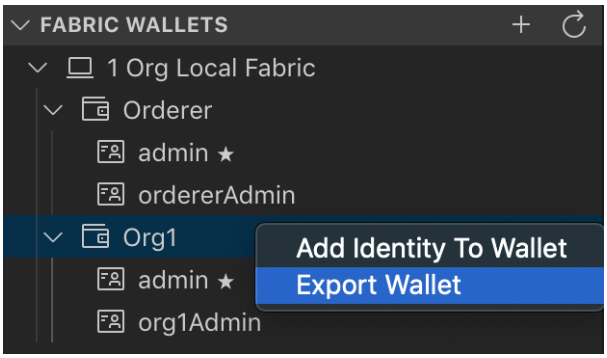
folder.



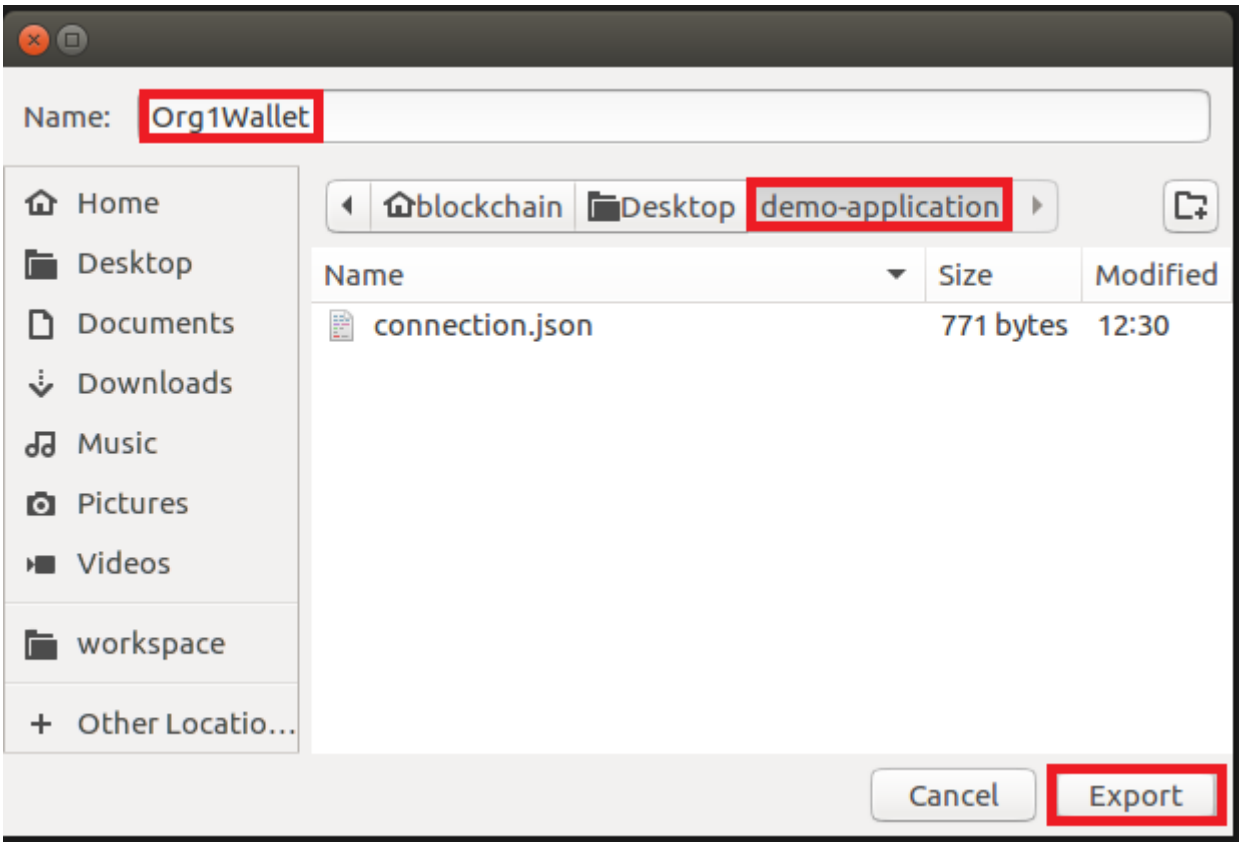
We will now export our wallet.

A5.4: In the Fabric Wallets view, expand '1 Org Local Fabric', right click 'Org1' and select 'Export Wallet'.

Take care not to click on the Orderer organization's wallet by mistake.



A5.5: Navigate into the 'demo-application' folder, change the name to 'Org1Wallet' and click Export to save the wallet.

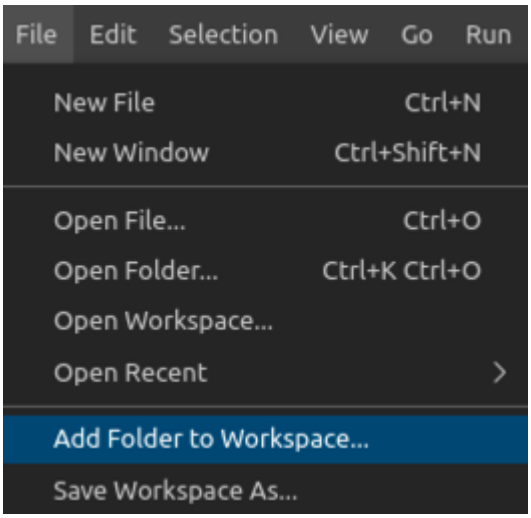


A5.6: Expand the next section of the tutorial to continue.

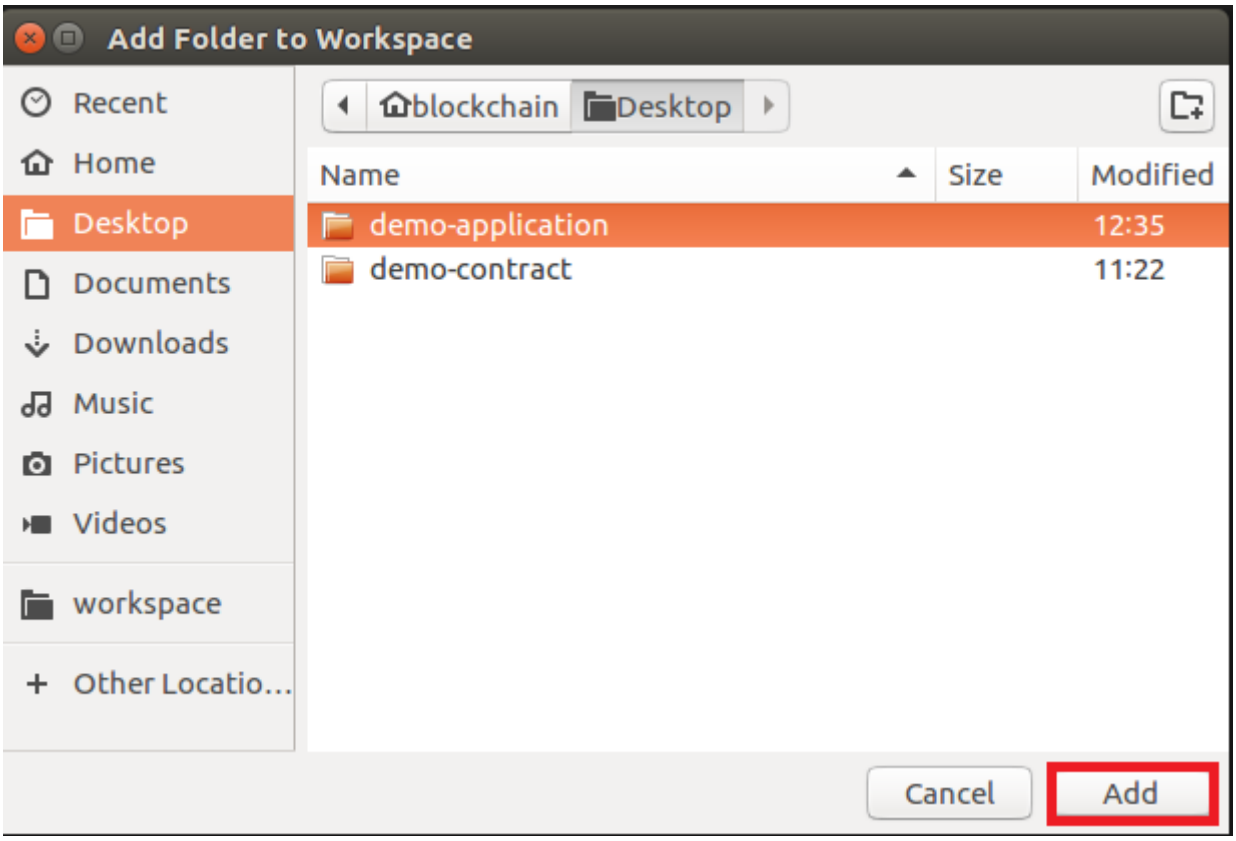
▶ **Create the external application**

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

A5.7: From the VS Code menu bar click "File" -> "Add Folder to Workspace..."

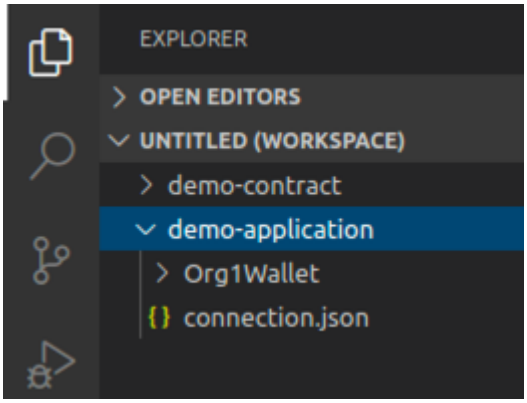


A5.8: Highlight the 'demo-application' folder and click 'Add'.



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'.

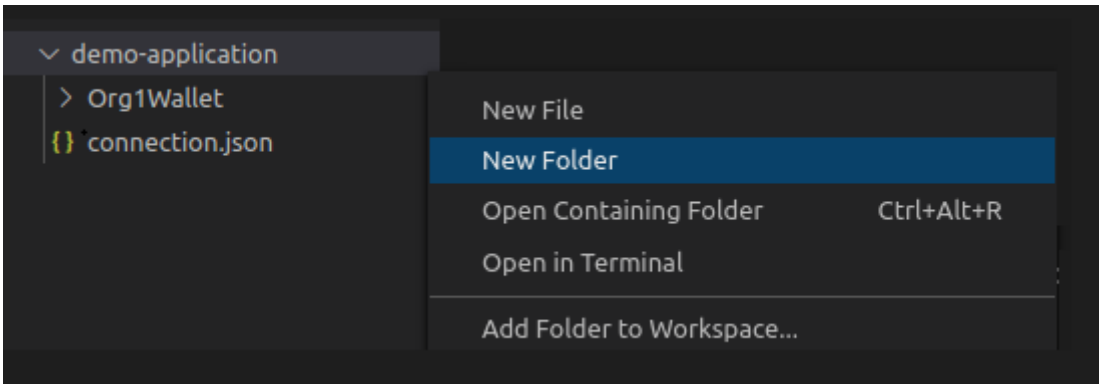


In order to build a working Typescript application we will now create three files in addition to the wallet and connection profile:

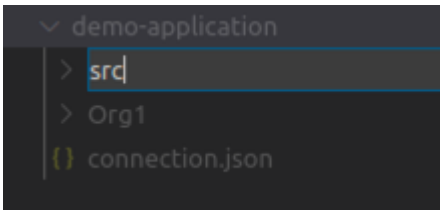
- **create.ts**: The TypeScript application containing the logic required to connect to the Hyperledger Fabric sample network and submit a new transaction.
- **tsconfig.json**: TypeScript compiler options, including source and destination locations
- **package.json**: Application metadata, including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

We will start by creating *create.ts* inside a *src* folder.

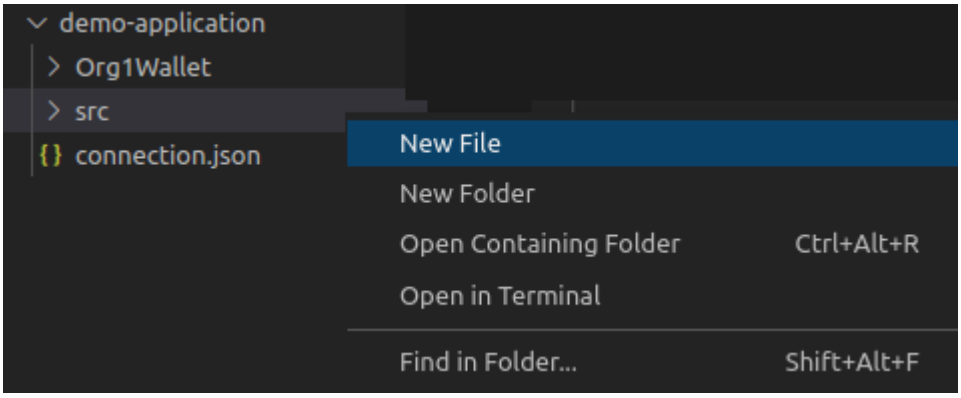
A5.9: Right-click 'demo-application' and select 'New Folder'.



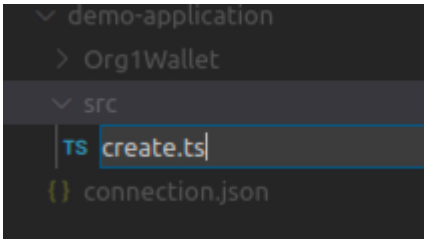
A5.10: Name the folder 'src'.



A5.11: Right-click 'src' and select 'New File'.



A5.12: Name the file '**create.ts**'.



A5.13: In the editor view for the new create.ts file, copy and paste the following text. (The contents are also available here).

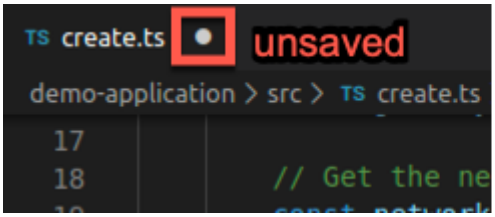
```
import { FileSystemWallet, Gateway } from 'fabric-network';
import * as path from 'path';

async function main() {
    try {

        // Create a new file system based wallet for managing identities.
        const walletPath = path.join(process.cwd(), 'Org1Wallet');
```

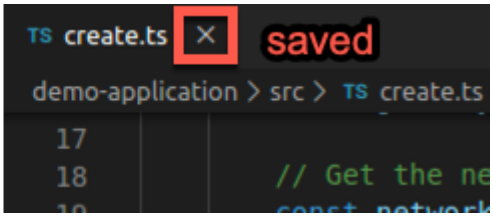
```
        const wallet = new FileSystemWallet(walletPath);
        console.log(`Wallet path: ${walletPath}`);

        // Create a new gateway for connecting to our peer node.
        const gateway = new Gateway();
        const connectionProfile = path.resolve(__dirname, '..',
'connection.json');
        let connectionOptions = { wallet, identity: 'org1Admin', discovery:
{ enabled: true, asLocalhost: true }};
        await gateway.connect(connectionProfile, connectionOptions);

        // Get the network (channel) our contract is deployed to.
        const network = await gateway.getNetwork('mychannel');

        // Get the contract from the network.
        const contract = network.getContract('demo-contract');

        // Submit the specified transaction.
        await contract.submitTransaction('createMyAsset', '002', 'Night
Watch');
        console.log(`Transaction has been submitted`);

        // Disconnect from the gateway.
        await gateway.disconnect();

    } catch (error) {
        console.error(`Failed to submit transaction: ${error}`);
        process.exit(1);
    }
}
main();
```

Your file should be 36 lines long. We will look through what the application is doing later on in this tutorial.

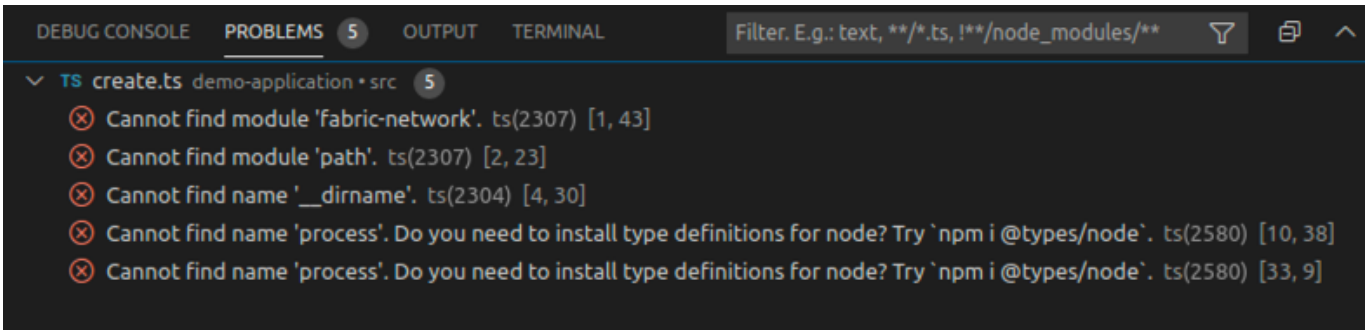
- [ ] A5.14: Save the file ('File' -> 'Save').

Saving the file will change the tab for the editor to show a cross; a solid circle here means that you have unsaved changes:



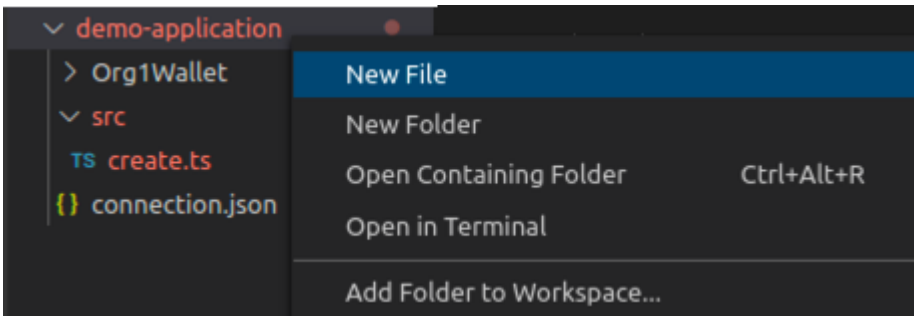


When you save, you will see various errors reported by VS Code. This is because we have not yet configured the set of external dependencies.

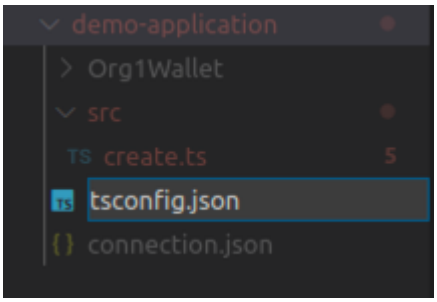


We will next create the *tsconfig.json* file.

A5.15: Right-click 'demo-application' (NOT 'src') and select 'New File'.



A5.16: Name the file '**tsconfig.json**'.



A5.17: In the editor view for the new tsconfig.json file, copy and paste the following text. (The contents are also available here).

```
{
    "compilerOptions": {
      "target": "es6",
      "module": "commonjs",
      "allowJs": true,
      "sourceMap": true,
      "outDir": "./dist/",
      "strict": true,
      "noImplicitAny": true,
      "strictNullChecks": true,
      "strictFunctionTypes": true,
      "strictBindCallApply": true,
      "strictPropertyInitialization": true,
      "noImplicitThis": true,
      "alwaysStrict": true,
      "esModuleInterop": true,
```

```
      "forceConsistentCasingInFileNames": true
    },
    "include": [
      "./src/**/*"
    ],
    "exclude": [
      "node_modules"
    ]
}
```

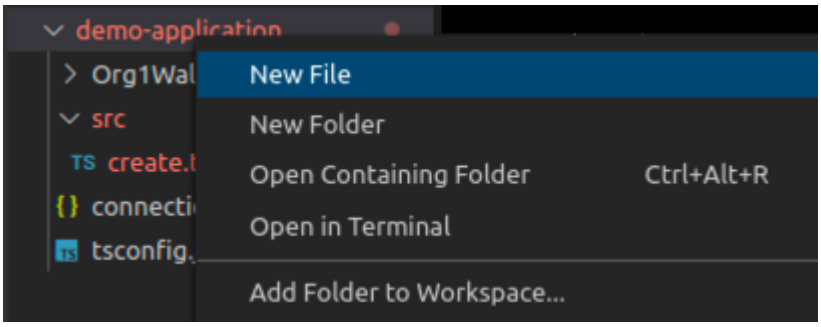
Your file should be 25 lines long.

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.

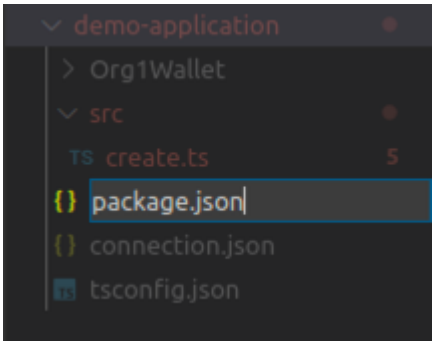
- [ ] A5.18: Save the file ('File' -> 'Save').

Finally, we will create the *package.json* file.

- [ ] A5.19: Right-click 'demo-application' (NOT 'src') and select 'New File'.



- [ ] A5.20: Name the file '**package.json**'.



- [ ] A5.21: In the editor view for the new package.json file, copy and paste the following text. (The contents are also available here).

```
{
  "name": "demo-application",
  "version": "1.0.0",
  "description": "Demo Application implemented in TypeScript",
  "main": "dist/index.js",
  "typings": "dist/index.d.ts",
  "engines": {
```

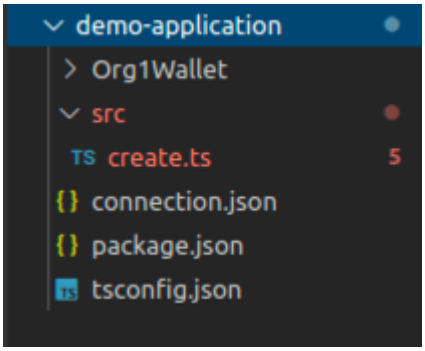
```
    "node": ">=8",
    "npm": ">=5"
  },
  "scripts": {
    "lint": "tslint -c tslint.json 'src/**/*.ts'",
    "pretest": "npm run lint",
    "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
    "resolve": "npx npm-force-resolutions",
    "build": "tsc",
    "build:watch": "tsc -w",
    "prepublishOnly": "npm run build",
    "start": "node ./dist/create.js",
    "create": "node ./dist/create.js",
    "query": "node ./dist/query.js",
    "listener": "node ./dist/listener.js"
  },
  "engineStrict": true,
  "author": "Hyperledger",
  "license": "Apache-2.0",
  "dependencies": {
    "fabric-network": "~1.4.0"
  },
  "devDependencies": {
    "@types/chai": "^4.2.0",
    "@types/mocha": "^5.2.7",
    "@types/node": "^10.12.10",
    "@types/sinon": "^7.0.13",
    "@types/sinon-chai": "^3.2.3",
    "chai": "^4.2.0",
    "chai-as-promised": "^7.1.1",
    "jsrsasign": "^8.0.13",
    "minimist": "^1.2.5",
    "mocha": "^6.2.0",
    "nyc": "^14.1.1",
    "sinon": "^7.4.1",
    "sinon-chai": "^3.3.0",
    "ts-node": "^8.3.0",
    "tslint": "^5.19.0",
    "typescript": "^3.6.2"
  },
  "nyc": {
    "extension": [
      ".ts",
      ".tsx"
    ],
    "exclude": [
      "coverage/**",
      "dist/**"
    ],
    "reporter": [
      "text-summary",
      "html"
    ],
    "all": true,
```

```
    "check-coverage": true,
    "statements": 100,
    "branches": 100,
    "functions": 100,
    "lines": 100
  },
  "resolutions": {
    "minimist": "^1.2.5",
    "mkdirp": "^1.0.4",
    "jsrsasign": "^8.0.13"
  }
}
```

Your file should be 73 lines long. It describes the module dependencies of our application, including any required versions. It also configures a few scripts that we will run during the course of these tutorials.

A5.22: Save the file ('File' -> 'Save').

At this stage, your 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.



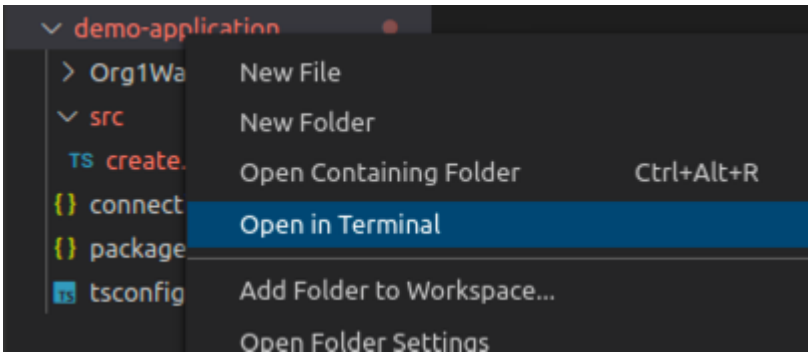
In the next section we will build the application.

A5.23: Expand the next section to continue.

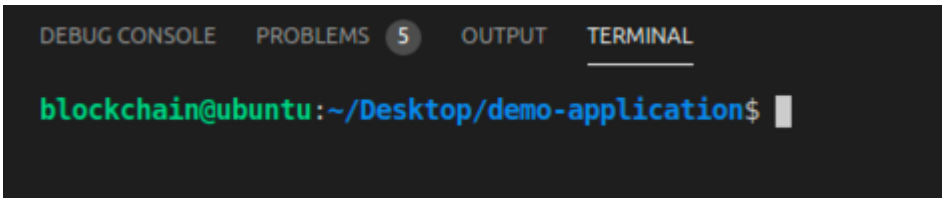
▶ **Build the external application**

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.

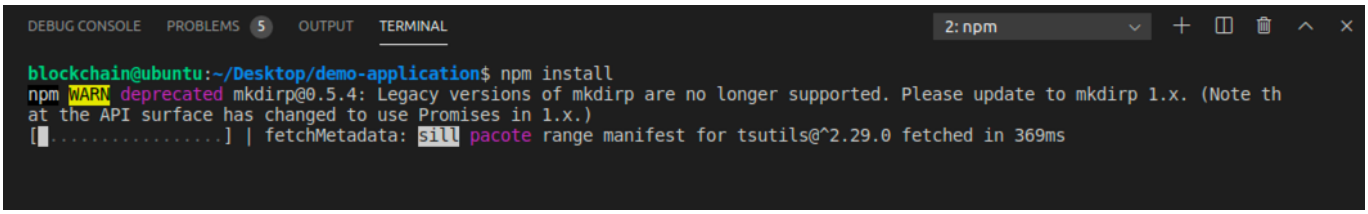
A5.24: Right-click 'demo-application' and select 'Open in Terminal'.



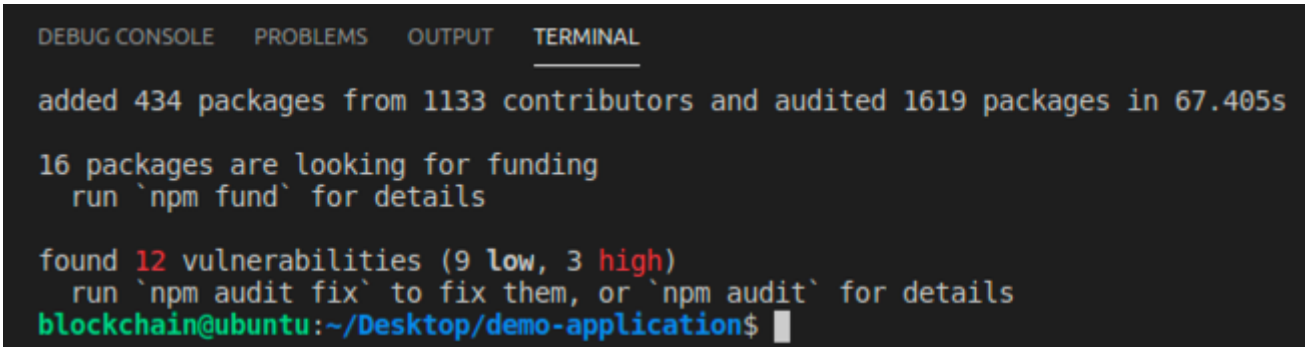
This will bring to focus a terminal prompt inside VS Code.



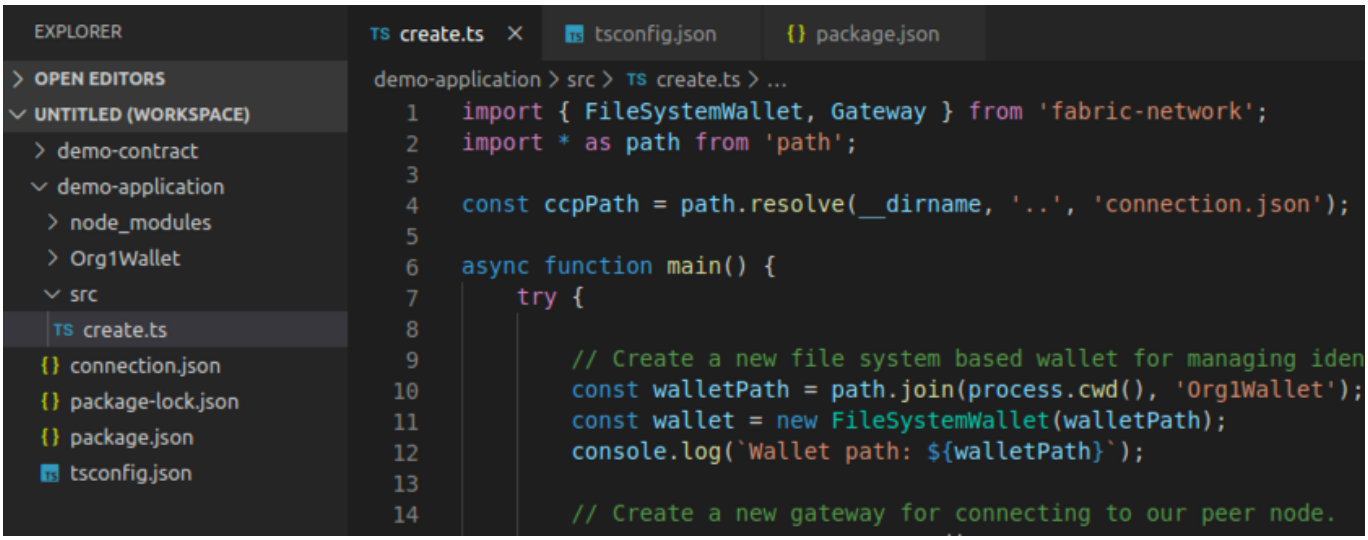
A5.25: In the terminal window type `npm install` and press Enter.



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.

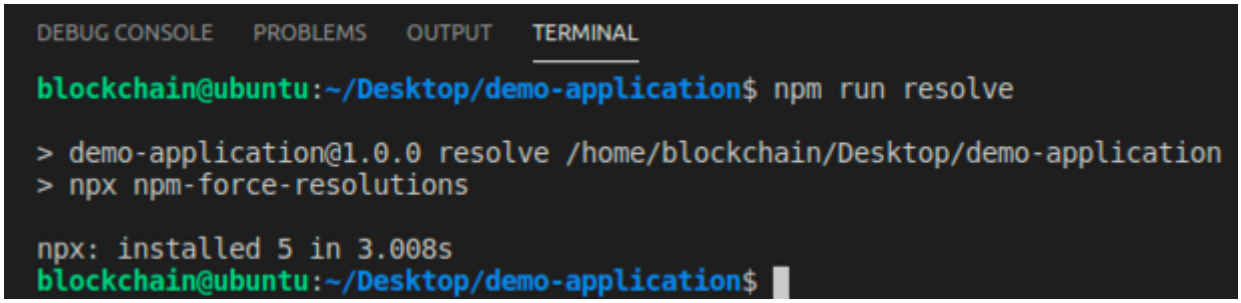


The errors that were previously reported will now all vanish, and the demo-application folder will now contain a new 'node_modules' folder that contains the imported dependencies.



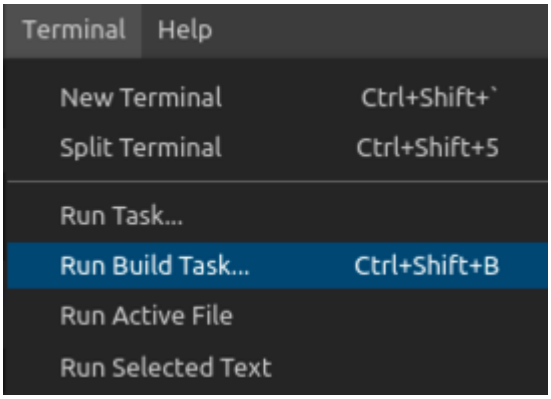
Resolving vulnerabilities

Depending on the versions of your installed components, you might see software vulnerabilities reported in the terminal after installation and it is good practice to fix these. We recommend running the *npm run resolve* command in the same terminal window. This runs the *resolve* script described in our package.json.



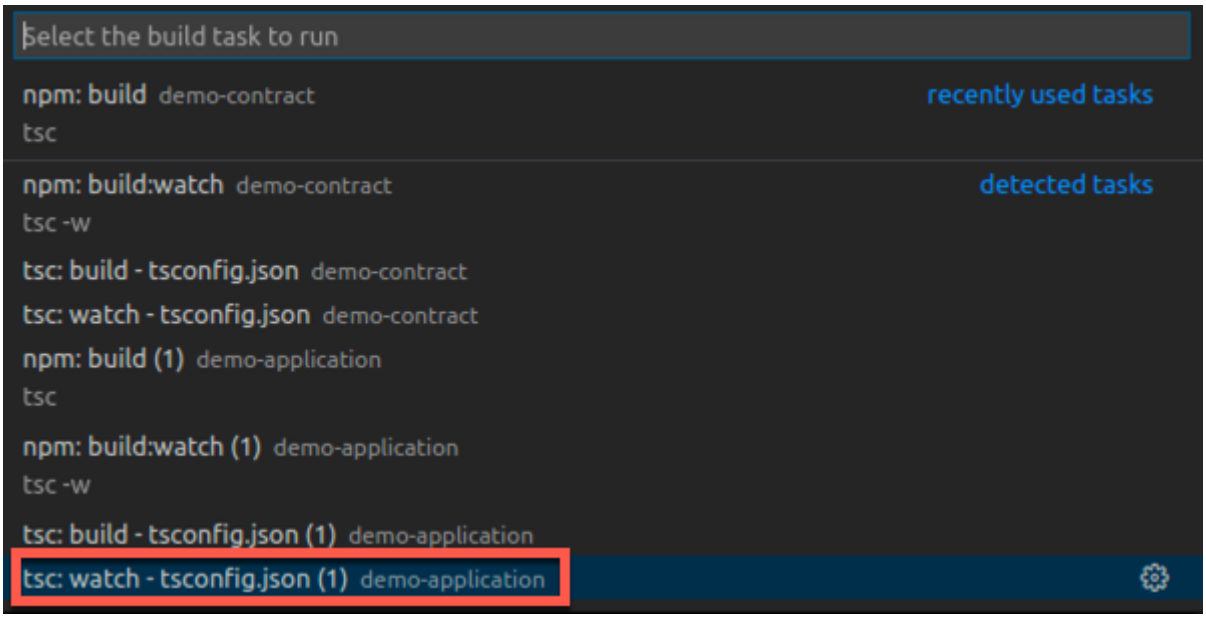
With our dependencies resolved we can now build our application.

- A5.26: In the main VS Code menu, click 'Terminal' -> 'Run Build Task...'.

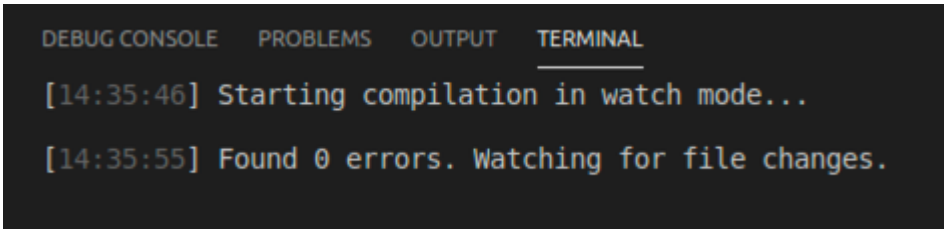


- A5.27: In the command palette, find and click 'tsc: watch - tsconfig.json demo-application'.

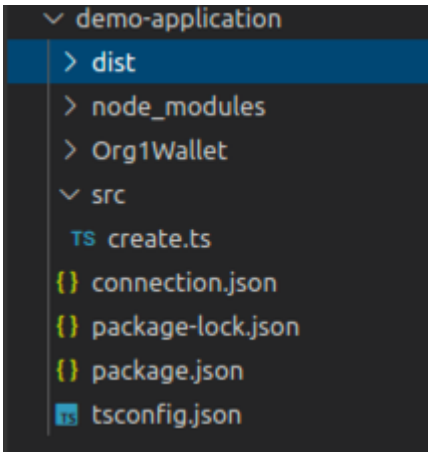
Take care to select the correct option as there will be similar looking alternatives (build options for our smart contract project, for example). You might need to scroll the list to find the correct task.



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.



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.

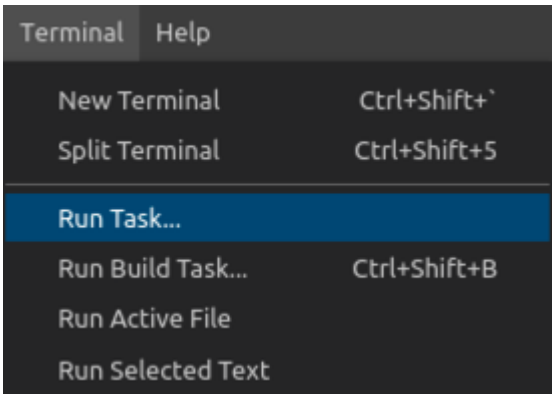


A5.28: Expand the next section to continue.

▶ **Run the external application**

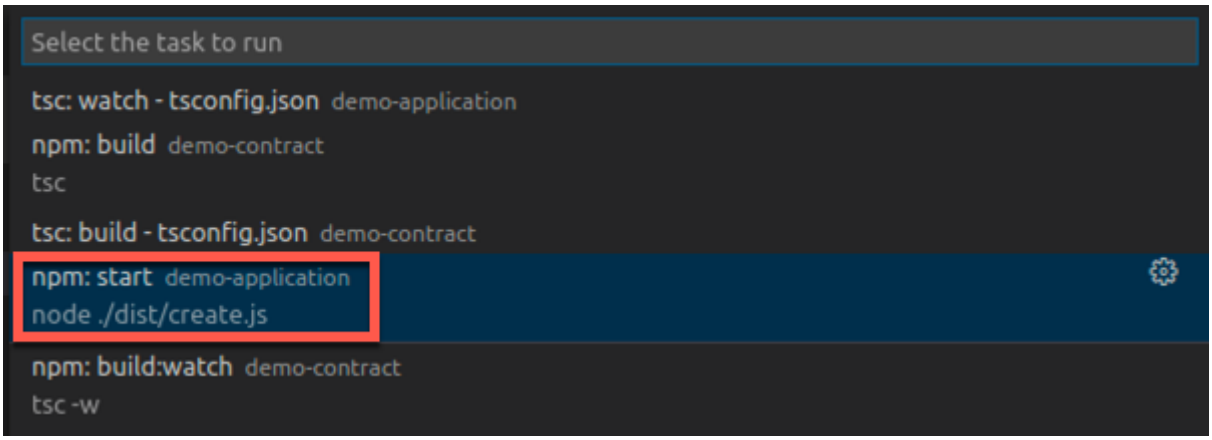
We can run our application wherever we choose - it is just a standard Node.js application. However, VS Code additionally provides an integrated terminal facility whereby different tasks can be run in different terminals. We'll use that now to make all our outputs easily accessible within VS Code.

A5.29: In the main VS Code menu, click 'Terminal' -> 'Run Task...'.



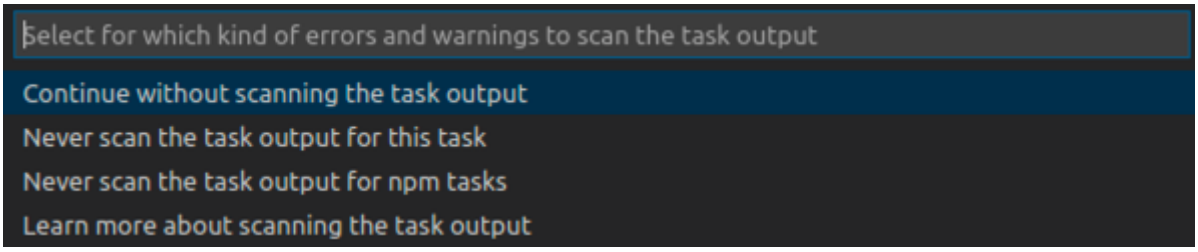
- `A5.30`: Find and select the task 'npm: start demo-application'.

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.



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

- `A5.31`: Click 'Continue without scanning the task output'.

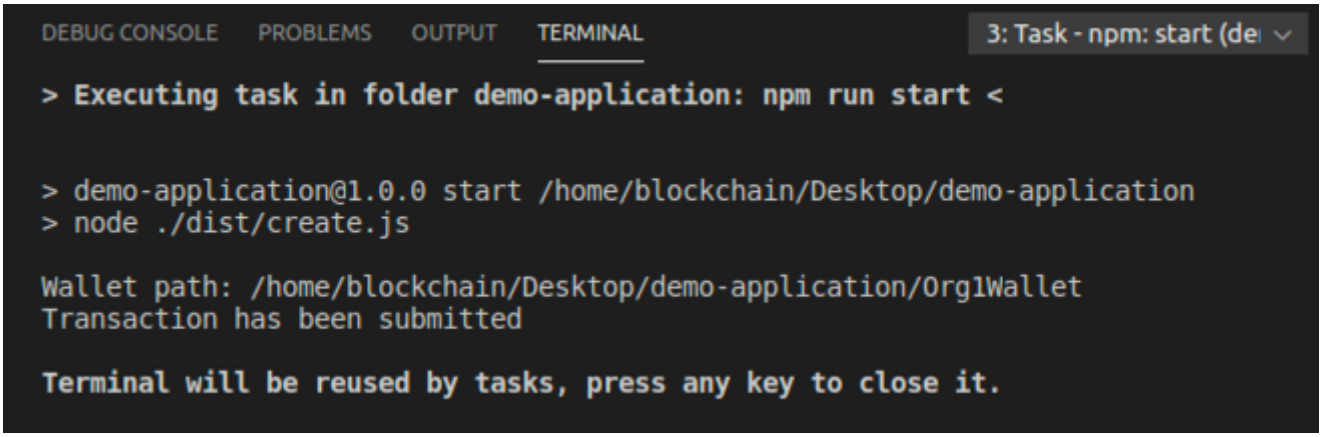


The task will now run. What it will do is run the *start* script that is defined in demo-application's package.json, which is the command `node ./dist/create.js`. You could run the same node command in any appropriately configured environment and achieve the same output.

(Note that the *start* script is identical to *create*; while convention is to name our default script *start*, having *create* also allows us to be more explicit about the application we are running.)

```
"scripts": {
  "lint": "tslint -c tslint.json 'src/**/*.ts'",
  "pretest": "npm run lint",
  "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
  "resolve": "npx npm-force-resolutions",
  "build": "tsc",
  "build:watch": "tsc -w",
  "prepublishOnly": "npm run build",
  "start": "node ./dist/create.js",
  "create": "node ./dist/create.js",
  "query": "node ./dist/query.js",
  "listener": "node ./dist/listener.js"
},
```

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



> **Running the command a second time?**
> If you run the command again you will see errors like 'The my asset 002 already exists' and 'Endorsement has failed'.
> If you review the implementation of the *createMyAsset* transaction, you'll see that this is intentional.
> You can quickly fix this by submitting the appropriate *deleteMyAsset* transaction in the Fabric Gateways view.

- A5.32: 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.

Reviewing the application

At this stage it is worthwhile reviewing what the application actually did.

- A5.33: Click on the 'create.ts' tab in the VS Code editor (or load it from the Explorer view).

You can see the sequence of steps in the source file. The key elements are:

1. The wallet location on the local file system is specified using `FileSystemWallet()`
2. A gateway is connected to using `gateway.connect()`. The gateway uses a set of connection options that control how it interacts with the network channels accessed via the connection profile. A gateway has a fixed set of options which include the identity, and whether or not it uses service discovery for example.
3. The gateway provides access to a set of network channels using `gateway.getNetwork()`.
4. The network channel provides access to a set of smart contracts using `network.getContract()`.
5. A new transaction is submitted using `contract.submitTransaction()`.
6. Control is returned to the application only when consensus is complete and the distributed ledger has been updated.

We will now make a small change to the external application to call a different transaction.

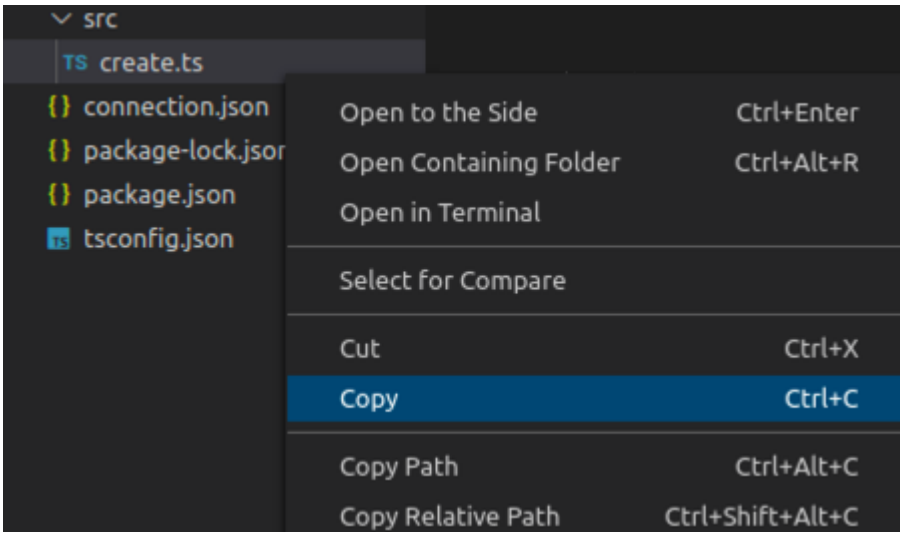
A5.34: Expand the next section of the tutorial to continue.

▶ **Modify the external application**

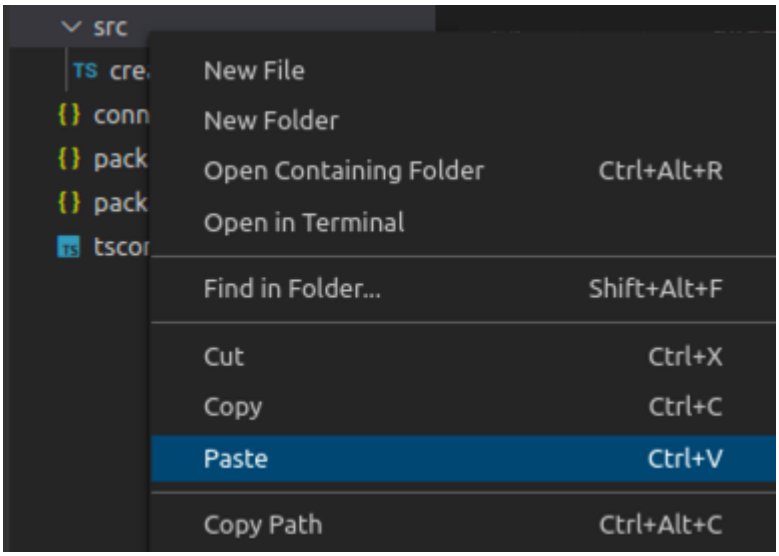
Although submitting a transaction is the most important blockchain application operation, querying the ledger is the most common.

We're now going to query the ledger to verify that the createMyAsset transaction was added to the ledger correctly. To do this, we'll create a new query application based on our existing application.

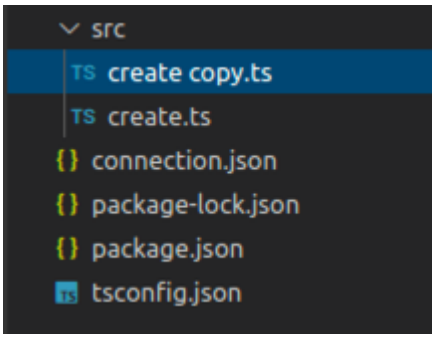
A5.35: In the Explorer view, right click 'create.ts' and select 'Copy'.



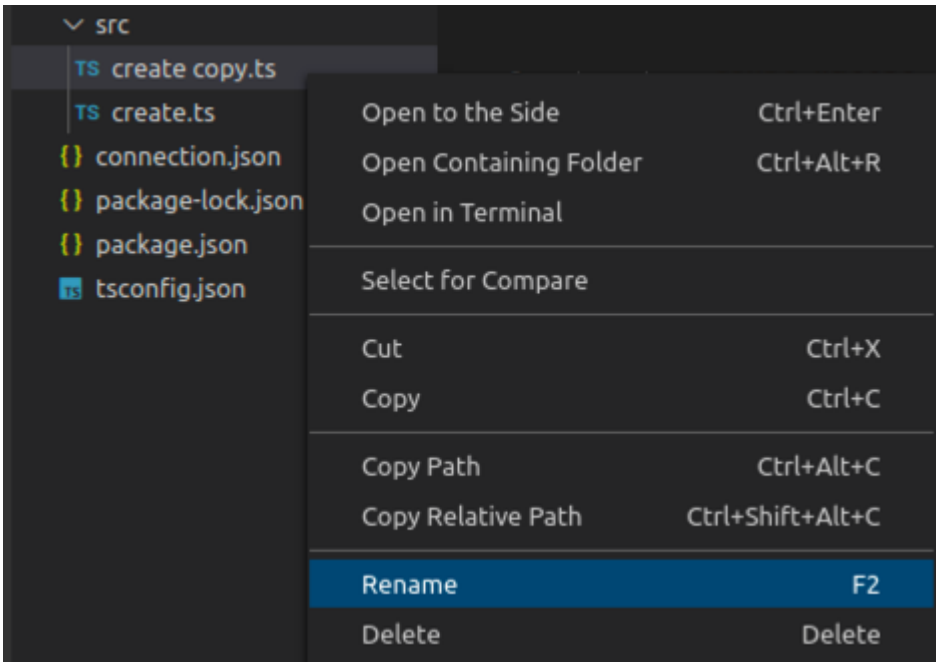
A5.36: Select the 'src' folder, right click and select 'Paste'.



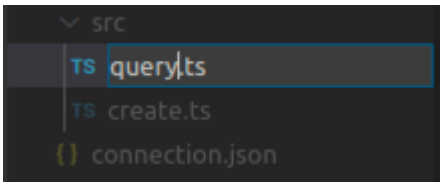
This will create a new file inside the 'src' folder called 'create copy.ts'.



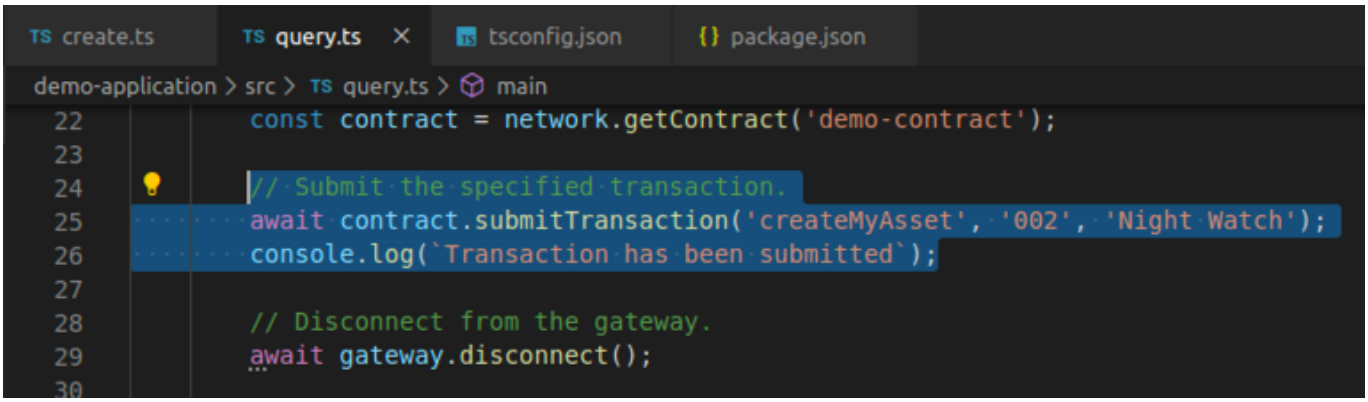
A5.37: Right click the new file and select 'Rename'.



A5.38: Rename the file to 'query.ts', and press Enter to confirm.



A5.39: With query.ts loaded in the editor, find and select the lines that submit the transaction:



Our new application will attempt to evaluate the readMyAsset transaction and display the result.

A5.40: Use copy and paste to replace these lines with the following:

```
// Evaluate the specified transaction.
const result = await contract.evaluateTransaction('readMyAsset', '002');
console.log(`Transaction has been evaluated, result is:
${result.toString()}`);
```

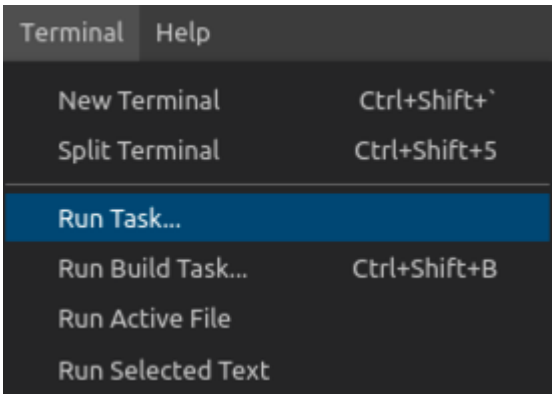
A5.41: Save the file ('File' -> 'Save').

As we previously enabled watch mode in the compiler, saving the file will automatically cause a recompilation. This will take a few seconds.

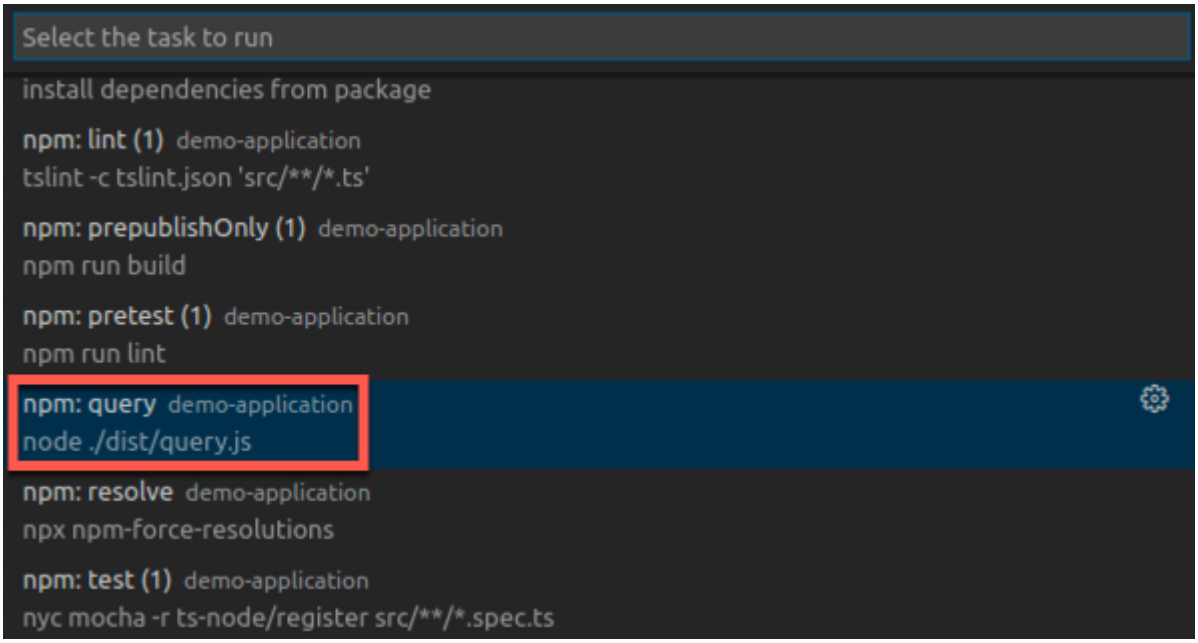
We can test the new application by running the correct task in npm. To save time, we created the query task when we implemented the package.json file earlier:

```
"scripts": {
  "lint": "tslint -c tslint.json 'src/**/*.ts'",
  "pretest": "npm run lint",
  "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
  "resolve": "npx npm-force-resolutions",
  "build": "tsc",
  "build:watch": "tsc -w",
  "prepublishOnly": "npm run build",
  "start": "node ./dist/create.js",
  "create": "node ./dist/create.js",
  "query": "node ./dist/query.js",
  "listener": "node ./dist/listener.js"
},
```

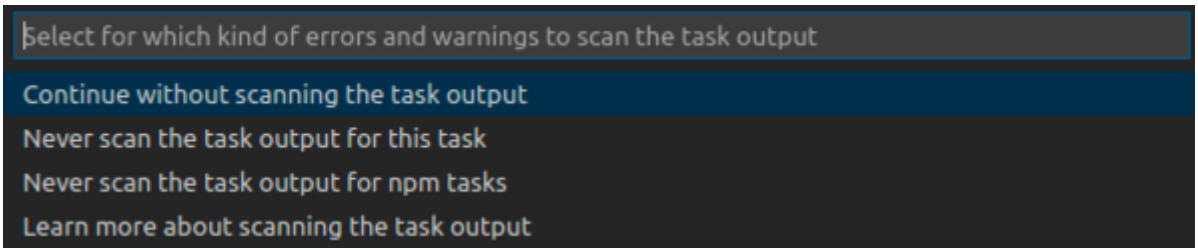
- A5.42: In the main VS Code menu, click 'Terminal' -> 'Run Task...'.



- A5.43: Find and select the task 'npm: query demo-application'.



- A5.44: Click 'Continue without scanning the task output'.



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

```
> demo-application@1.0.0 query /home/blockchain/Desktop/demo-application
> node ./dist/query.js

Wallet path: /home/blockchain/Desktop/demo-application/Org1Wallet
Transaction has been evaluated, result is: {"value":"Night Watch"}

Terminal will be reused by tasks, press any key to close it.
```

Note that the output now shows the current value of the '002' key we entered earlier.

Congratulations! You've now written two applications: one that submits new transactions to the ledger and another that queries the current value of the ledger.

Summary

In this tutorial we have built client applications that can submit and evaluate transactions on a Hyperledger Fabric blockchain.

Although the Fabric instance we've been working with is part of our VS Code environment, these applications can connect to any Hyperledger Fabric blockchain: you simply use the identity and connection profile for the target network when you create a gateway object you instantiate in your application.

In the same way as applications can change, smart contracts can evolve too. In the next tutorial we will try modifying our smart contract to see how the upgrade process works.

→ **A6: Upgrading a smart contract**