Tutorial A9: Publishing an event

Estimated time: `30 minutes`

In Hyperledger Fabric, smart contracts can generate events that can be received by applications that have registered for event notifications. In this tutorial we will:

- Update a smart contract to emit an event
- Subscribe to an event notification in the IBM Blockchain Platform VS Code extension
- Submit a transaction and receive the resulting event notification
- Subscribe to the same event notification in a client application

In order to successfully complete this tutorial, you must have first completed tutorial A5: Invoking a smart contract from an external application in the active workspace. It is desirable to have completed up to tutorial A8: Testing a smart contract.

⬚   `A9.1`:   Expand the first section below to get started.

▶ **Implement the event logic**

So far in this tutorial series, applications that submit new transactions are closely coupled to applications that query the ledger. For example, one application has to submit a transaction and let consensus complete before another application can query the ledger to confirm the result of that transaction.

An alternative pattern to couple applications connected to a Hyperledger Fabric network is using *events* and *notifications*. Applications can call smart contracts which generate transactions containing an event, and when that transaction is successfully committed to the ledger, a notification can be received by one or more applications that have registered for event notification. This is a loosely coupled paradigm; the event consumer is unaware of the event producer.

An event is used to describe any significant situation that will be of interest to one or more applications; for example, when an update to an asset has occurred. Indeed, it can be appropriate to use this style of application coupling in many different scenarios:

- a regulator might want to be notified of a trade in near real time
- a supplier might want to use the acceptance of a new customer order to start a new business process to fulfill their contractual agreements
- a stock management system might want to keep an up-to-the-second accurate count of a particular transaction type in order to manage internal inventory

A transaction generated by a smart contract can include an event. When the transaction is committed to the ledger, all applications that are connected a suitable peer will receive a notification of this event. This is the publish/subscribe metaphor; a smart contract generates an event that can be consumed by zero or more applications. Applications can come and go; the event consumer and event producer are said to be loosely coupled to each other.

As well as smart contract events, Hyperledger Fabric components can also publish system events when interesting things occur, such as when a block is committed.

Modifying the smart contract

Before we can register for event notifications, we first need a smart contract with a transaction containing an event. We will use the 'createMyAsset' transaction in the smart contract we previously created to do this.

▢   A9.2:   Switch to the 'my-asset-contract.ts' file in the editor.

If the file is not already open, use the Explorer side bar to navigate to 'demo-contract' -> 'src' -> 'my-asset-contract.ts'.

▢   A9.3:   Navigate to the 'createMyAsset' method and use copy and paste to insert the following two lines at the end of the transaction's implementation:

```
const eventPayload: Buffer = Buffer.from(`Created asset ${myAssetId}
(${value})`);
ctx.stub.setEvent('myEvent', eventPayload);
```

The updated method should look like this:

```
17
18      @Transaction()
19      public async createMyAsset(ctx: Context, myAssetId: string, value: string): Promise<void> {
20          const exists = await this.myAssetExists(ctx, myAssetId);
21          if (exists) {
22              throw new Error(`The my asset ${myAssetId} already exists`);
23          }
24          const myAsset = new MyAsset();
25          myAsset.value = value;
26          const buffer = Buffer.from(JSON.stringify(myAsset));
27          await ctx.stub.putState(myAssetId, buffer);
28          const eventPayload: Buffer = Buffer.from(`Created asset ${myAssetId} (${value})`);
29          ctx.stub.setEvent('myEvent', eventPayload);
30      }
31
```

▢   A9.4:   Save the file ('File' -> 'Save').

The setEvent method takes two parameters: in our example, we see an event name (the string "myEvent") and the event payload (a buffer containing the text "Created asset" and some details of the asset. We'll refer to this information again a little later.
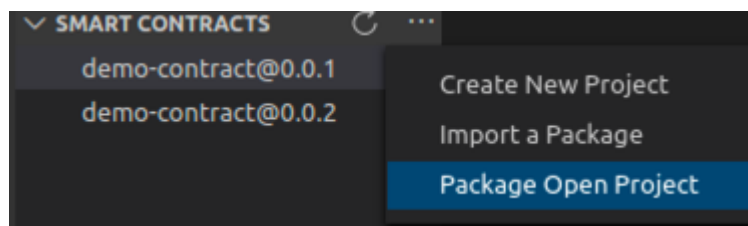
Upgrading the smart contract

We now need to upgrade our smart contract package to include the event emission logic.

☐ **A9.5:** Switch to the 'demo-contract' -> 'package.json' editor and update the value of the "version" field to `"0.0.3"`.

```
demo-contract > {} package.json > ⊞ version
  1    {
  2        "name": "demo-contract",
  3        "version": "0.0.3",
  4        "description": "My Smart Contract",
  5        "main": "dist/index.js",
  6        "typings": "dist/index.d.ts",
  7        "engines": {
  8          "node": ">=8"
```
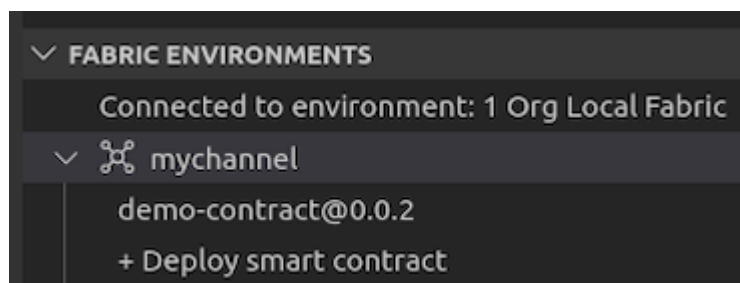
☐ **A9.6:** Save the file ('File' -> 'Save').

☐ **A9.7:** Hover over the Smart Contracts view in the IBM Blockchain Platform side bar, click the ellipsis (...) and select 'Package Open Project' for the 'demo-contract' project.

```
∨ SMART CONTRACTS          ⟳  ···
    demo-contract@0.0.1        Create New Project
    demo-contract@0.0.2        Import a Package
                               Package Open Project
```

Wait a few seconds for the v0.0.3 smart contract to be built and shown in the Smart Contracts view.

```
∨ SMART CONTRACTS
    demo-contract@0.0.1
    demo-contract@0.0.2
    demo-contract@0.0.3
```
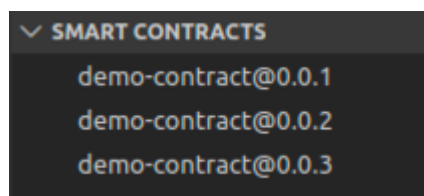
☐ **A9.8:** In the Fabric Environments view, expand "mychannel" and click "+ Deploy smart contract".

```
∨ FABRIC ENVIRONMENTS
    Connected to environment: 1 Org Local Fabric
  ∨ ⛓ mychannel
      demo-contract@0.0.2
      + Deploy smart contract
```

☐ **A9.9:** In the Deploy Smart Contract form Step 1, select "demo-contract@0.0.3" from the drop down list, and click 'Next'.

☐ **A9.10:** In step 2 of the Deploy Smart Contract form, default values for Definition name and version of the updated contract are provided, click 'Next'.

☐ `A9.11`:   In step 3 of the form, click 'Deploy' to start the deployment.

You may need to wait a minute or so for the deployment of the upgraded smart contract to complete.

The upgraded version of the smart contract will be displayed in the Fabric Environments view under mychannel.
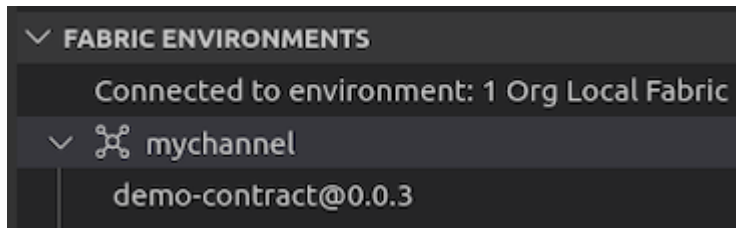
The upgraded smart contract is now ready to use.

☐ `A9.12`:   Expand the next section of the tutorial to continue.


▶ **Subscribe to the topic from IBM Blockchain Platform**

With this change to the smart contract, every time a 'createMyAsset' transaction is committed, an event notification will sent to all applications registered for this event.

Smart contract notifications are only generated when a transaction containing an event is valid; an invalid transaction will not result in an event notification of the event within the transaction. Also, read-only transactions cannot generate events.

Any authorized client application can request event notification. It is possible request event notification within the IBM Blockchain Platform VS Code extension. Let's do that now.

☐ `A9.13`:   In the Fabric Gateways view, ensure that the local gateway is connected.

If the gateway is disconnected, click on "1 Org Local Fabric - Org1" in this view to connect, and select 'org1Admin' as the identity.
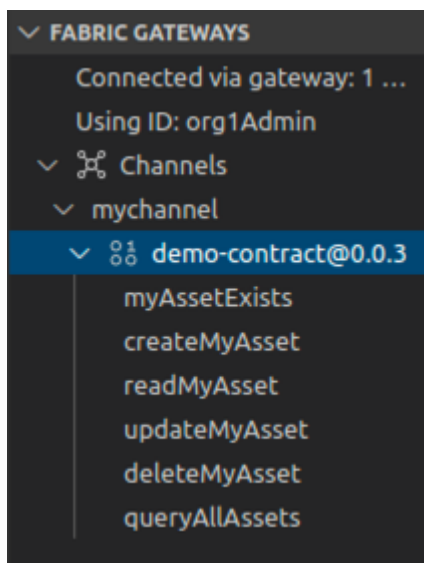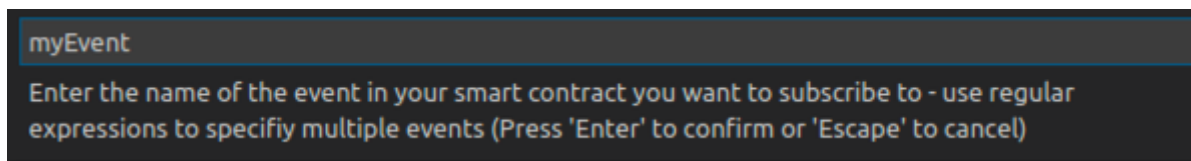

☐ `A9.14`:   Right-click 'demo-contract@0.0.3' and select 'Subscribe to Events'.
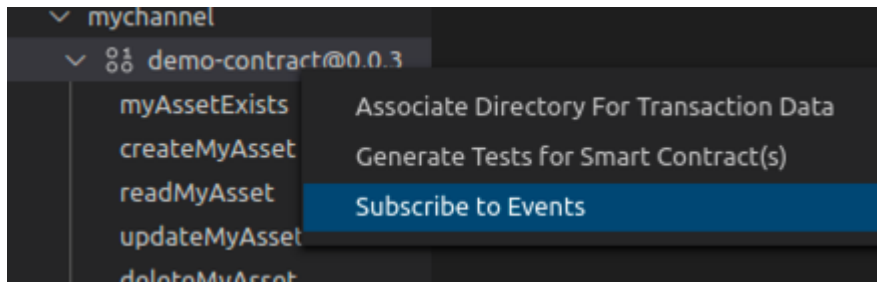
We need to specify which event(s) we are interested in. As you will recall from our event emission code, we named our topic 'myEvent'.

☐ **A9.15**: Type `myEvent` and press Enter.

You will see a notification that confirms that the subscription has been registered.

> **Subscribing to multiple topics:**
> Regular expressions can be used to subscribe to multiple topics at once. For example, entering .* will subscribe to all custom events emitted from the smart contract.

☐ **A9.16**: Expand the next section of the tutorial to continue.

▶ **Receive an event notification**

Now that we have successfully subscribed to the 'myEvent' topic, we'll be able to receive notifications of those events in the VS Code output console.

In order to generate an event, we'll need to generate a createMyAsset transaction.

☐ **A9.17**: In the Fabric Gateways view, right-click the 'createMyAsset' transaction and select 'Submit Transaction'.
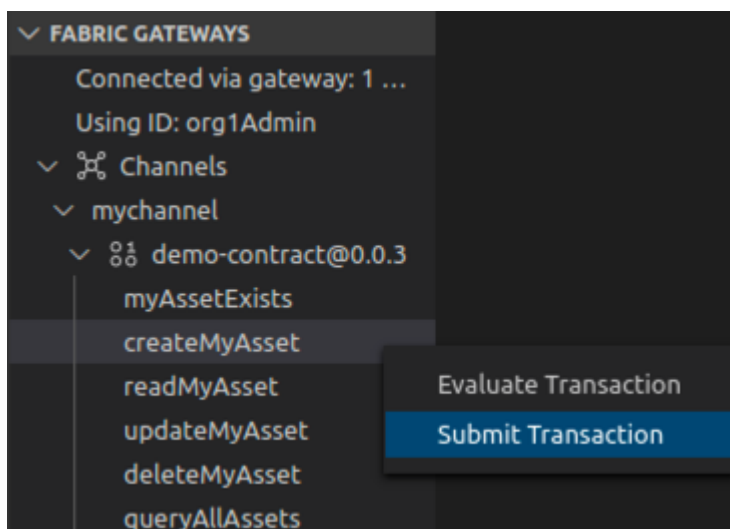
☐ **A9.18:** Replace the input parameters with `["004","Dogs Playing Poker"]` and press Enter.

```
["004","Dogs Playing Poker"]

optional: What are the arguments to the transaction, (e.g. ["arg1", "arg2"]) (Press 'Enter' to
confirm or 'Escape' to cancel)
```

☐ **A9.19:** Press Enter a second time to accept the transient data defaults and submit the transaction.

```
{}

optional: What is the transient data for the transaction, e.g. {"key": "value"} (Press 'Enter' to
confirm or 'Escape' to cancel)
```

☐ **A9.20:** Review the output of this transaction.

The Output panel will show not only the transaction output, but also information about the event that was emitted.

```
[              ] [INFO] submitting transaction createMyAsset with args 004,Dogs Playing
Poker on channel mychannel
[              ] [SUCCESS] No value returned from createMyAsset
[              ] [INFO] Event emitted: chaincode_id: demo-contract, tx_id:
023660e94a41a7a0c4c7556d6bf3bdf1b9b2b970899751c69d624ad2616eb0d8, event_name: "myEvent",
payload: Created asset 004 (Dogs Playing Poker)
```

This information is only displayed because of the active subscription. The subscription will persist until the gateway is disconnected.


▶ **Subscribe to the topic from a client application**

We will now try and subscribe to the same topic from within a client application. We will create a *listener.ts* client application that will be part of the project we created in tutorial A5: Invoking a smart contract from an external application.

☐ **A9.21:** Switch to the Explorer view, and right click the 'src' folder underneath 'demo-application'. Select 'New File'.
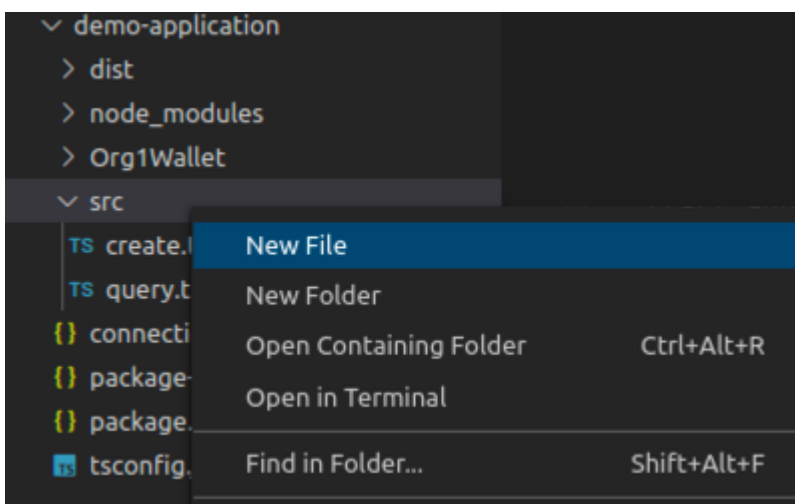
☐ **A9.22:** Name the file 'listener.ts' and press Enter to load the new file in the editor.
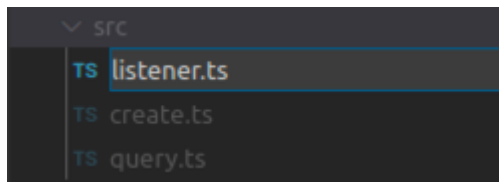
A9.23:   Copy and paste the following source code into the new file. (It is also available here.)

```typescript
import { Gateway, Wallets, ContractListener } from 'fabric-network';
import * as path from 'path';
import * as fs from 'fs';

async function main() {
    try {

        // Create a new file system based wallet for managing identities.
        const walletPath = path.join(process.cwd(), 'Org1Wallet');
        const wallet = await Wallets.newFileSystemWallet(walletPath);
        console.log(`Wallet path: ${walletPath}`);

        // Create a new gateway for connecting to our peer node.
        const gateway = new Gateway();
        const connectionProfilePath = path.resolve(__dirname, '..',
'connection.json');
        const connectionProfile =
JSON.parse(fs.readFileSync(connectionProfilePath, 'utf8'));
        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');

        // Listen for myEvent publications
        const listener: ContractListener = async (event) => {
            if (event.eventName === 'myEvent') {
                const eventString: string = `chaincodeId:
${event.chaincodeId}, event_name: "${event.eventName}", payload:
${event.payload.toString()}`;
                console.log(`Event caught: ${eventString}`);
            }
        };
        contract.addContractListener(listener);

        console.log(`Listening for myEvent events...`);
        while (true) {
            await sleep(5000);
            // ... do other things
        }
```

```
        } catch (error) {
            console.error(`Error: ${error}`);
            process.exit(1);
        }
    }

    function sleep(ms: number) {
        return new Promise(resolve => setTimeout(resolve, ms));
    }

    main();
```

Your source file should be 51 lines long.

⬜ A9.24:   Save the file ('File' -> 'Save').

If you completed tutorial A5: Invoking a smart contract from an external application in the same VS Code session, the compiler will be in watch mode, and the application will now rebuild itself and you can skip to A9.26.

⬜ A9.25:   If necessary, rebuild the client application by clicking 'Terminal' -> 'Run Build Task' and selecting 'tsc: watch - tsconfig.json demo-application'.
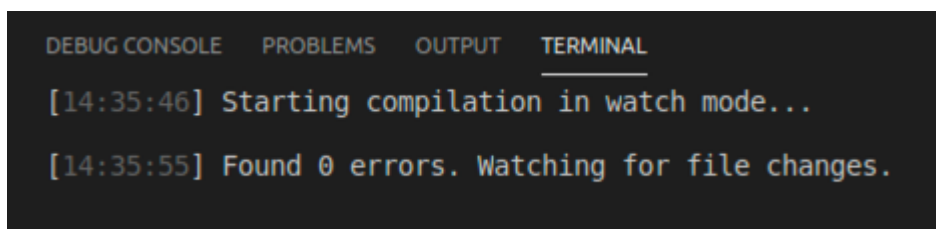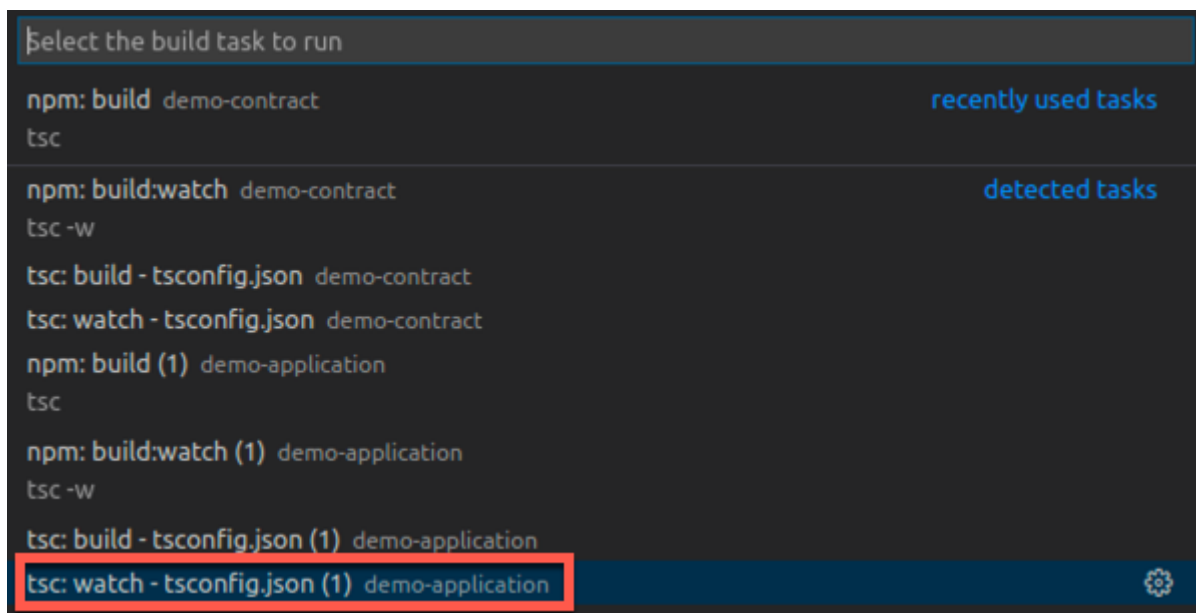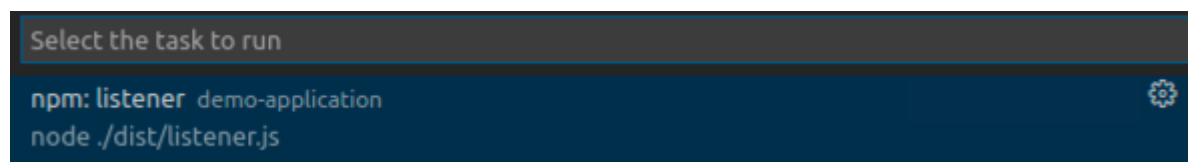

We will now run the application using an npm script that we created in tutorial A5: Invoking a smart contract from an external application. When we created our package.json file in this tutorial, we included an npm script called 'listener' that will run our new application:
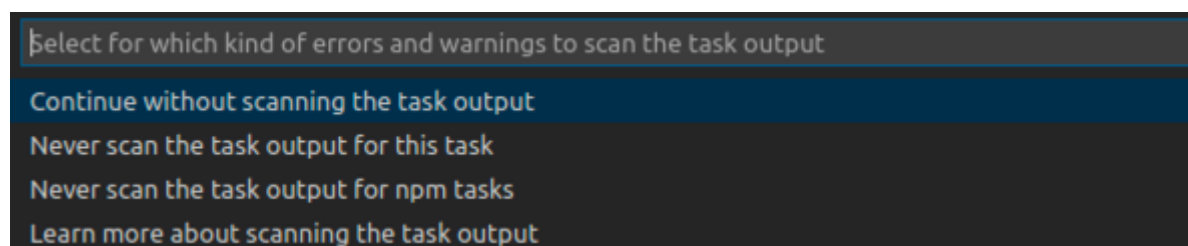
```
"scripts": {
  "lint": "tslint -c tslint.json 'src/**/*.ts'",
  "pretest": "npm run lint",
  "test": "nyc mocha -r ts-node/register src/**/*.spec.ts",
  "resolve": "npx npm-force-resolutions",
  "build": "tsc",
  "build:watch": "tsc -w",
  "prepublishOnly": "npm run build",
  "start": "node ./dist/create.js",
  "create": "node ./dist/create.js",
  "query": "node ./dist/query.js",
  "listener": "node ./dist/listener.js"
},
```

▢ A9.26:   Click 'Terminal' -> 'Run Task', and find and select 'npm: listener demo-application'.
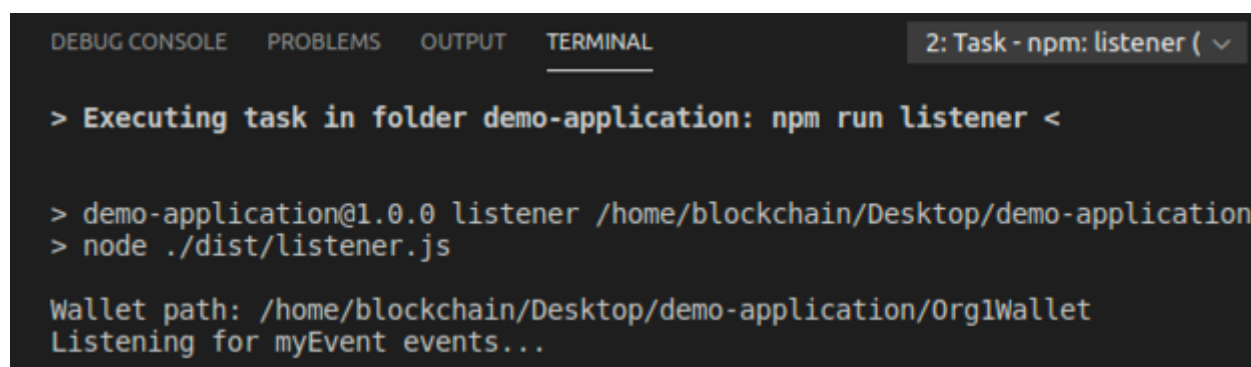
As always, take care in selecting the correct task as the options can look alike.

```
Select the task to run

npm: listener  demo-application                                          ⚙
node ./dist/listener.js
```

▢ A9.27:   As before, select the option to continue without scanning the output.

```
Select for which kind of errors and warnings to scan the task output

Continue without scanning the task output
Never scan the task output for this task
Never scan the task output for npm tasks
Learn more about scanning the task output
```

The client application will now run: the application will connect to the gateway, register for 'myEvent' notifications and then pause indefinitely while it waits to receive them.

```
DEBUG CONSOLE    PROBLEMS    OUTPUT    TERMINAL              2: Task - npm: listener ( ⌄

 > Executing task in folder demo-application: npm run listener <


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

Wallet path: /home/blockchain/Desktop/demo-application/Org1Wallet
Listening for myEvent events...
```

Note that we now have *two* subscribers: the IBM Blockchain Platform extension subscriber that we configured earlier, and our new client application. We will now test that notifications can be received by both of them.

▢ A9.28:   Switch to the IBM Blockchain Platform sidebar, and in the Fabric Gateways view right click the 'createMyAsset' transaction and select 'Submit transaction'.
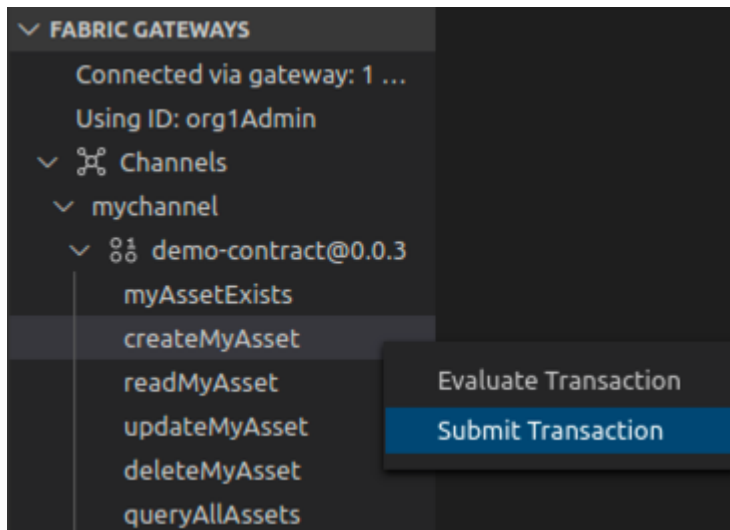
☐ **A9.29:** Replace the input parameters with `["005","The Starry Night"]`. As before, press Enter to confirm and Enter again to skip transient data.
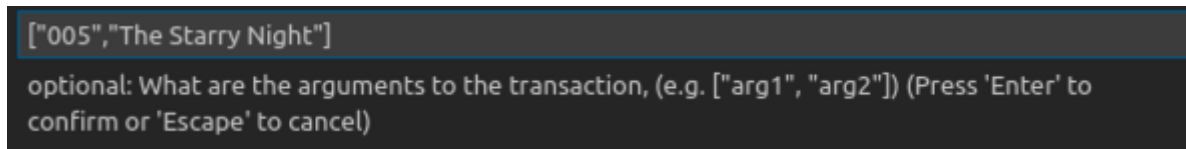

The transaction will be submitted and the Output view updated with the log of the transaction. Note that it includes the new event, which has been received and logged by the first subscriber (IBM Blockchain Platform):
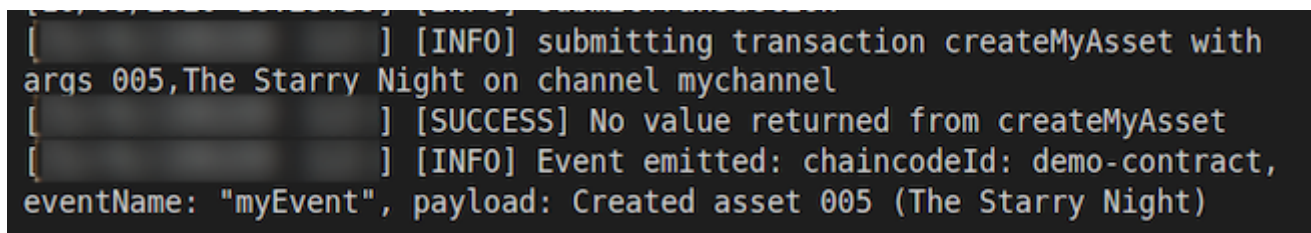
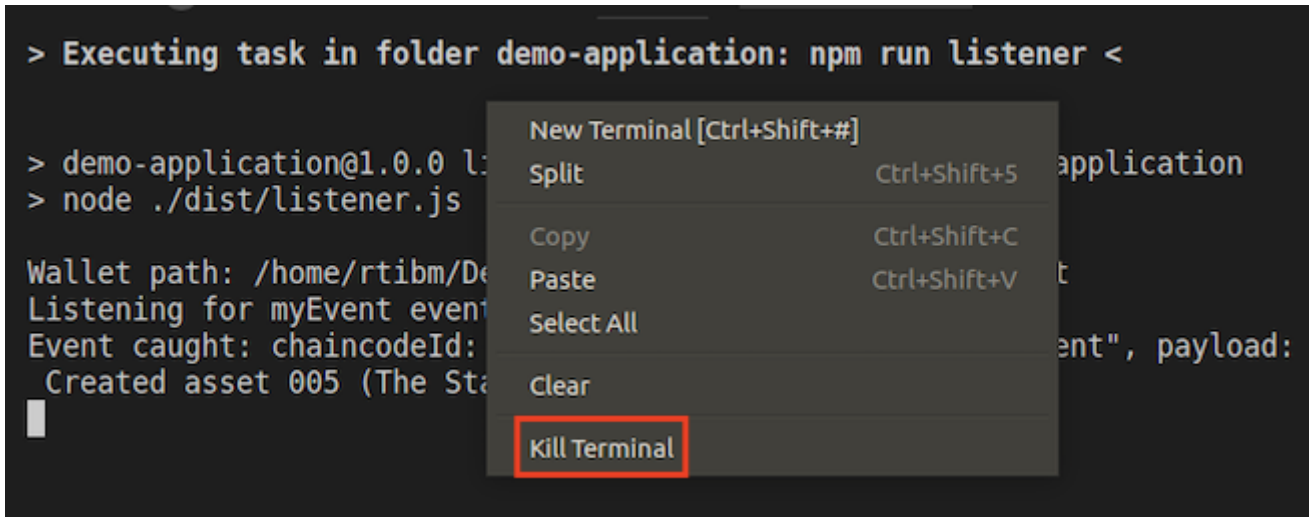We will now verify that the event has also been received by the second subscriber (the listener application):

☐ **A9.30:** Click the Terminal tab to show the running listener application.

Notice that the subscriber has successfully caught the event and run the handler code:
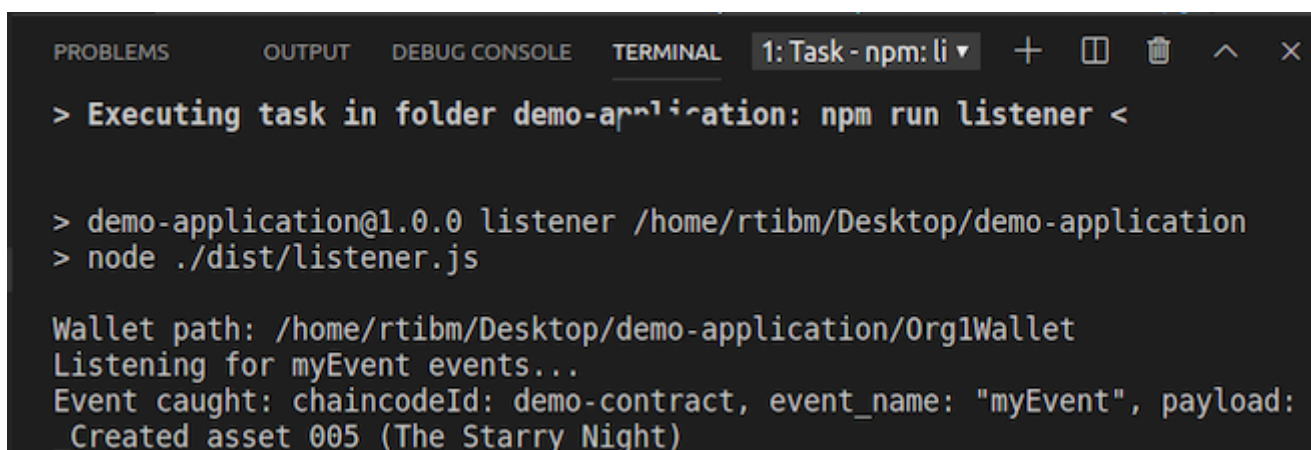
Feel free to continue trying out the event framework. When you have finished, remember to close down the listener application:

☐ **A9.31:** Right click in the running listener's terminal and select 'Kill Terminal'.

This will force the application to close.

Summary

In this tutorial, we have updated one of the transactions in a smart contract to emit an event, subscribed to this event by specifying the correct topic string, and observed the event being output to the VS Code console. We then created a client application to subscribe to the same topic, and checked that it too could receive events.

In the final tutorial of this set, we will summarize what we have covered so far and invite you to claim a badge for your efforts!