Architecture Ethereum DApp with Angular, Angular Material and NgRx

Alex & Daniel Yevseyevish

Table of Contents

Introduction

This is a step-by-step guide demonstrating how to build a modern Ethereum blockchain DApp with Ethers.js and IPFS using Angular and NgRx. We will create an end-to-end marketplace DApp named `FleaMarket` that runs on Ethereum blockchain and implements the functionality of the `Safe Remote Purchase` contract. The entire application is built with the VS Code. All JavaScrip libraries have been updated to the latest major release. The complete source code is located at the GitHub repository.

Tutorial Structure

This tutorial will be presented in the following parts:

1. Part 1: Setup a smart contract project using the Truffle framework and description of the Solidity contracts.
2. Part 2: Overview of some useful tools for catching style inconsistencies and security vulnerabilities.
3. Part 3: Deploying the smart contracts on the Ropsten network.
4. Part 4: Testing our smart contracts against the Ganache network.
5. Part 5: Start scaffolding a new Angular project using the Angular CLI toolkit.
6. Part 6: Customize the default error handling in Angular application.
7. Part 7: Introduction to the NgRx global store.
8. Part 8: Wire up the ethers.js Web3 Provider to manage communication between MetaMask and Ethereum blockchain.
9. Part 9: Setup the connection to an IPFS node running on Infura.
10. Part 10: Uploading images to the IPFS peer. Using IPFS hashes to retrieve data and display it on the browser.
11. Part 11: Extend our NgRx Store to include a new lazily-loaded feature state. Introduce the entity state to manage the product collection.
12. Part 12: Walkthrough the code that drives the end-to-end logic of purchasing products.

Part 1: The FleaMarket Escrow Smart Contract Project

We are going to be using the Truffle framework to build our project. First, run the following commands to upgrade Truffle to the latest version:

```
npm uninstall -g truffle
npm install -g truffle
```

Open the VS Code and create a new project directory `flea-marketplace-dapp`. In the terminal window initialize a new Truffle project:

```
truffle init
```

Run the `npm init` command to create a default `package.json` file. Then, install the following OpenZeppelin smart contracts library:

```
npm install @openzeppelin/contracts --save-dev
```

There are several contracts included in this project:

1. `SafeRemotePurchase.sol` is the main contract representing a specific product for sale.
2. `FleaMarketFactory.sol` is the factory contract, which lets the users create and deploy their sale contracts `SafeRemotePurchase.`
3. `HitchensUnorderedKeySet.sol` is a helper library that implements the Solidity CRUD pattern.
4. `Ownable.sol` is the OpenZeppelin contract that provides exclusive access to specific contract functions.
5. `SafeMath.sol` is the OpenZeppelin utility library for performing safe arithmetic operations.

The code for the contract `SafeRemotePurchase,` with several modifications, has been adopted from the Solidity documentation site.

```
pragma solidity ^0.6.0;
```

```
import "@openzeppelin/contracts/math/SafeMath.sol";
import "./Ownable.sol";

contract SafeRemotePurchase is Ownable {
    using SafeMath for uint256;

    uint256 private _commissionRate;  // for example, 350 ==> (350/100)=3.5%
    address payable public seller;
    address payable public buyer;
    uint256 public price;
    bytes32 public key;  // unique string identifier
    string public description;
    string public ipfsImageHash;

    enum State {
        Created,
        Locked,
        Canceled,
        ItemReceived,
        SellerPaid,
        OwnerPaid,
        Completed
    }
    State public state;

    // Contract created by the seller
    // Ensure that `msg.value` is an even number.
    constructor(
        uint256 _rate,
        address payable _seller,
        bytes32 _key,
        string memory _description,
        string memory _ipfxImageHash
    ) public payable {

        require(_key != 0x0, "Key cannot be 0x0");
        require(bytes(_description).length > 0, "Description can't be empty");
        require(_rate > 0, "Must specify the commission rate");
        require(_seller != address(0), "The seller is the zero address");

        _commissionRate = _rate;
        seller = _seller;
        key = _key;
        ipfsImageHash = _ipfxImageHash;
```

```
  description = _description;
  price = msg.value.div(2);

  require(price > 0, "Must specify price");
  require((price * 2) == msg.value, "Price has to be even");
}

modifier condition(bool _condition) {
    require(_condition, "Condition is not valid.");
    _;
}
modifier onlyBuyer() {
    require(msg.sender == buyer, "Only buyer can call this.");
    _;
}
modifier onlySeller() {
    require(msg.sender == seller, "Only seller can call this.");
    _;
}
modifier inState(State _state) {
    require(state == _state, "Invalid state.");
    _;
}

event LogCanceledBySeller(
    address indexed sender,
    uint256 amount,
    bytes32 key
);
event LogPurchaseConfirmed(
    address indexed sender,
    uint256 amount,
    bytes32 key
);
event LogReceivedByBuyer(
    address indexed sender,
    uint256 amount,
    bytes32 key
);
event LogWithdrawBySeller(
    address indexed sender,
    uint256 amount,
    bytes32 key
);
event LogWithdrawByOwner(
```

```
    address indexed sender,
    uint256 amount,
    bytes32 key
);

// Confirm the purchase as buyer.
// Transaction has to include `2 * value` ether.
// The ether will be locked until buyerConfirmReceived is called
function buyerPurchase()
    external
    payable
    inState(State.Created)
    condition(msg.value == price.mul(2))
    returns (bool result)
{
    buyer = msg.sender;
    state = State.Locked;
    emit LogPurchaseConfirmed(msg.sender, msg.value, key);
    return true;
}

// Confirm that the buyer received the item from the seller.
// The buyer will receive the locked ether in the amount of the price.
function buyerConfirmReceived()
    external
    onlyBuyer
    inState(State.Locked)
    returns (bool result)
{
    state = State.ItemReceived;
    buyer.transfer(price);
    emit LogReceivedByBuyer(msg.sender, price, key);
    return true;
}

// Abort the purchase and reclaim the ether.
// Can only be called by the seller before
// the contract is locked.
function abortBySeller()
    external onlySeller
    inState(State.Created)
    returns (bool result)
{

    uint256 amount = balanceOf();
```

```
    state = State.Canceled;

    // We use transfer here directly. It is
    // reentrancy-safe, because it is the
    // last call in this function and we
    // already changed the state.
    seller.transfer(amount);
     emit LogCanceledBySeller(msg.sender, amount, key);
    return true;
}

function withdrawBySeller()
    external
    onlySeller
    condition(state == State.ItemReceived || state == State.OwnerPaid )
    returns (bool result)
{
  if (state == State.OwnerPaid) {
        uint256 amount = balanceOf();
        state = State.Completed;
        seller.transfer(amount);
        emit LogWithdrawBySeller(msg.sender, amount, key);
        return true;
      } else if (state == State.ItemReceived) {
          // calculate commission part
          uint256 commission  = (price.mul(_commissionRate)).div(10000);
          // subtracts commission part: 3.5% ==> 350/100
          uint256 amount = (price.mul(3)).sub(commission);
          state = State.SellerPaid;
          seller.transfer(amount);
          emit LogWithdrawBySeller(msg.sender, amount, key);
          return true;
      } else {
          return false;
      }
}

function withdrawByOwner()
    external
    onlyOwner
    condition(state == State.ItemReceived || state == State.SellerPaid )
    returns (bool result)
{
    if (state == State.SellerPaid) {
          uint256 amount = balanceOf();
```

```
            state = State.Completed;
            owner().transfer(amount);
            emit LogWithdrawByOwner(msg.sender, amount, key);
            return true;
        } else if (state == State.ItemReceived) {
            // calculate commission part: 3.5% ==> 350/100
            uint256 commission  = (price.mul(_commissionRate)).div(10000);
            state = State.OwnerPaid;
            owner().transfer(commission);
            emit LogWithdrawByOwner(msg.sender, commission, key);
            return true;
        } else {
            return false;
        }
    }

    // only owner (==deployer) and seller can see it
    function commissionRate()
        external
        view
        condition(isOwner() || msg.sender == seller )
        returns (uint commission)
    {
        return _commissionRate;
    }

    // Get balance of the contract
    function balanceOf() public view returns(uint) {
        return address(this).balance;
    }

    // Prevent someone sending ether to the contract
    // It will cause an exception,
    // because the fallback function does not have
    // the 'payable' modifier.
    fallback () external {
        revert("No Ether excepted");
    }

}
```

On every new product we're putting up for sale, the factory smart contract `FleaMarkeFactory` will spawn a new child product contract represented by the `SafeRemotePurchase` smart contract. We deal with the implementation of the Solidity CRUD pattern with the following requirements:

- Each product has a unique key
- Ensure key uniqueness
- Insert a product with a key identifier
- Retrieve a product by its key identifier
- Remove a product by its key identifier
- Obtain a count of the products that exist
- Iterating over the products

Fortunately, a general-purpose library HitchensUnorderedKeySet that tackles this task has been introduced by Rob Hitchens in his great post Solidity CRUD- Epilogue.

By applying this pattern to `FleaMarketFactory` contract, the code should look like this:

```
pragma solidity ^0.6.0;

import "./HitchensUnorderedKeySet.sol";
import "./SafeRemotePurchase.sol";
import "./Ownable.sol";

contract FleaMarketFactory is Ownable {
    using HitchensUnorderedKeySetLib for HitchensUnorderedKeySetLib.Set;

    HitchensUnorderedKeySetLib.Set private widgetSet;
    string public contractName;
    struct WidgetStruct {
      // pointer on the child contract
      address purchaseContract;
    }
    mapping(bytes32 => WidgetStruct) private widgets;

    constructor() public {
        contractName = "FleaMarket Smart Contract";
    }


    event LogCreatePurchaseContract(
        address sender,
        bytes32 key,
        address contractAddress
```

```
);
event LogRemovePurchaseContract(address sender, bytes32 key);

// commissionRate, for example, 350 ==>  (350/100)=3.5%
function createPurchaseContract(
   bytes32 key,
   string calldata description,
   string calldata ipfsImageHash,
   uint256 commissionRate
  ) external payable returns(bool createResult) {
   // Note that this will fail automatically if
   // the key already exists.
   widgetSet.insert(key);
   WidgetStruct storage wgt = widgets[key];

   // msg.sender would be the seller
   SafeRemotePurchase c =
        (new SafeRemotePurchase).value(msg.value)(
     commissionRate,
     msg.sender,
     key,
     description,
     ipfsImageHash
   );
   /*
   When a new children contract is created
   the msg.sender value passed to the Ownable
   is the address of the parent contract.
   So we need to tell the child contract who is the contract manager
   */
   c.transferOwnership(owner());
   // cast contract pointer to address
   address newContract = address(c);
   wgt.purchaseContract = newContract;
   emit LogCreatePurchaseContract(msg.sender, key, newContract);
   return true;
}

function getContractCount() public view returns(uint contractCount) {
   return widgetSet.count();
 }

 function getContractKeyAtIndex(uint index)
   external
```

```
        view
        returns(bytes32 key)
     {
        return widgetSet.keyAtIndex(index);
     }

    function getContractByKey(bytes32 key)
        external
        view
        returns(address contractAddress)
    {
        require(
            widgetSet.exists(key),
            "Can't get a widget that doesn't exist."
        );
        WidgetStruct storage w = widgets[key];
        return (w.purchaseContract);
    }

    function removeContractByKey(bytes32 key)
        external
        onlyOwner
        returns(bool result)
    {
        // Note that this will fail automatically if the key doesn't exist
        widgetSet.remove(key);
        delete widgets[key];
        emit LogRemovePurchaseContract(msg.sender, key);
        return true;
    }
}
```

There are a few obvious benefits of using this application in real-life. Potential problems of purchasing remotely arise from the buyer and seller's distrust for each other. The buyer would like to receive an item from the seller, and the seller would like to get money for the item in return. The issue here is in the delivery phase. For this reason, escrow services (the middleman) exists. The escrow service takes the buyer's money, holds on to it and only releases it to the seller when the buyer confirms that he has received it.

The smart contract above is an attempt to solve this problem. Both parties have to put twice the value of the item into the contract as an escrow. As soon as this happens, the money will stay locked inside the

contract until the buyer confirms that he has received the item. After that, the buyer is refunded back half of his deposit and the seller gets three times the value of the item (his deposit plus the item value he sold). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked in forever.

In summary:

1. It eliminates the need for face-to-face contact between buyer and seller.
2. It eliminates the need for a third-party escrow agent service.
3. There is no need to pay the listing fee.
4. It motivates both sellers and buyers to care for the transaction fairly from start to finish, by requiring both parties to deposit ETH worth 2x the value of the item for sale.

To compile the smart contracts, we need to edit the Truffle configuration file `truffle-config.js` to specify the Solidity compiler settings. The configuration should look like this:

```
compilers: {
    solc: {

      version: '0.6.0',
      settings: {
        optimizer: {
          enabled: true,
        },

        evmVersion: 'petersburg', //'byzantium'
      },
    },
  },
```

Notice, that in Solidity 0.6.0 we have to explicitly define the EVM version for the compile to avoid some Solidity EVM runtime errors, as it is discussed here.

Then we execute the command:

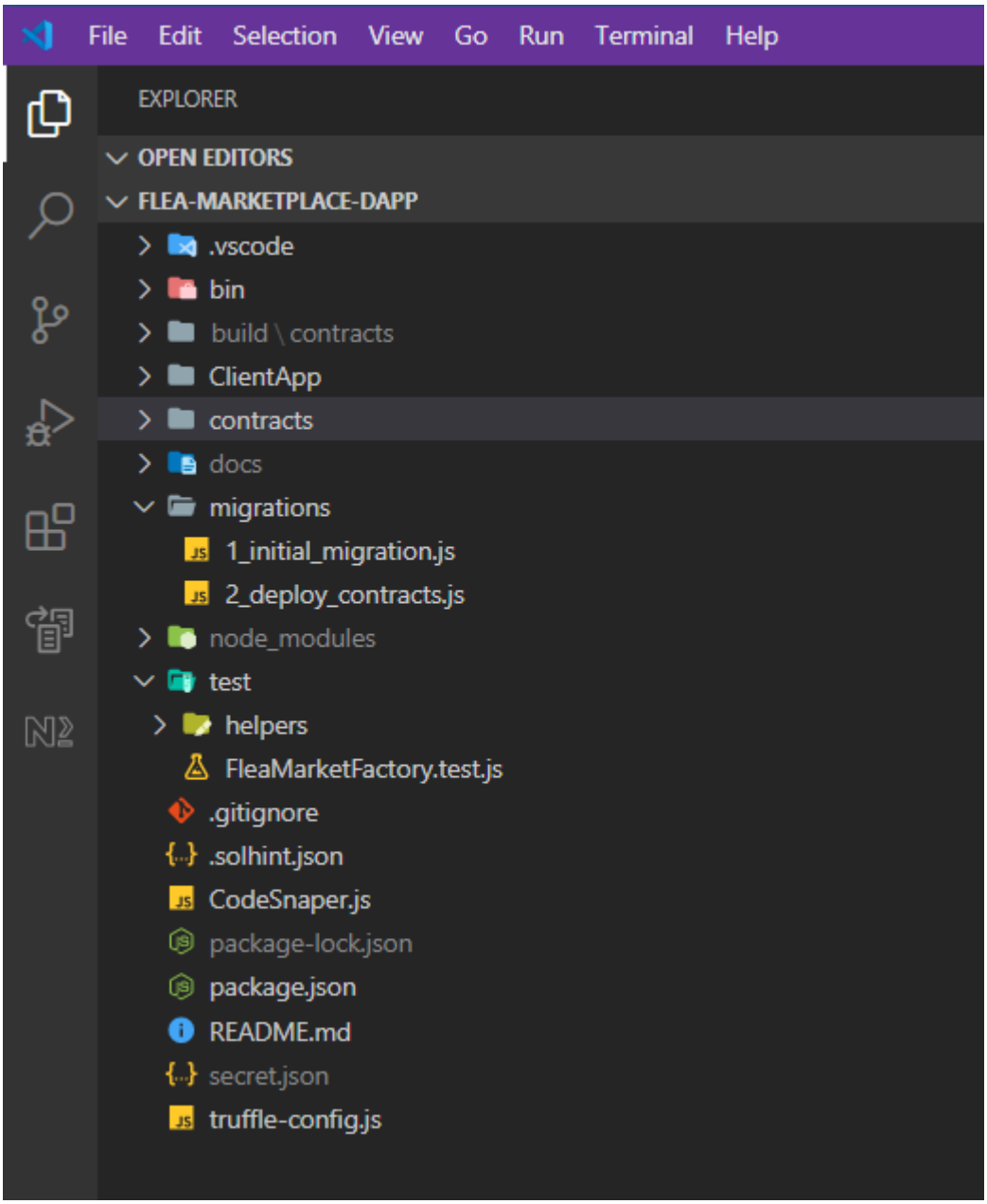
```
truffle compile
```

The contracts should successfully compile. Artifacts generated by Truffle compilation will be placed in the `build/contracts/` directory.

To push our project to the GitHub repository, create the `.gitignore` file, and tell it to ignore the `node_modules`. Next, go to the terminal command line in the VS code and run the following set of the commands:

```
git init git add -A
git commit -m "Initial commit"
git remote add origin [remote repository URL]
git push -u origin master
```

Our project structure should look something like this:


File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
FLEA-MARKETPLACE-DAPP
.vscode
bin
build \ contracts
ClientApp
contracts
docs
migrations
1_initial_migration.js
2_deploy_contracts.js
node_modules
test
helpers
FleaMarketFactory.test.js
.gitignore
.solhint.json
CodeSnaper.js
package-lock.json
package.json
README.md
secret.json
truffle-config.js

Part 2: Contract Security and Style Guide Validations

"It's not how much money you make, but how much money you keep …"

-- Robert Kiyosaki

Due to the immutable nature of smart contracts, it's basically impossible to fix/change the already deployed contracts. It is important to apply defensive tools to analyze code for possible security vulnerabilities and styling errors. In this tutorial, we will use Solhint and MythX.

Solhint is for linting the solidity code and provides both reviews for security and styling validations. To install the solhint tool module, run the following command:

```
npm install --save-dev solhint
```

Next, add the solhint configuration file `solhint.json` to the project by running the command:

```
./node_modules/.bin/solhint init-config
```

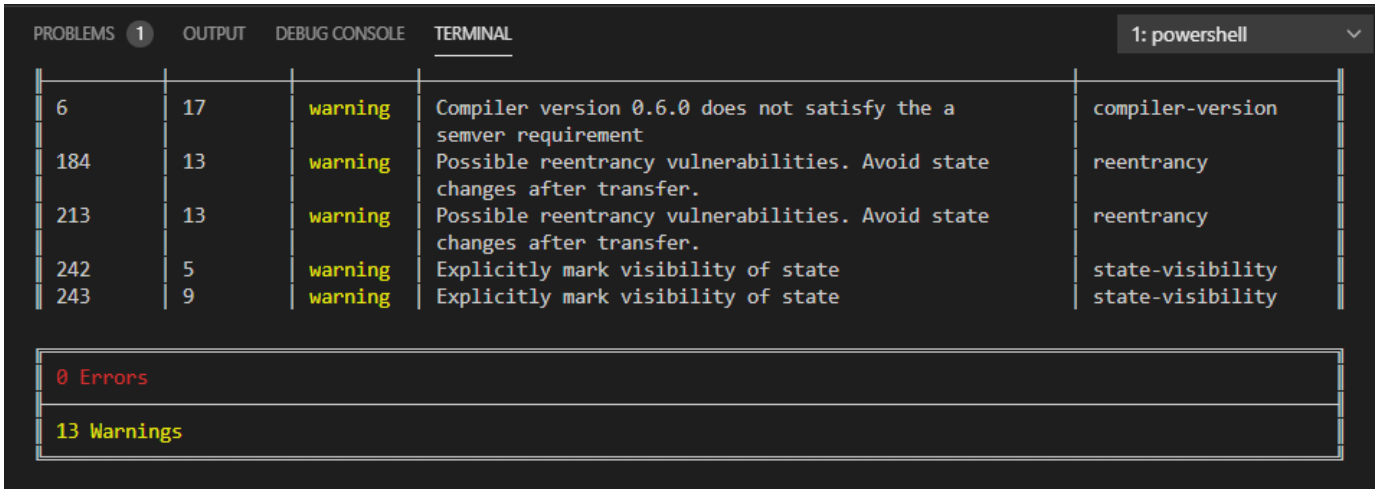
This file can be customized by adding supported security rules or disable certain validations.

```
{
  "extends": "solhint:recommended",
  "plugins": [],
  "rules": {
    "avoid-suicide": "error",
    "avoid-sha3": "warn",
    "avoid-tx-origin:": "warn",
    "avoid-throw": "off",
    "indent": "off",
    "no-mix-tabs-and-spaces": "off",
    "compiler-version": "warn"
  }
}
```

For convenience, we can edit the `package.json` to include a script to run the Solhint.

```
"solhint": "./node_modules/.bin/solhint -f table contracts/**/*.sol"
```

The resulting output may look similar to this:



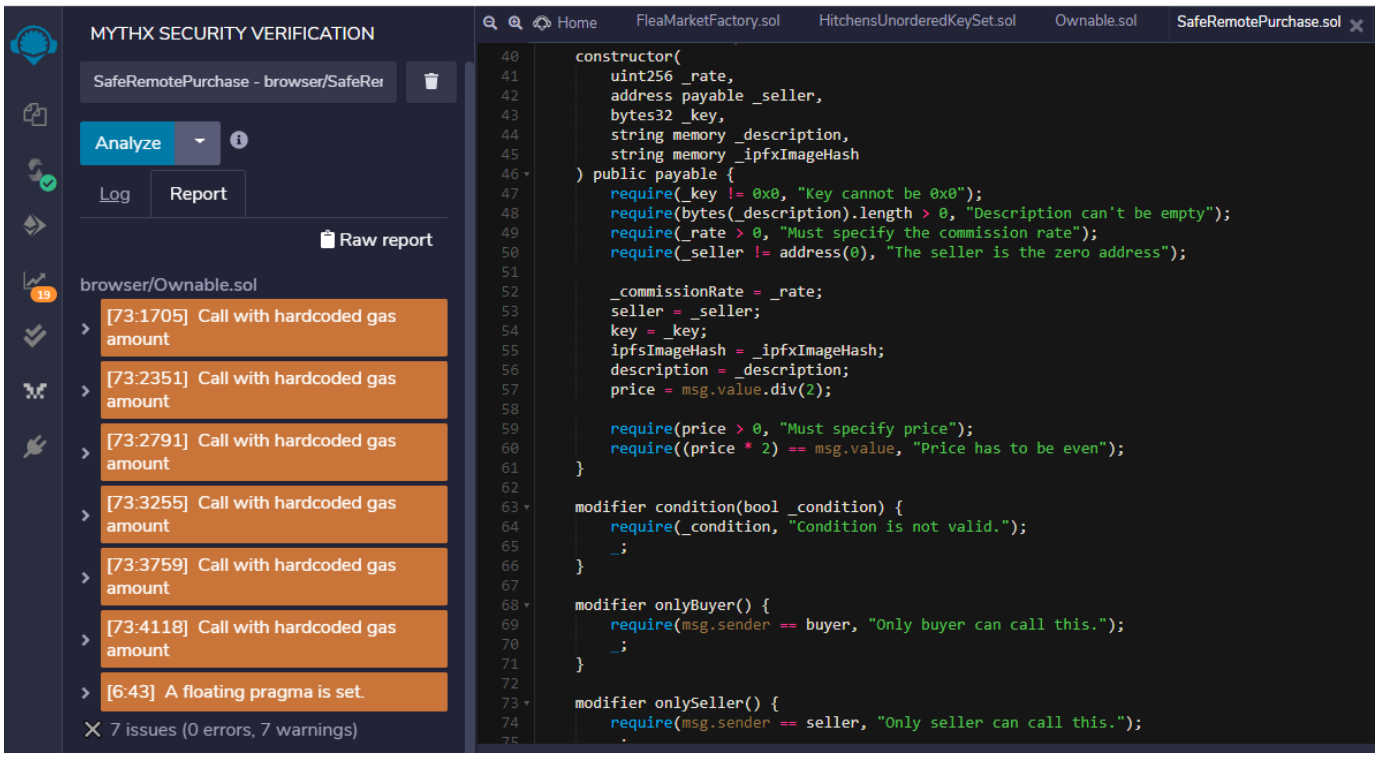
PROBLEMS 1 | OUTPUT | DEBUG CONSOLE | TERMINAL | 1: powershell

6	17	warning	Compiler version 0.6.0 does not satisfy the a semver requirement	compiler-version
184	13	warning	Possible reentrancy vulnerabilities. Avoid state changes after transfer.	reentrancy
213	13	warning	Possible reentrancy vulnerabilities. Avoid state changes after transfer.	reentrancy
242	5	warning	Explicitly mark visibility of state	state-visibility
243	9	warning	Explicitly mark visibility of state	state-visibility

0 Errors

13 Warnings

MythX is a security-specific tool that addresses smart contract vulnerabilities. It can be used as the VS Code extension, but it also available as a plugin for Remix. To use this tool in unrestricted mode, you'll need to sign up for an account in the MythX dashboard. Once you have done this, you will need to add your credentials to the Remix plugin panel and run Analyzer:

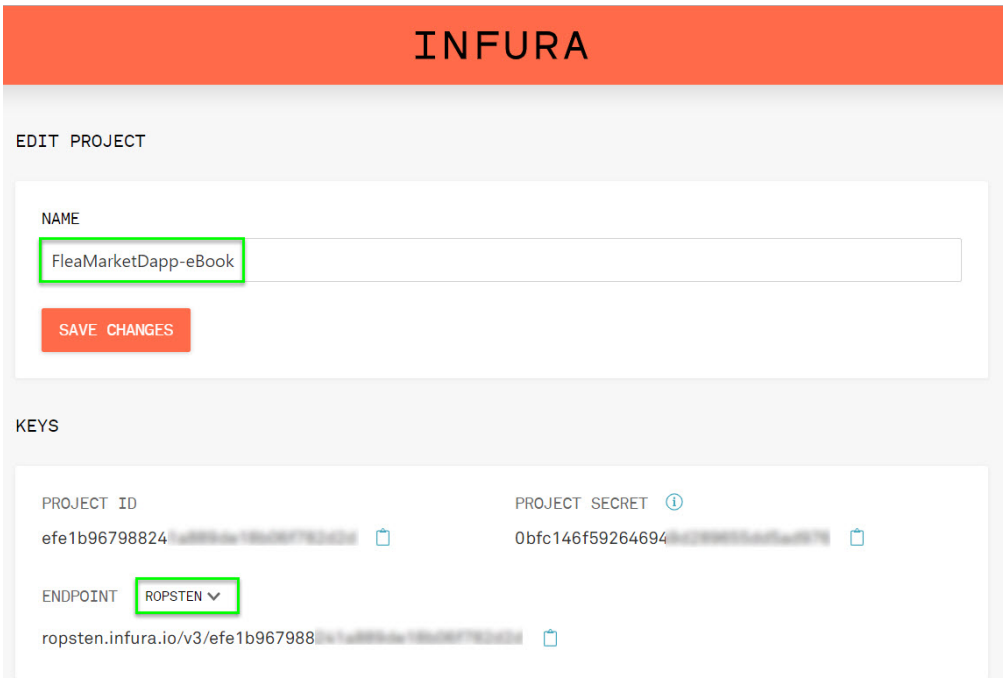

MYTHX SECURITY VERIFICATION
SafeRemotePurchase - browser/SafeRe
Analyze
Log
Report
Raw report
browser/Ownable.sol
[73:1705] Call with hardcoded gas amount
[73:2351] Call with hardcoded gas amount
[73:2791] Call with hardcoded gas amount
[73:3255] Call with hardcoded gas amount
[73:3759] Call with hardcoded gas amount
[73:4118] Call with hardcoded gas amount
[6:43] A floating pragma is set.
7 issues (0 errors, 7 warnings)
Home
FleaMarketFactory.sol
HitchensUnorderedKeySet.sol
Ownable.sol
SafeRemotePurchase.sol
constructor(
uint256 _rate,
address payable _seller,
bytes32 _key,
string memory _description,
string memory _ipfxImageHash
) public payable {
require(_key != 0x0, "Key cannot be 0x0");
require(bytes(_description).length > 0, "Description can't be empty");
require(_rate > 0, "Must specify the commission rate");
require(_seller != address(0), "The seller is the zero address");
_commissionRate = _rate;
seller = _seller;
key = _key;
ipfsImageHash = _ipfxImageHash;
description = _description;
price = msg.value.div(2);
require(price > 0, "Must specify price");
require((price * 2) == msg.value, "Price has to be even");
}
modifier condition(bool _condition) {
require(_condition, "Condition is not valid.");
_;
}
modifier onlyBuyer() {
require(msg.sender == buyer, "Only buyer can call this.");
_;
}
modifier onlySeller() {
require(msg.sender == seller, "Only seller can call this.");

Part 3: Deploy Smart Contract on Ropsten Network

Now that we've compiled and validated our smart contract, we can deploy it to the Ropsten public test blockchain. To do this, we need to add a second migration file.
In the `./migrations` folder, create a deployment script file `2_deploy_contracts.js` with the following content:

```
const FleaMarketContract = artifacts.require("FleaMarketFactory");

module.exports = async (deployer) => {
   await deployer.deploy(FleaMarketContract);
   const contract = await FleaMarketContract.deployed();
   console.log(`Contract has been deployed successfully: ${contract.address}`);
};
```

The next step is to edit the `truffle-config.js` file to provide all the necessary configuration for deploying our smart contract to Ropsten. First, we need to authenticate our application with Infura which provides access to Ropsten nodes via APIs. To do this, let's login to Infura. In the dashboard, create a new project:



A new project will be set up with `Project ID`, along with the link to the corresponding V3 API endpoint URL. This endpoint URL will be used to send Ethereum requests from our DApp to the Ropsten Ethereum blockchain.

Next, let's install the Truffle HDWalletProvider. This provider will manage our private keys and handle the transaction signing as well as the connection to the Ropsten network.

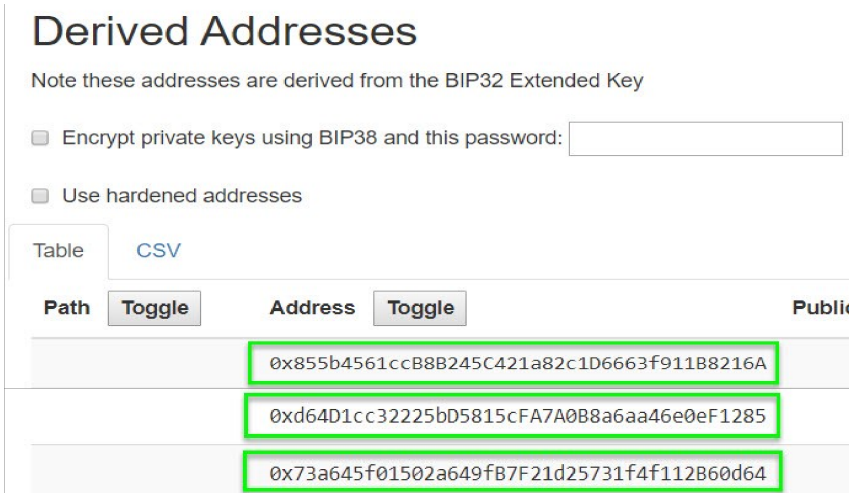
```
npm install @truffle/hdwallet-provider --save
```

Notice, that if you are on Windows, you may be required to install the Windows Build Tools npm package. In the terminal with Administrator rights, run

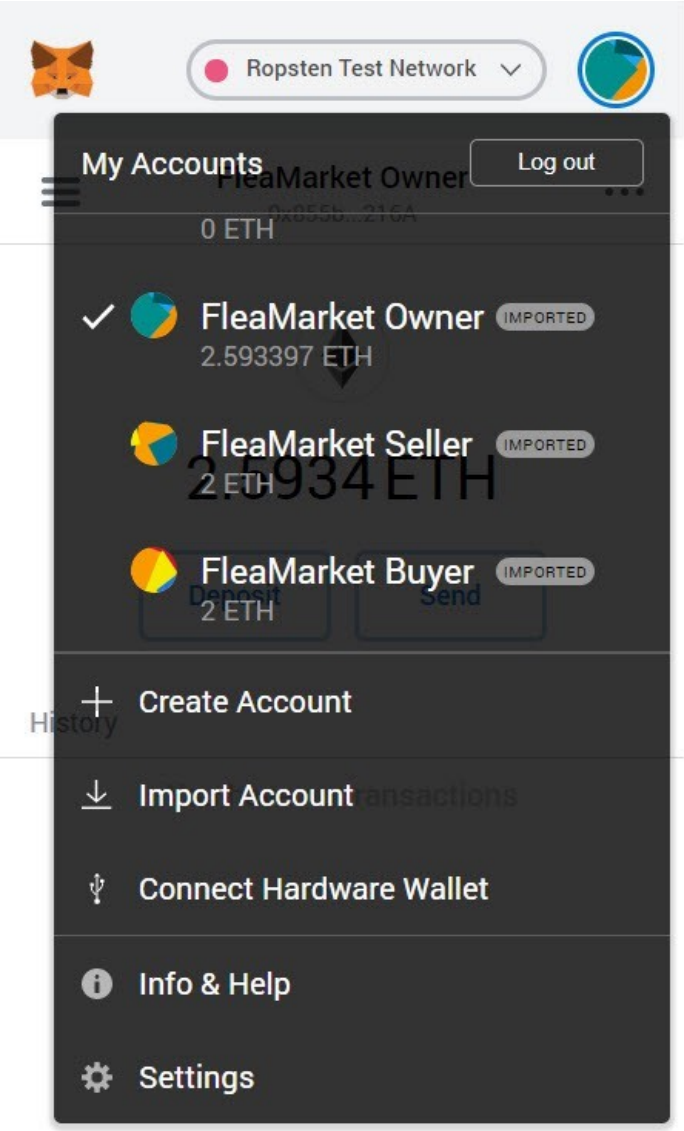
```
npm install -g --production windows-build-tools
```

To use the hdwallet provider, we need to provide a mnemonic that will generate the accounts used for deployment on the Ropsten blockchain. We can generate one using the online mnemonic generator.

We can also see the list of the accounts that are associated with the given mnemonic and can be used on any Ethereum blockchain including Ropsten:



Next, connect your Chrome browser to MetaMask digital wallet. On MetaMask, change the network to 'Ropsten Test Network' and import the first three accounts as follows:



We also want to make sure that each account has enough ETH to run the transactions. Visit the [Ropsten MetaMask faucet]() for requesting some free ETH.

The next step is to edit the `truffle-config.js` file to use with the hdwallet provider:

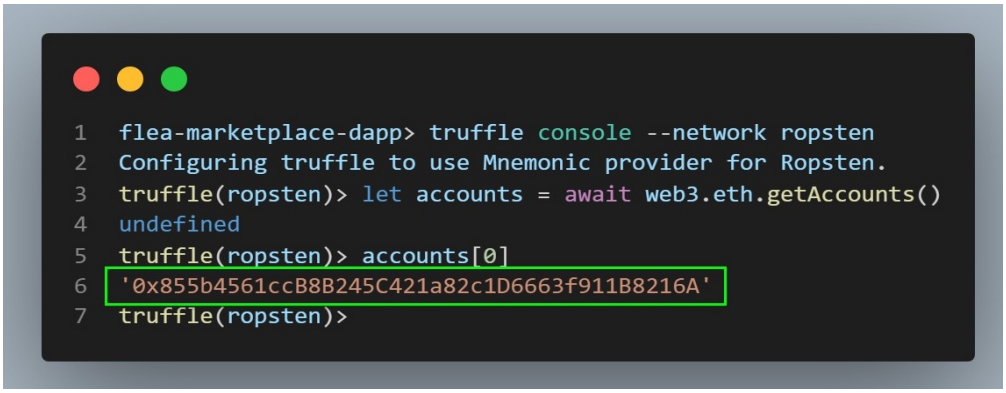
```
ropsten:{
  provider: () => {
    try {
          const fileContents =
            readFileSync(path.join(__dirname, 'secret.json'), 'utf8');
          const data = JSON.parse(fileContents);

          const privateKey = data.mnemonic;
          // Your project id, which we copied from Infura.io
```

```
            const infuraProjectId = data.infuraProjectToken;
            const rpcUrl = `https://ropsten.infura.io/v3/${infuraProjectId}`;
            // Account in charge to deploy a smart contract
            const ropstenAccountId = 0;
            // How many addresses in the wallet should we unlock
            const numAddressesToUnlock = 3;
            console.log('Configuring truffle to use Mnemonic provider for Ropsten.');

            return new HDWalletProvider(privateKey, rpcUrl,ropstenAccountId,
               numAddressesToUnlock);
         } catch (err) {
            console.error('Error', err);
         }
   },
   network_id: 3, // Ropsten network id
   gas: 4700000,
},
```

Here, the `mnemonic` is a 12-word secret mnemonic phrase to be used by our hdwallet provider, and `infuraProjectToken` is the `Project ID` that has been granted by Infura. We store this information in the `secrets.json` which has been also added to the *`.gitignore`* to avoid be checked-in into the public repository.

Notice, by default, in the `HDWalletProvider` constructor, the account in charge of the smart contract deployment will be the first one generated by the mnemonic. If we pass it in a specific index, then it will use that address instead (the index is zero-based).

The quick way to validate `HDWalletProvider`, that it has been configured correctly and able to communicate with the Ropsten network, is to open the Truffle console and run the following commands:

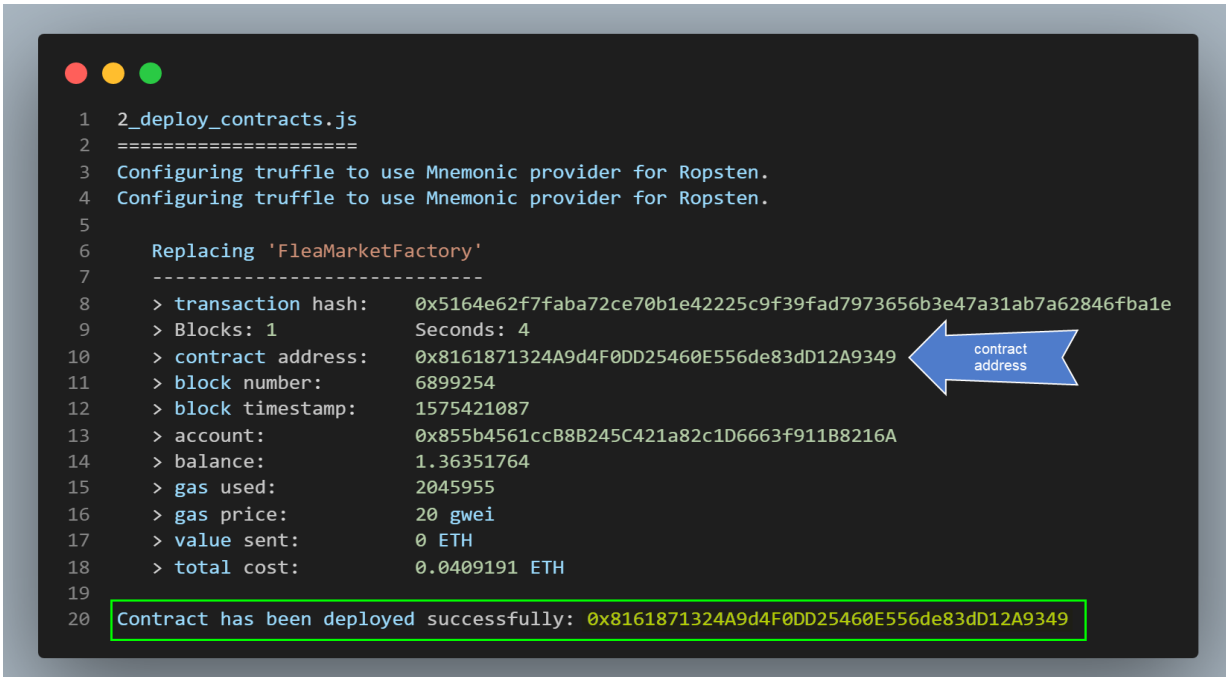


We can confirm that the `account[0]` is the same as the first account generated by the mnemonic.

To compile and deploy our smart contracts on the blockchain with the migration scripts we created above, execute the command:

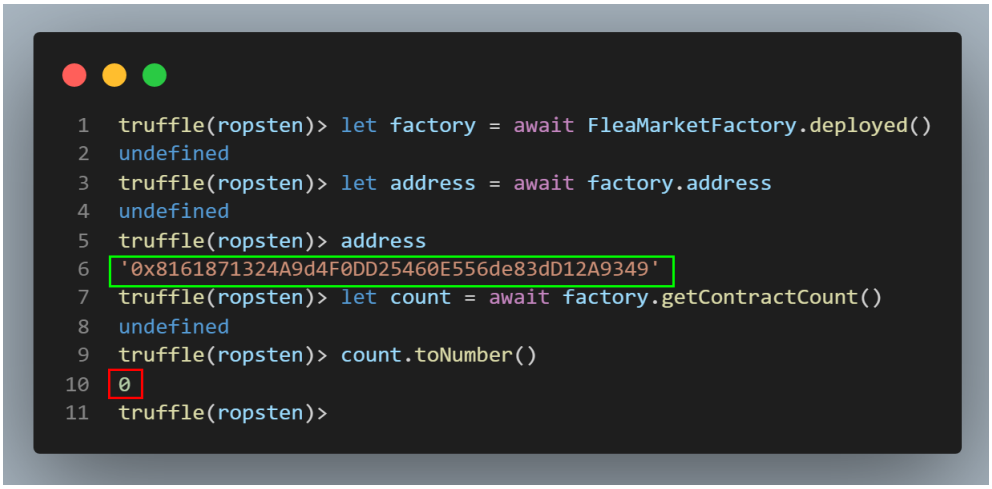
```
truffle migrate --reset --network ropsten
```

If everything goes as well as expected, we should observe a console log in the terminal as follows:



Next, we can open the Truffle console and check if we can interact with our contract:

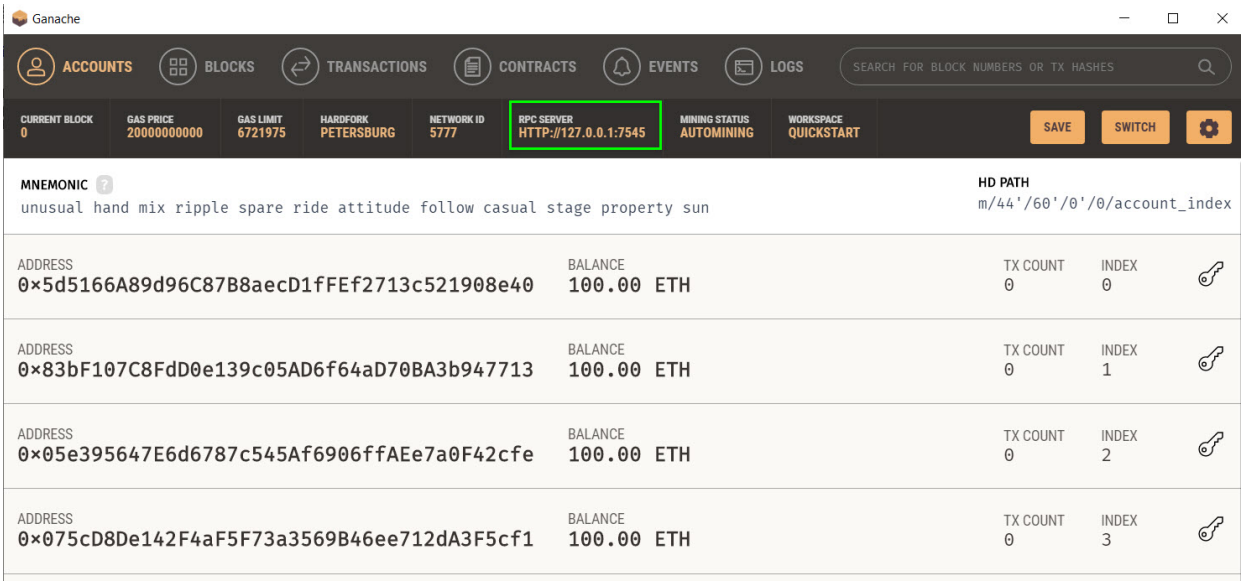
```
truffle console --network ropsten
```



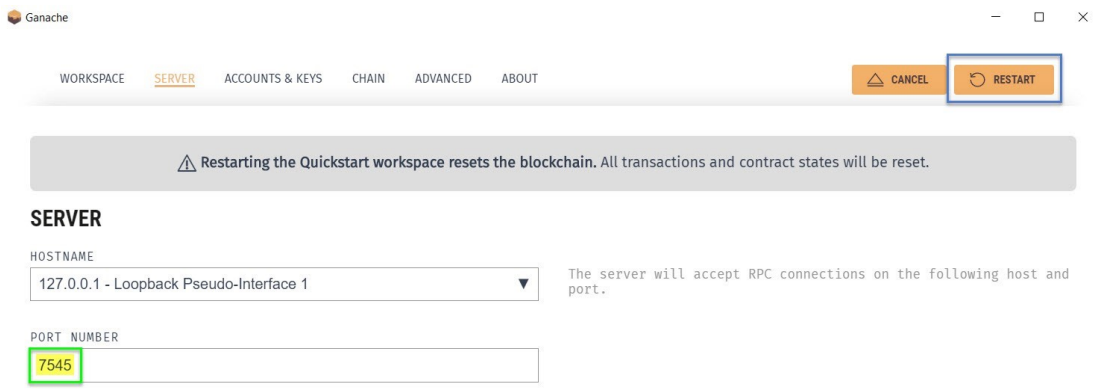
Once the code is run, we can confirm that the smart contract has been successfully deployed to the Ropsten.

Part 4: Testing Smart Contracts

In this part, we look at how to test our smart contracts. The first step is to set the Ethereum blockchain network running on the local machine. Install and run Ganache from the Truffle suite. Once we run Ganache, we should see a screen like this:



By default, Ganache is listening on the port `7545` and comes up with several accounts funded with fake money that we can use to deploy and interact with the smart contract during development. From the Ganache setting page, we can modify the port number and refill its accounts.



Notice that even though Ganache is a virtual Ethereum blockchain running on our local machine, it could be used the same way as any real Ethereum blockchain, which means you can connect to it with wallets like MyEtherWallet or a browser wallet like MetaMask. By connecting MetaMask to Ganache, we can also take advantage of Remix for compilation, debugging, and interaction with contracts. To connect Ganache and Remix, we need to select the option 'Injected Web3' from Remix's Environment menu.

To access the Ganache blockchain from Truffle framework, we need to add the Ganache network settings to the truffle project's configuration file `truffle-config.js`:

Next, we add the Chai libraries in our project file `package.json`

```
npm install --save-dev chai chai-bn chai-as-promised
```

The first library is for using the assertions with Chai, such as *`assert`*, *`expect`*, and *`should`*. The second one provides the Chai assertions of the big numbers in JavaScript bn.js. The third one is for testing promises.

We also install the openzeppelin-test-helpers library, which gives us some handy utilities for validating contract state, such as *`expectRevert`* and *`expectEvent`*.

```
npm install --save-dev @openzeppelin/test-helpers
```

The complete test suite that checks and validates the different aspects of the smart contract functionality is provided in Appendix A. It simulates client-side interaction with our smart contracts.

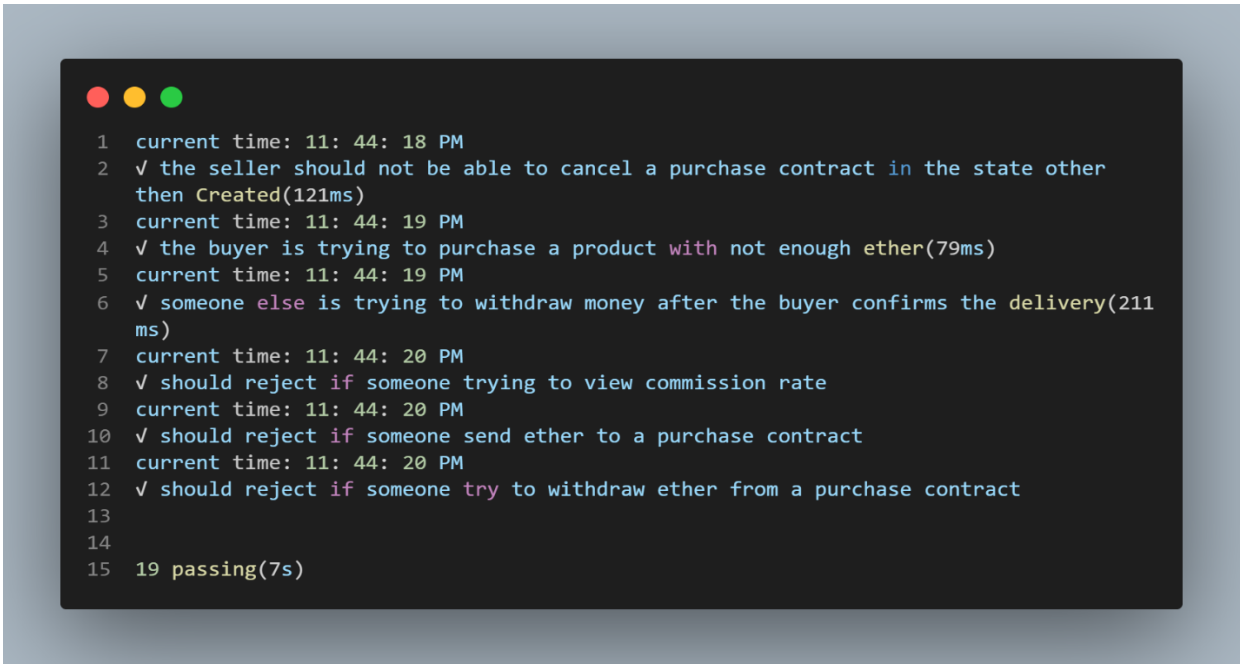
Finally, run the test against the Ganache network with the truffle command:

```
truffle test --network ganache
```

This does the following:

1. Compiles our contracts
2. Runs migrations to deploy the contracts to the network
3. Runs tests against the contracts deployed on the network

We should see the resulting output similar to this:



Part 5: Scaffolding Angular App

Let's get started by installing the latest major versions of the Angular CLI.

```
npm install -g @angular/cli
```

First, open up a terminal window in the root folder of your Truffle project and create a new Angular project by running the following CLI command:

```
ng new ClientApp --skip-install=true --minimal=true
--style=css --routing=true --skipGit=true
```

Next, install the Angular Material, by running the following schematic:

```
ng add @angular/material
```

Add a navigation component to the application by running the `navigation` schematic:

```
ng generate @angular/material:nav nav
```

We can also add a dashboard component using the material schematic:

```
ng generate @angular/material:dashboard dashboard
```

Then, install the Flex-layout module, which is a superior layout engine to assist with the CSS flex-box features:

```
npm install @angular/flex-layout --save
```

Now we can wire up the NgRx framework, which lays in the core of our application's state management design.

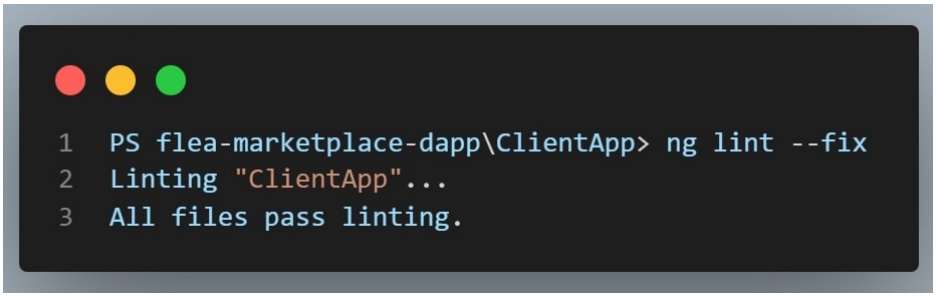
```
npm install @ngrx/store, @ngrx/effects, @ngrx/router-store,
@ngrx/entity, @ngrx/store-devtools --save
```

The latest NgRx library comes with the support of many cool futures such as `createAction`, `createReducer`, and `createEffect` functions. One of the best online examples utilizing @NgRx libraries and showcasing common patterns and the benefits of using them could be found here.

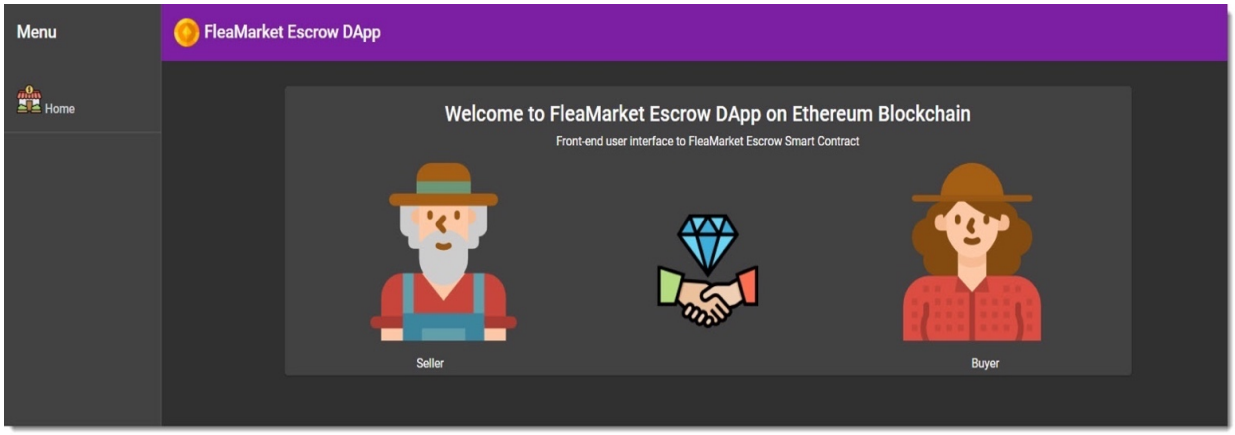
Before we build our project, we want to first lint our code and try to automatically fix any linting errors:

```
ng lint --fix
```

We should see the output similar to this:



To build the app, we run the `ng serve` command. Once we compile and run the basic application, it should look like this:



Part 6: Error Handling

A decentralized application built with Angular that interacts with the Ethereum blockchain and IPFS network has many moving parts and communication channels. With the complexity of the application comes the probability that unexpected error can happen more often.

By default, Angular provides a hook that intercepts all unhandled Errors with the `ErrorHandler` class. We can modify this default behavior by creating a new class that implements the `ErrorHandler`:

```
import { ErrorHandler, Injectable, Injector } from '@angular/core';
import { HttpErrorResponse } from '@angular/common/http';
import { serializeError } from 'serialize-error';
import { SnackBarService } from '../services/snack-bar.service';
import { AppearanceColor } from '../models';

@Injectable({ providedIn: 'root' })
export class AppErrorHandler extends ErrorHandler {

  // We should manually inject the services with Injector.
  constructor(private injector: Injector) {
    super();
   }

   handleError(error: Error | HttpErrorResponse) {
     const notifier = this.injector.get(SnackBarService);
     let message: string;

     if (error instanceof HttpErrorResponse) {
       // Server Error
       message = error.message;
     } else {
       // Client Error
       message = serializeError(error).message;
     }
     notifier.show({
```

```
        message,
        color: AppearanceColor.Error,
        horizontalPosition: 'center',
        verticalPosition: 'bottom'
      });

      // continue with the default global error handler
      super.handleError(error);
    }
  }
```

Since the global error handler is loaded by application first, we can't use the dependency injection in the constructor to inject other services. Instead, we have to inject them with the Injector. With this, we implement the `SnackBarService` service, where we add the logic to display notifications using Angular Material Snackbar:

```
@Injectable({ providedIn: 'root' })
export class SnackBarService {

  static readonly SNACKBAR_DELAY: number = 7000;

  constructor(private matSnackBar: MatSnackBar) { }

  show(messageInfo: SnackBarInterface): void {
    this.matSnackBar.openFromComponent(SnackBarComponent, {
      data: {
        message: messageInfo.message,
        color: messageInfo.color
      },
      duration: SnackBarService.SNACKBAR_DELAY
    });
  }

}
```

Here we also use a very handy npm library `serialize-error` that converts the Error object into the properly formatted error text no matter what type of error is thrown.

We can also take advantage of the HttpInterceptor that is aimed to intercept all HTTP requests and handle them before passing them along.

```
import { Injectable, Injector, ErrorHandler } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpErrorResponse
                  } from '@angular/common/http';
import { Observable } from 'rxjs';
import { tap, retry } from 'rxjs/operators';

// Passes HttpErrorResponse to application global error handler
@Injectable()
export class HttpErrorInterceptor implements HttpInterceptor {
  constructor(private injector: Injector) { }

  intercept(request: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(request).pipe(
      retry(1), // retry one more time
      catchError((err: HttpErrorResponse) => {
         // here we inject the global ErrorHandler
         // which will use our custom global error handler AppErrorHandler
         const appErrorHandler = this.injector.get(ErrorHandler);
         appErrorHandler.handleError(err);
         return throwError(err);
      })
    );
  }
}
```

We retry once before we passing down the error to our custom global error handler.

We also need to tell Angular framework that we are changing the default behavior for error handling in our application:

```
providers: [
    // register the GlobalErrorHandler provider
    { provide: ErrorHandler, useClass: AppErrorHandler },
    { provide: HTTP_INTERCEPTORS, useClass: HttpErrorInterceptor, multi: true },
  ]
```

Part 7: Setting Up the Root State

The application state in NgRx is a single immutable data tree structure. The state tree will start from the root state once the `AppModule` is loaded.

The root state is shared globally among all components. It is registered with `StoreModule.forRoot()` and ensures that the root state is available to all areas of our application immediately. It will then continue to grow in size incrementally as more lazily-loaded Angular modules are loaded into the application. A similar concept applies to the root effects. We register root-level effects with the `EffectsModule.forRoot()` in the CoreModule. It ensures that effects start running immediately after the app is loaded and start listening for all relevant actions as soon as possible.

We define our top-level state interface as follows:

```
export interface AppState {
    router: fromRouter.RouterReducerState<any>;
    spinner: fromSpinner.SpinnerState;
    error: fromError.ErrorState;
  }
```

Currently, it is composed of the following separate slices: the router state, the spinner state, and the error state. We will register additional root sub-states down the road. We also register the three root-level side effects:

```
import { ErrorEffects } from './error.effects';
import { SnackBarEffects } from './snack-bar.effect';
import { SpinnerEffects } from './spinner.effect';

export const effects: any[] = [ErrorEffects, SnackBarEffects,
SpinnerEffects];

export * from './error.effects';
export * from './snack-bar.effect';
export * from './spinner.effect';
```

The Spinner State

When dealing with DApps, it often takes some time for a transaction to return from the state-changing operations on the Ethereum blockchain. From the user experience point of view, we should give some visible indication of the result of these operations.

Let's create a service which will be using Angular Material CDK overlay to display spinner:

```
import { Injectable } from '@angular/core';
import { Overlay, OverlayRef } from '@angular/cdk/overlay';
import { ComponentPortal } from '@angular/cdk/portal';
import { MatSpinner } from '@angular/material';
import { Observable, Subject } from 'rxjs';
import { scan, map, distinctUntilChanged } from 'rxjs/operators';

@Injectable({
    providedIn: 'root',
})
export class SpinnerOverlayService {
    private spinnerTopRef: OverlayRef;
    private spin$: Subject<number> = new Subject();

    constructor( private overlay: Overlay ) {

      this.spinnerTopRef = this.overlay.create({
        hasBackdrop: true,
        positionStrategy: this.overlay.position()
          .global()
          .centerHorizontally()
```

```
        .centerVertically()
    });

    this.spin$
      .asObservable()
      .pipe(
        scan((acc, next) => {
          if (!next) {
            return 0;
          }
          return (acc + next) >= 0 ? acc + next : 0;
        }, 0),

        map(val => val > 0),
        distinctUntilChanged()
      )
      .subscribe(
        (res) => {
          if (res) {
            this.spinnerTopRef.attach(new ComponentPortal(MatSpinner));
          } else if (this.spinnerTopRef.hasAttached()) {
            this.spinnerTopRef.detach();
          }
        }
      );
  }

  show = () => this.spin$.next(1);
  hide = () => this.spin$.next(-1);
  reset = () => this.spin$.next(0);
}
```

Because we are using the `SpinnerEffects` class to manage the spinner state, we need to inject the `SpinnerOverlayService` into it:

```
import { Injectable } from '@angular/core';
import { createEffect } from '@ngrx/effects';
import { tap, map } from 'rxjs/operators';
import * as fromStore from '../reducers';
import { Store, select } from '@ngrx/store';
import { SpinnerOverlayService } from '../../services/spinner-overlay.service';

@Injectable()
export class SpinnerEffects {
```

```
    constructor(
      private store$: Store<fromStore.AppState>,
      private spinner: SpinnerOverlayService) {}

    handleSpinner$ = createEffect(
      () =>
       this.store$.pipe(
         select(fromStore.getSpinnerShow),
         tap( isShow =>
           isShow ? this.spinner.show() : this.spinner.hide()
       )),
       { dispatch: false }

      );
}
```

We are toggling the display of the loading indicator by dispatching the `show()` and `hide()` actions to the store.

The Error State

The way to properly define the error state often dependents on the question, where do we put the error information. The component may need to display the error message itself or have some conditional logic to show or hide some parts of the DOM based on the existence of an error. In our application, we are using the Angular Material Snackbar to display error notifications and we handle this within an effect. We may need also to manage the error logic on a component level, so we placed the error state in the NgRx global store.

The error action is defined as follows:

```
import { createAction, props } from '@ngrx/store';

export const errorMessage = createAction('[Error] Error Message',
props<{ errorMsg: string }>());
```

And it is handled by the side effect:

```
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { tap, map } from 'rxjs/operators';
import { ErrorActions, SnackBarActions } from '../actions';
import { AppearanceColor, SnackBarInterface } from '../../models';

@Injectable()
export class ErrorEffects {
     constructor(private readonly actions$: Actions) {}

     handleError$ = createEffect(
       () =>
         this.actions$.pipe(
         ofType(ErrorActions.errorMessage),
         map(action => action.errorMsg),
         tap(errorMsg => console.error('Got error:', errorMsg)),
         map(errorMsg => {
           const msg: SnackBarInterface = {
             message: errorMsg,
             color: AppearanceColor.Error
           };
           return SnackBarActions.open({payload: msg});
         } )
       )
     );
}
```

The Snackbar Effect

In this section, we are going to look at how we can implement the snackbar notification state. We start by defining a custom snackbar component that could receive the notification data.

```
<mat-card>
    <mat-card-subtitle>SnackBar Message</mat-card-subtitle>
    <mat-card-content>
        <p [ngClass]="{
            'accent-color': data.color == 'accent',
            'primary-color': data.color == 'primary',
            'red-color': data.color == 'warn'
            }">
            {{data.message}}
        </p>
    </mat-card-content>
</mat-card>
```

The snackbar state is managed by dispatching the following action to the store:

```
import { createAction, props } from '@ngrx/store';
import { SnackBarInterface } from '../../models';

export const open = createAction('[SnackBar] Open', props<{ payload: SnackBarInterface }>());
```

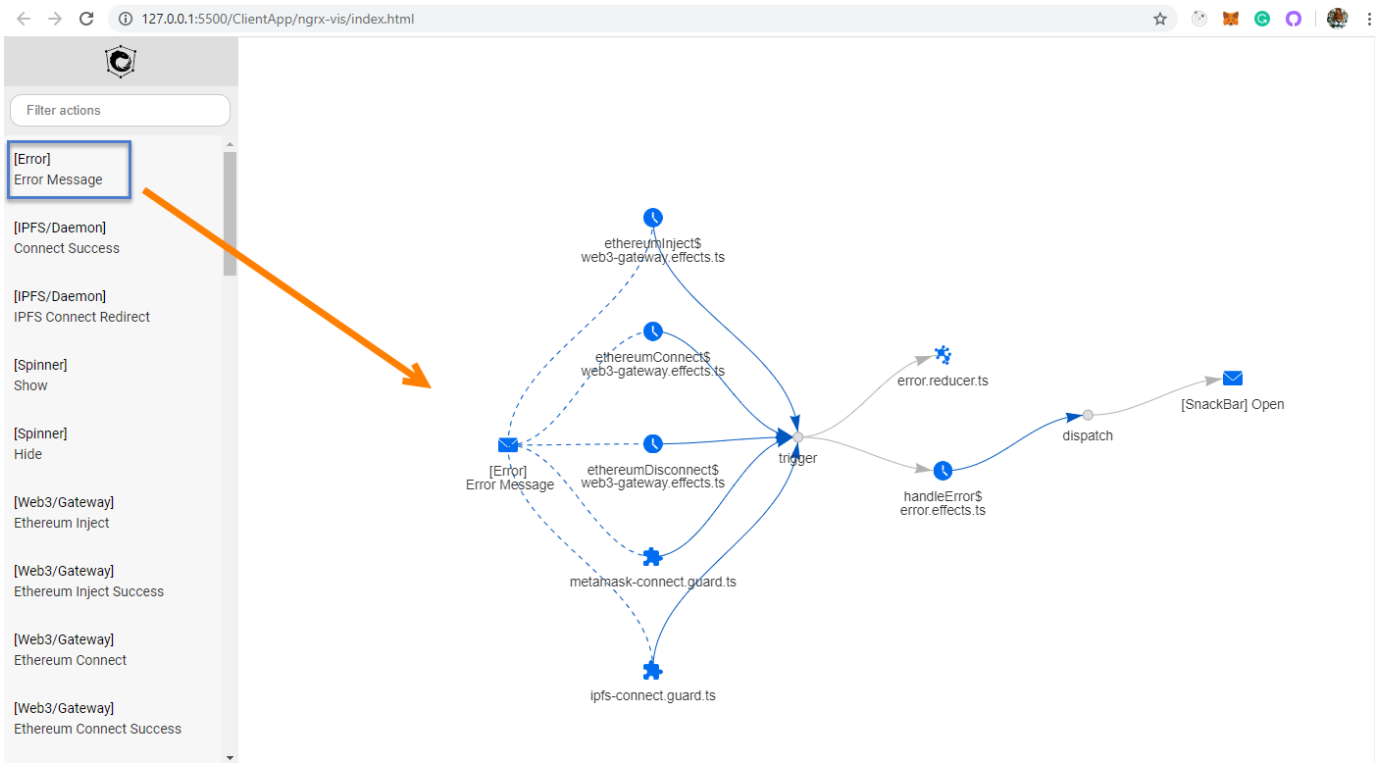
Then, it is handled by the Effect:

```
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { tap, map } from 'rxjs/operators';
import { SnackBarActions } from '../actions';
import { SnackBarService } from '../../services/snack-bar.service;

@Injectable()
```

```
export class SnackBarEffects {

    static readonly SNACKBAR_DELAY: number = 7000;
    constructor(private readonly actions$:
      Actions, private notifier: SnackBarService) {}

    openSnackbar$ = createEffect(
      () =>
        this.actions$.pipe(
          ofType(SnackBarActions.open),
          map(action => action.payload),
          tap(payload => this.notifier.show(payload)
        ),
      { dispatch: false }
    );
}
```

To visualize the flow of actions in our NgRx store, we can use a very handy [NgRx Vis] tool. With NgRx Vis, we get in which file action is created, where it is dispatched, and how it is processed by an effect or a reducer. For example, below is the snapshot showing the flow of the error action in our application.

```
const errorMessage = createAction('[Error] Error Message', props<{ errorMsg: string }>())
```



Part 8: Managing Connection to MetaMask Ethereum Wallet

To interact with Ethereum blockchain, we'll be using the MetaMask web browser extension. MetaMask is a digital wallet and used to store, send, and receive Ethereum and Ethereum based tokens. With MetaMask comes a build-in Web3 provider. MetaMask injects its Web3 provider into the browser in the global JavaScript object `window.ethereum`.

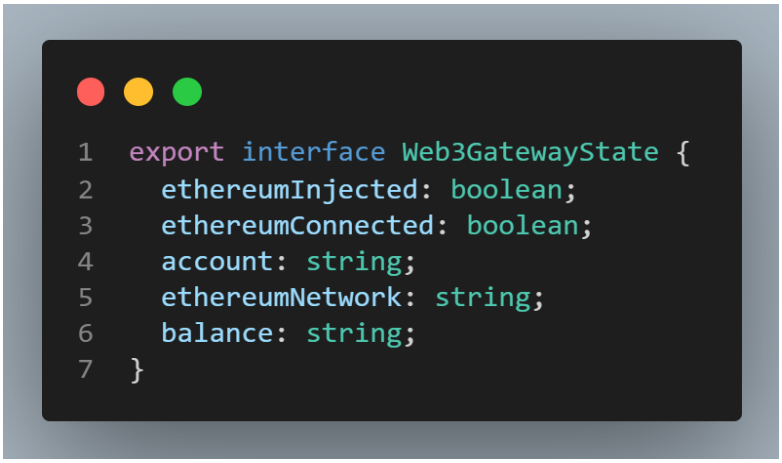
Let's define a factory InjectionToken `EthereumProviderToken` and instruct it to inject the MetaMask Web3 provider `windows.ethereum`.

```
import { InjectionToken} from '@angular/core';

export const EthereumProviderToken = new InjectionToken(
    'Ethereum provider',
    {
        providedIn: 'root',
        factory: () => (window as any).ethereum
    }
);
```

We will use the ethers.js library, which wraps the MetaMask Web3 provider and exposes the ethers.js Provider API. The ethers.js library has been developed by Richard Moore (RicMoo). The approach we use closely resembles the idea discussed in the blog written by GrandSchtroumpf. To install ethers.js run the following command:

```
npm install ethers --save
```

To effectively manage the connection between MetaMask wallet and the user's Ethereum account(s), we introduce a separate slice of the global root state defined by the interface `Web3GatewayState`:



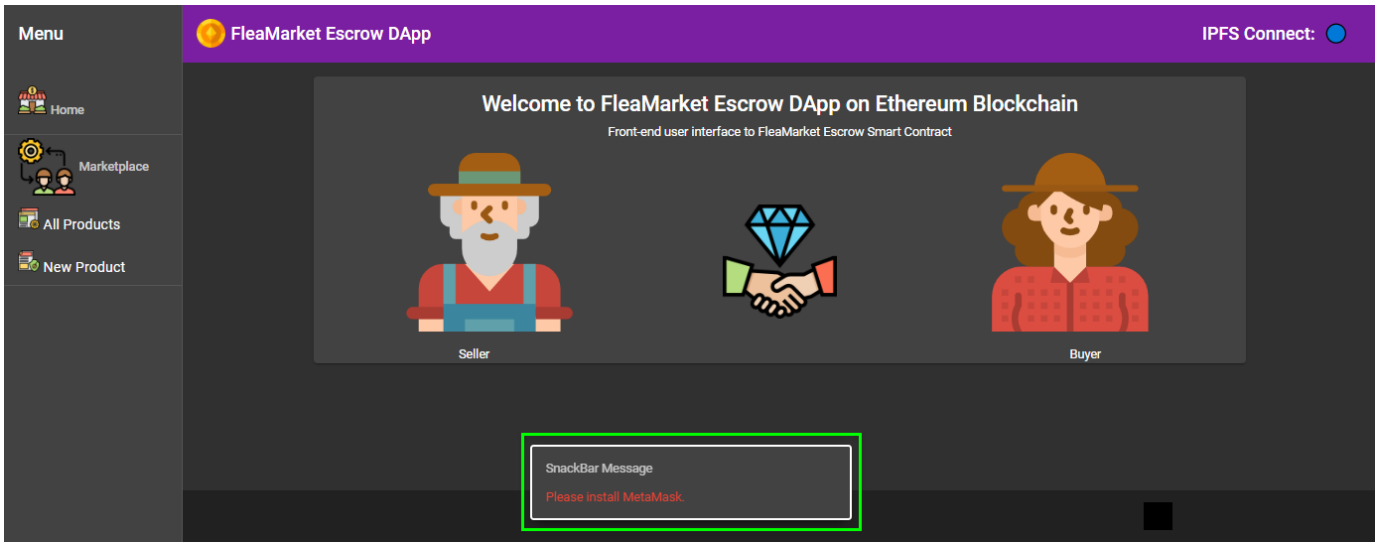
```
export interface Web3GatewayState {
  ethereumInjected: boolean;
  ethereumConnected: boolean;
  account: string;
  ethereumNetwork: string;
  balance: string;
}
```

The `ethereumInjected` flag is representing whether the user has MetaMask installed. The other boolean property `ethereumConnected` defines whether the user is connected to MetaMask, in other words, if MetaMask has access to a user's Ethereum account.

To ensure that the user has MetaMask installed, register the `ethereumInject$` effect that is triggered on the `ROOT_EFFECTS_INIT` lifecycle hook upon bootstrapping the app:

```
ethereumInject$ = createEffect(() =>
    this.actions$.pipe(
    ofType(ROOT_EFFECTS_INIT),
    map(() => {
      if (!this.ethProvider || !this.ethProvider.isMetaMask) {
        return ErrorActions.errorMessage({ errorMsg:
          `Please install MetaMask.` });
      }
      return Web3GatewayActions.ethereumInjectSuccess();
     })
    )
);
```

It will notify the user to install the MetaMask extension if it is not detected.



To access user accounts and initiate account-blockchain related calls such as sending transactions or signing messages, MetaMask requires that we call the `ethereum.send('eth_requestAccount')` method from the new MetaMask inpage provider APIs. MetaMask windows will popup to encourage the user to initiate a connection attempt to retrieve the user's Ethereum account. Based on MetaMask recommendations, in DApps we should initiate a connection request in response to a direct user action, and not on page load. We manage this in the `ethereumConnect$` store effect by listening for the `ethereumConnect()` action.

```
ethereumConnect$ = createEffect(() =>
    this.actions$.pipe(
      ofType(Web3GatewayActions.ethereumConnect),
      exhaustMap(() => {
        // This is equivalent to ethereum.enable()
        // return list of user account
        // currently only ever one: accounts[0]
        return from(this.ethProvider.send('eth_requestAccounts')).pipe(
          map((ethAccounts: any[]) => {
            if (ethAccounts.length === 0) {
              return ErrorActions.errorMessage({ errorMsg:
                `Can't get any user accounts` });
            }
            console.log(`Ethereum provider has been granted access
            to the account:`, ethAccounts[0]);
            return Web3GatewayActions.ethereumConnectSuccess();
          }),
          // User denied account access
          catchError((err: Error) => of(this.handleError(err),
            SpinnerActions.hide()))
```

```
      );
    })
  )
);

private handleError(error: Error) {
  const friendlyErrorMessage = serializeError(error).message;
  return ErrorActions.errorMessage({ errorMsg: friendlyErrorMessage });
}
```

We introduce the CanLoad `MetaMaskConnectGuard`, which controls the route resolution process and prevents loading of future modules

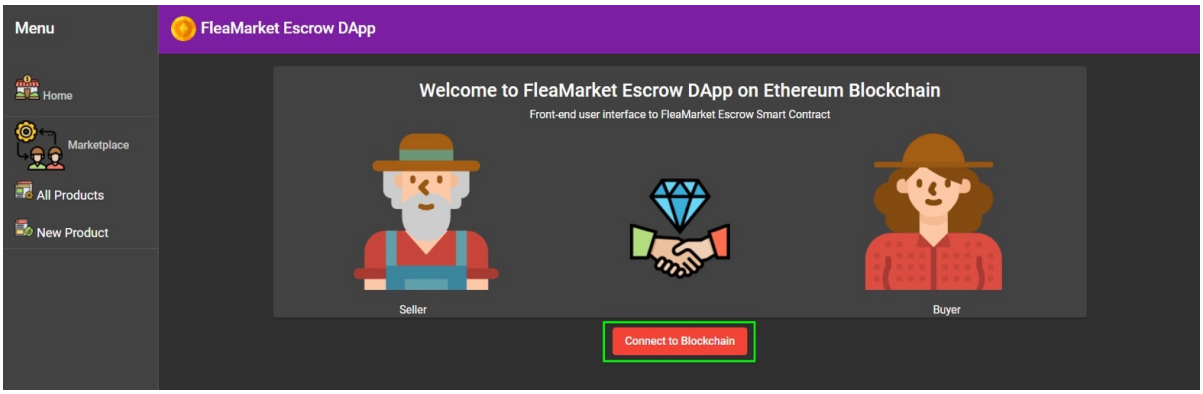
```
@Injectable({
  providedIn: 'root'
})
export class MetaMaskConnectGuard implements CanLoad {
  constructor(private store: Store<fromRoot.AppState>) {}
  canLoad(): Observable<boolean> {
    return this.store.pipe(
      select(fromRoot.getEthereumConnected),
      tap(connected => {
       if (!connected) {
         this.store.dispatch(fromRoot.ErrorActions.
           errorMessage({ errorMsg: `Unable to detect Ethereum account` }));
         this.store.dispatch(fromRoot.Web3GatewayActions.ethereumConnectRedirect());
         return false;
       }
       return true;
      }),
      take(1)
    );
  }
}
```

The guard is observing the `ethereumConnected` property of the state. If this value is false, the guard will dispatch the `ethereumConnectRedirect()` action to broadcast the request to redirect the user to the home page specified by the root path `'/'`.

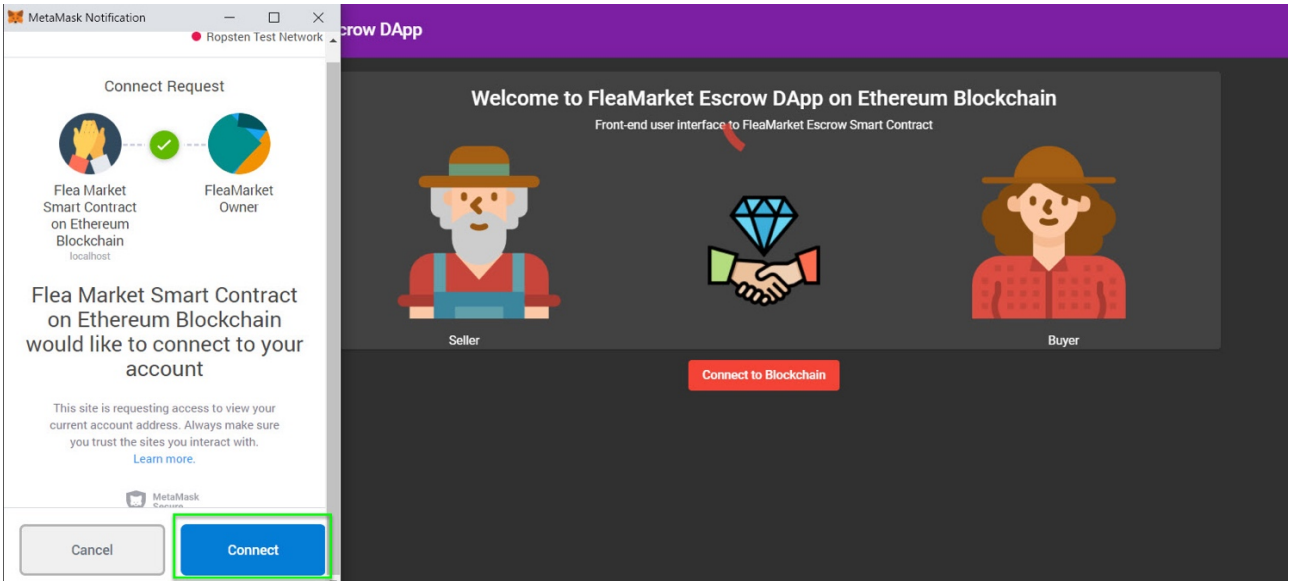
```
 connectRedirect$ = createEffect(
      () =>
        this.actions$.pipe(
          ofType(Web3GatewayActions.ethereumConnectRedirect),
```

```
          tap(_ => {
            this.router.navigate(['/']);
          })
        ),
      { dispatch: false }
);
```

The user has an option to click on the 'Connect to Blockchain' button that becomes active when the value `ethereumConnected` is false.



The click event will dispatch the `ethereumConnect()` action to the `ethereumConnect$` effect and trigger the popup windows from MetaMask to ask for permission to access the user account:



After we've successfully set up the connection with the Ethereum blockchain, we can leverage the use of the NgRx store to display the user account information such as the selected account address, the

blockchain network, and the account balance. We just need to dispatch the `ethereumConnectSuccess()` action to the Store. In the `Web3GatewayEffects` class, we define the `getAccountInfo$` effect that listens for this action and maps it into the other three actions that are handled by the dedicated effects:

```
getAccountInfo$ = createEffect(() =>
    this.actions$.pipe(
      ofType(Web3GatewayActions.ethereumConnectSuccess),
      switchMap(() => {
        return [Web3GatewayActions.getNetwork(),
          Web3GatewayActions.getAccount(), Web3GatewayActions.getBalance()];
      })
    )
);

getAddress$ = createEffect(() =>
    this.actions$.pipe(
      ofType(Web3GatewayActions.getAccount),
      switchMap(() =>
        this.web3ProviderSrv.getSelectedAddress().pipe(
          map((address: string) => Web3GatewayActions.accountSuccess({ address })),
          catchError((err: Error) => of(this.handleError(err)))
        )
      )
    )
);

getBalance$ = createEffect(() =>
    this.actions$.pipe(
      ofType(Web3GatewayActions.getBalance),
      switchMap(() =>
        this.web3ProviderSrv.getBalance().pipe(
          map((balance: string) =>
            Web3GatewayActions.balanceSuccess({ balance })
          ),
          catchError((err: Error) => of(this.handleError(err)))
        )
      )
    )
);

getNetwork$ = createEffect(() =>
    this.actions$.pipe(
```

```
      ofType(Web3GatewayActions.getNetwork),
      switchMap(() =>
        this.providerSrv.getNetwork().pipe(
          map((network: string) =>
            Web3GatewayActions.networkSuccess({ network })
          ),
          catchError((err: Error) => of(this.handleError(err)))
        )
      )
    )
);
```

In the `Web3GatewayEffects` constructor, we inject the `EthersWeb3ProviderService`, which interacts with ethers.js APIs and allows us to retrieve the address, balance, and network of the current account:

```
import { Injectable, Inject } from '@angular/core';
import { ethers, utils, Signer } from 'ethers';
import { Observable, from } from 'rxjs';
import { map, tap } from 'rxjs/operators';
import { EthereumProviderToken } from '../services/tokens';

@Injectable({providedIn: 'root'})
export class EthersWeb3ProviderService {

    constructor(private provider: EthersWeb3Provider) {
    }
    // There is only ever up to one account in MetaMask exposed
    public getSelectedAddress(): Observable<string> {
      const web3Provider:ethers.providers.JsonRpcProvider =
        new ethers.providers.Web3Provider(this.ethProvider);
      const signer:Signer = web3Provider.getSigner();

      return from(signer.getAddress()).pipe(
        tap(address => console.log('address', address))
      );
    }

    public getNetwork(): Observable<string> {
      const web3Provider:ethers.providers.JsonRpcProvider =
        new ethers.providers.Web3Provider(this.ethProvider);

      return from(web3Provider.getNetwork()).pipe(
```

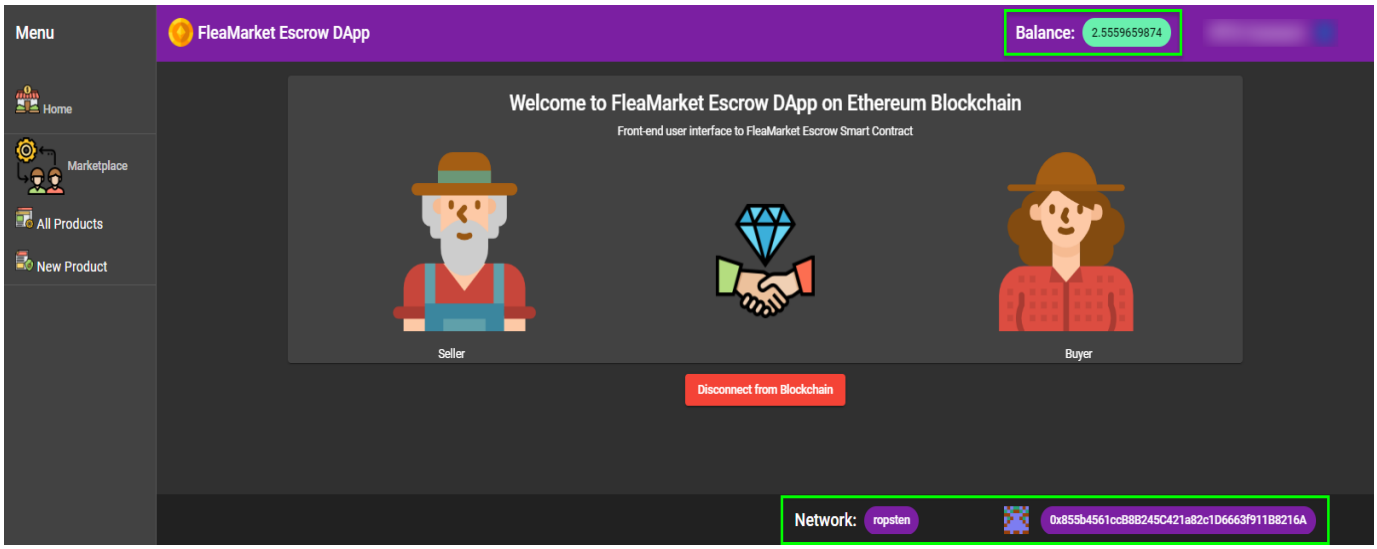
```
      map(network => network.name)
      tap(name => console.log(`network name: ${name}`))
    );
  }

  public getBalance(): Observable<string> {
    const web3Provider:ethers.providers.JsonRpcProvider =
      new ethers.providers.Web3Provider(this.ethProvider);
    const signer:Signer = web3Provider.getSigner();

    return from(web3Provider.getBalance(signer.getAddress())).pipe(
      tap(weiBalance => console.log('wei balance', weiBalance)),
      map(weiBalance => utils.formatEther(weiBalance)),
      tap(balance => console.log('eth balance', balance)),
    );
  }
}
```

Each external call to Ethers API returns a `Promise` that holds the result of the API method once the `Promise` is resolved. We use the RxJS `from()` operator to convert `Promise` to an observable. In the effect, we use the `map()` operator to flatten the observable returned from the service and dispatch a new action on success. The action is dispatched to the Store where it is handled by the corresponding reducer to update the state as needed.

In the `NavComponent`, we use the selector and the async pipe functions to retrieve the account, balance, and network name values from the global state.

We also took the opportunity to use the `ethereum-blockies` library to make Ethereum addresses looks more user-friendly.

Essentially, once the user confirms the connection to the blockchain, the component's selectors will emit the current account information and display it on the toolbar.



We would like also to trigger the reload application on an account change in MetaMask. That can be done with the Web3 provider `accountsChanged` event handler. One of the powerful features of Effects is that it can observe streams other than the `action$` stream. It could be from Promises or other observable streams. Using the RxJS `fromEvent()` function we can build up an observable stream of the MetaMask's Web3 provider events. Here is the effect that is listening for the `accountsChanged` event:

```
accountWatcher$ =
    !!this.ethProvider ? fromEvent(this.ethProvider, 'accountsChanged').pipe(
       withLatestFrom(this.store$.pipe(select(fromStore.getAccount))),
       /* we only want to refresh the browser when:
       - we logout from MetaMask (accounts.length == 0)
       - when we switch account to a different account
       (!!currentAccount && currentAccount !== accounts[0])
       */
       filter(([accounts, currentAccount]) => {
```

```
          if ((accounts as any).length === 0)
            return true;
          /*
          I noticed that using ethers.js it returns account in the hex string like this
          0xd64d1cc32225bd5815cfa7a0b8a6aa46e0ef1285
          but from the event 'accountsChanged' it return the same account hex string like this:
          0xd64D1cc32225bD5815cFA7A0B8a6aa46e0eF1285
          !Notice the capital letters. So we should take care of this situation
          */
          const curAdd = currentAccount ? utils.getAddress(currentAccount) : currentAccount;
          const newAdd = accounts[0] ? utils.getAddress(accounts[0]) : accounts[0];

          if (!!curAdd && (curAdd !== newAdd)) {
            return true;
          }
          return false;
        }),
        map(([accounts, currentAccount]) => {
          this.document.location.reload();
        })
) : of(1);

accountChanged$ = createEffect(
    () => this.accountWatcher$,
    { dispatch: false }
);
```

The `accountsChanged` event will return the updated array of the currently available accounts. In our case, when the user switches an Ethereum account, we ask the browser to reload the app, which then resets the entire NgRx state.

Part 9: Connecting to the IPFS peer running on Infura

In this section, we'll focus on the process of setting up the communication with the IPFS node running on Infura. We will take advantage of the IPFS network to upload and store image files of products for sale. The `SafeRemotePurchase` contract defines a state variable `ipfsImageHash` that stores the product image file's IPFS hash value.

The important advantage of using IPFS is that it is guaranteed that both the file and its corresponding hash code that is stored on the IPFS network are immutable. This means that it is impossible to change an image file without changing its hash.

To communicate with a remote IPFS node hosted on the Infura network we need to install the IPFS HTTP API client library:

```
npm install ipfs-http-client
```

To successfully compile a project which includes this cryptographic-based library, we need to tell Angular to include in compilation some cryptographic `node` modules. We can do this by appending the default built-in angular webpack with a custom webpack config file `custom-webpack.config.js`:

```
const path = require('path');

module.exports = {
  node: {
    crypto: true,
    path: true,
    os: true,
    stream: true,
    buffer: true,
  },

};
```

To deal with a custom webpack, we need to install the custom webpack builder:

```
npm install --save-dev @angular-builders/custom-webpack
```

Then specify the custom builder in the `angular.js` file.

Let's define a new `InjectionToken` with the factory function that instantiates the global object `IpfsHttpClient`. This object is responsible for the connection to the IPFS node that runs on Infura:

```
import { InjectionToken } from '@angular/core';
import IpfsHttpClient from 'ipfs-http-client';

export const ipfsToken = new InjectionToken('The IPFS Token', {
  providedIn: 'root',

  // safe to put in the 'root',as it
  // will not throw any error until we call IPFS API
  factory: () => new IpfsHttpClient({
    host: 'ipfs.infura.io',
    port: '5001',
    protocol: 'https'
  })

});
```

Next, we inject this token into the IpfsDaemonService class that will handle the logic of communication with the IPFS Infura network including the uploading and retrieving files:

```
import { Injectable, Inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, from, defer } from 'rxjs';
import { switchMap, map, tap } from 'rxjs/operators';
import { ipfsToken } from './tokens';
import { Buffer } from 'buffer';

@Injectable({
  providedIn: 'root'
})
export class IpfsDaemonService {
    constructor(@Inject(ipfsToken) private ipfs,
      private httpClient: HttpClient) {}

    public getId(): Observable<string> {
      return from(this.ipfs.id()).pipe(
        tap((res: any) =>
          console.log(`IPFS node id object: ${JSON.stringify(res)}`)
        ),
        map(res => res.id)
      );
    }

    public getVersion(): Observable<string> {
      return from(this.ipfs.version()).pipe(
        tap((res: any) =>
          console.log(`IPFS node version object: ${JSON.stringify(res)}`)
        ),
        map(res => res.version)
      );
    }

    public addFile(file: File): Observable<string> {

      const data = {
       path: file.name,
       content: file
      };
      const options = {
        progress: (prog) => console.log(`progress report: ${prog}`)
      };
      const sz = defer(async () => {
        let res;
        for await (const result of this.ipfs.add(data, options)) {
```

```
        res = result;
      }
      return res;
    });

    return sz.pipe(
      tap((res: any) =>
        console.log(`IPFS node response json: ${JSON.stringify(res)}`)
      ),
      map((res: any) => res.cid.toString())
    );
  }

  public getFile = (hash: string): Observable<Blob> =>
    const sz = defer(async () => {
      const chunks = []
      for await (const chunk of this.ipfs.cat(hash)) {
        chunks.push(chunk)
      }
      const buff = Buffer.concat(chunks);
      return buff;
    });

    return sz.pipe(
      switchMap((buffer: Buffer) => {
        const byteString = buffer.toString('base64');
        const url = `data:application/octet-stream;base64,${byteString}`;

        return this.httpClient.get(url, {
          responseType: 'blob'
        });
      }))
  }
}
```

In the `getVersion()` method of the service, we validate the connection to the IPFS node by retrieving its `version` object. The `addFile()` method takes in a parameter `file: File` and passes it to the IPFS API `add()` method to import the image file into IPFS. Upon successful execution, the method returns an async iterable that yields objects describing the added data. We then use the RxJS `defer()` function to emits the returned value from the async iterable. We pipe this value to the `map()` operator to retrieve the CID object that describes the hash value of the stored file. The `getFile()` method is for reading image files from IPFS. It takes the hash param and calls the IPFS API `cat()` function. The `cat()` function

returns an async iterable that yields the `Buffer` object with the contents of the file that is addressed by the IPFS hash. We encode the raw buffer stream to a base64-encoded string. Then we wait for the image blob to be returned by using the `HttpClient` with the `responseType` option set to `'blob'`.

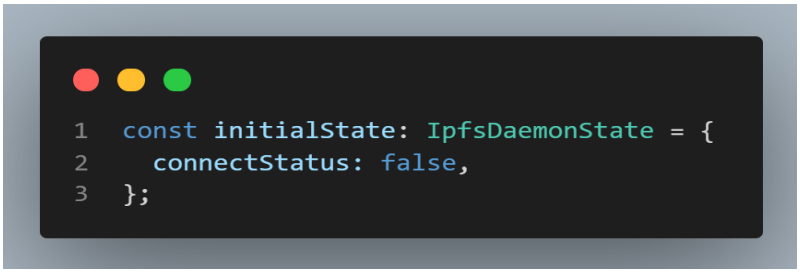
Let's inject this service into a new root Effects class that will help us manage the connection to an IPFS node.

```
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect, ROOT_EFFECTS_INIT } from '@ngrx/effects';
import { serializeError } from 'serialize-error';
import { of } from 'rxjs';
import { switchMap, map, tap, catchError } from 'rxjs/operators';
import { IpfsDaemonService } from '../../services/ipfs-daemon.services';
import { IpfsDaemonActions, ErrorActions } from '../actions';
import { Router } from '@angular/router;

@Injectable()
export class IpfsDaemonEffects {
    constructor(
      private ipfsSrv: IpfsDaemonService,
      private readonly actions$: Actions,
      private router: Router,
    ) { }

    onConnect$ = createEffect(
      () =>
        this.actions$.pipe(
          ofType(ROOT_EFFECTS_INIT),
          switchMap(() =>
            this.ipfsSrv.getVersion().pipe(
              tap(version => console.log(`IPFS node version: ${version}`)),
              map(_ => IpfsDaemonActions.connectSuccess()),
              catchError((err: Error) => of(this.handleError(err)))
            )
          )
        )
    );

    connectRedirect$ = createEffect(
      () =>
        this.actions$.pipe(
          ofType(IpfsDaemonActions.ipfsConnectRedirect),
          tap(_ => {
            this.router.navigate(['/']);
```

```
        })
      ),
    { dispatch: false }
  );

  private handleError(error: Error) {
    const friendlyErrorMessage = serializeError(error).message;
    return ErrorActions.errorMessage({ errorMsg: friendlyErrorMessage });
  }
}
```

In the `onConnect$` effect, we use the `ROOT_EFFECTS_INIT` lifecycle hook, which is triggered after all the root effects have been added upon bootstrapping the app.

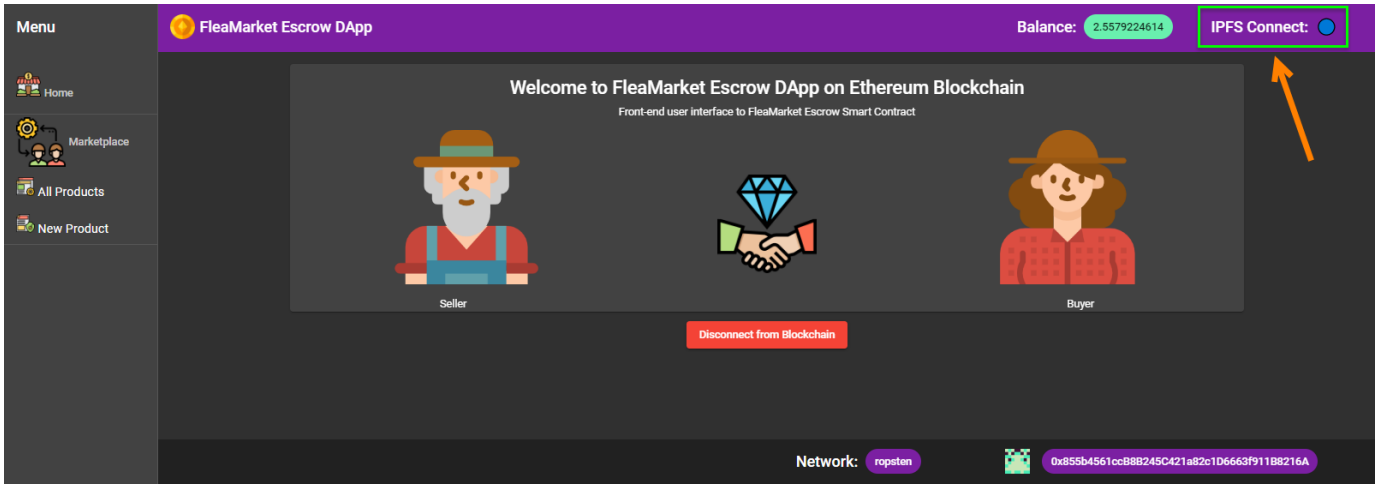
After successful verification that the IPFS node is ready, we dispatch the `connectSuccess()` action back to the Store where the reducer will update the global state property `connectStatus` to `true`:



```
const initialState: IpfsDaemonState = {
  connectStatus: false,
};
```

This change will be automatically observed by the corresponding selector from the store reducer, and the async pipe from the `Nav` component will update the IPFS node connection status on the toolbar.

-



We also use the CanLoad `IpfsConnectGuard` to ensure that a valid connection to the IPFS network is in place before the user can navigate to other parts of the application.

```
canLoad(): Observable<boolean> {
  return this.store.pipe(
    select(fromRoot.getIpfsConnectStatus),
    tap(connected => {
      if (!connected) {
        this.store.dispatch(fromRoot.ErrorActions.errorMessage({ errorMsg:
          `Unable to detect IPFS node.` }));
        this.store.dispatch(fromRoot.IpfsDaemonActions.ipfsConnectRedirect());
        return false;
      }
      return true;
    }),
    take(1)
  );
}
```

Part 10: Managing IPFS Image Upload and Retrieval

In this section, we'll focus on coding the functionality responsible for storing and retrieving image data hosted on the IPFS node. To keep our NgRx store organized, we bundle this functionality into a separate feature module `MarketPlaceModule`. The module is lazy-loaded using a new dynamic `import()` syntax and controlled by two guards: `MetaMaskConnectGuard` and `IpfsConnectGuard`:

```
{
  path: 'market-place',

    loadChildren: () => import('./market-place/market-place.module').then(mod => mod.MarketPlaceModule),

      canLoad: [guards.MetaMaskConnectGuard, guards.IpfsConnectGuard],
}
```

Here's our component `NewPurchaseContractComponent` where we wire up the logic to create a new product for sale:

```
import { Component, ViewChild, ElementRef, OnInit, OnDestroy }
   from '@angular/core';
import { FormBuilder, FormGroup, Validators, AbstractControl }
   from '@angular/forms';
import { MatDialog, MatDialogConfig } from '@angular/material';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';
import { utils } from 'ethers';
import { Store, select } from '@ngrx/store';
import * as fromPurchaseContract from '../../store/reducers';
import { IpfsImageActions, PurchaseContractActions }
   from '../../store/actions';
import { FileUploadStatus }
```

```
    from '../../store/reducers/ipfs-product-image.reducer'
import { ShowIpfsImageComponent }
    from '../../components/show-ipfs-image/show-ipfs-image.component';

// Utility function. Check if string is less then 32 bytes.
// This is required by the smart contract
function bites32StringValidator(control: AbstractControl):
    { [key: string]: any } | null {
    let pathTest = false;
    try {
      utils.formatBytes32String(control.value);
      pathTest = true;
    } catch (error) {
      // console.log('bites32StringValidator', error)
    }
    return !pathTest ? {
      forbiddenKey: {
        value: control.value
      }
    } : null;
}

@Component({
    selector: 'app-new-purchase-contract',
    templateUrl: './new-purchase-contract.component.html',
    styleUrls: ['./new-purchase-contract.component.css']
})
export class NewPurchaseContractComponent implements OnInit, OnDestroy {

    @ViewChild('file') fileControl: ElementRef;
    fileBlob: File;
    fileContent: ArrayBuffer;
    ipfsHash$: Observable<string>;
    uploadStatus$: Observable<FileUploadStatus>;
    private readonly IMAGE_PATTERN: RegExp = /^.+\.(png|jpg|jpeg|gif|png)$/;
    commissions: string[] = ['2.0', '2.5', '3.0', '3.5', '4.0'];

    constructor(
      private store$: Store<fromPurchaseContract.AppState>,
      private formBuilder: FormBuilder,
      public dialog: MatDialog
    ){ }

    frmGroup: FormGroup = this.formBuilder.group({
      productKey: ['', [Validators.required, bites32StringValidator]],
```

```
  description: ['', Validators.required],
  etherValue: ['', [Validators.required,
                    Validators.pattern(/^\d+(\.\d{1,3})?$/)]],
  commission: ['', Validators.required],
  fileArg: ['', [Validators.required, Validators.pattern(this.IMAGE_PATTERN)]],
  ipfsHash: ['', Validators.required] // to hold ipfsHash value
});

ngOnInit() {
  this.uploadStatus$ =
    this.store$.pipe(select(fromPurchaseContract.getIpfsUploadStatus));
  this.ipfsHash$ = this.store$.pipe(
    select(fromPurchaseContract.getIpfsHash),
    tap(value => this.frmGroup.get('ipfsHash').patchValue(value))
  );
}

formControl = (name: string) => this.frmGroup.get(`${name}`);
required = (name: string) =>
  this.formControl(name).hasError('required') && this.formControl(name).touched

invalidPattern = (name: string) =>
  this.formControl(name).hasError('pattern') && this.formControl(name).dirty

invalid32BytesKey = (name: string) =>
  this.formControl(name).hasError('forbiddenKey')
    && this.formControl(name).dirty

requiredFile = (name: string) => this.formControl(name).hasError('required');
invalidPatternFile = (name: string) =>
  this.formControl(name).hasError('pattern');

// here is the way to emulate the click on the file input control
selectFile() {
  this.fileControl.nativeElement.click();
}

onFileChange(event) {
  if (event.target.files && event.target.files.length) {
    this.fileBlob = event.target.files[0];
    this.frmGroup.get('fileArg').patchValue(this.fileBlob.name);

    const reader = new FileReader();
    reader.readAsDataURL(this.fileBlob);
    reader.onload = _ => {
```

```
        this.fileContent = reader.result as ArrayBuffer;
        this.store$.dispatch(IpfsImageActions.reset());
      };
    }
  }

  uploadFile() {
    this.store$.dispatch(IpfsImageActions.uploadImage({ file: this.fileBlob }));
  }

  isPending = (status: FileUploadStatus) => status === FileUploadStatus.Pending;
  isSuccess = (status: FileUploadStatus) => status === FileUploadStatus.Success;
  isError = (status: FileUploadStatus) => status === FileUploadStatus.Error;
  inProgress = (status: FileUploadStatus) => status ===
                  FileUploadStatus.Progress;

  loadImage() {
    const dialogConfig = new MatDialogConfig();
    dialogConfig.width = '460px';
    dialogConfig.disableClose = true;
    dialogConfig.autoFocus = true;

    this.dialog.open(ShowIpfsImageComponent, dialogConfig);
  }

  onCreate(): void {
    const { valid } = this.frmGroup;
    if (valid) {
      const { fileArg, ...model } = this.frmGroup.value;
      this.store$.dispatch(PurchaseContractActions.
                    createPurchaseContract({ payload: model }));
    }
  }

  ngOnDestroy(): void {
     this.store$.dispatch(IpfsImageActions.reset());
  }
}
```

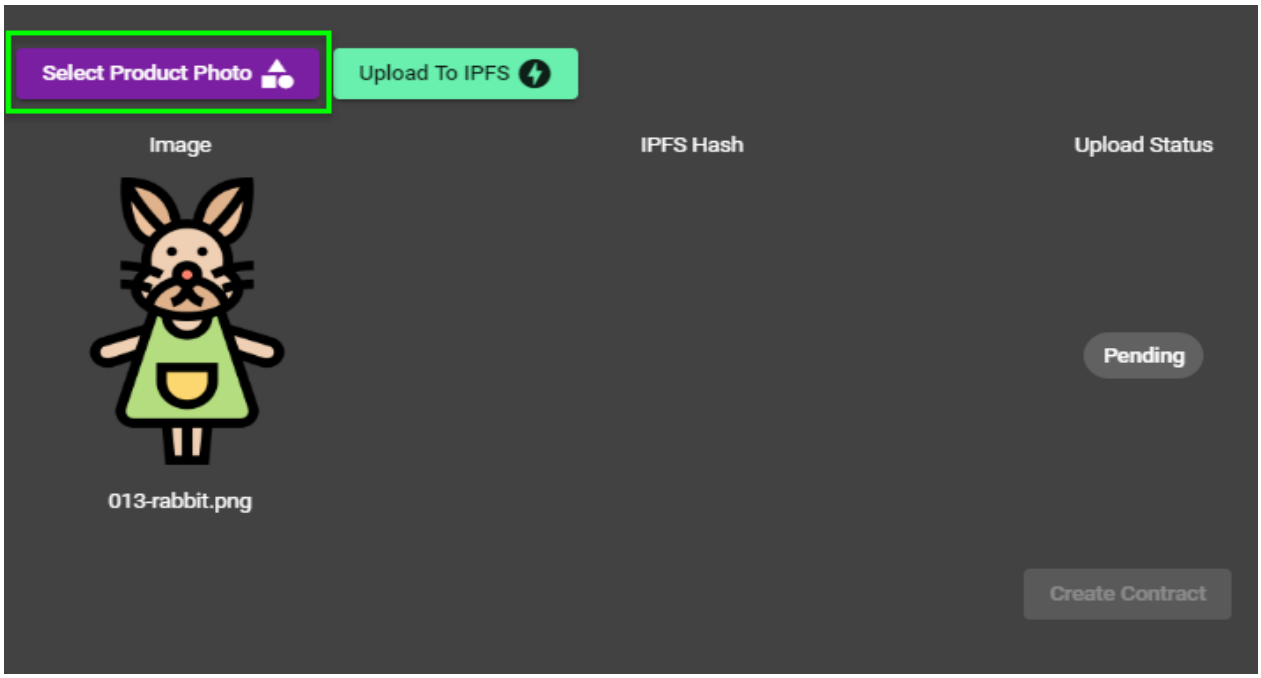
This is a smart component that among other things provides the user interface for:

- The image review
- Uploading the image to IPFS
- Retrieving the image file from IPFS based on its hash code

Let's take a further look at each of these tasks.

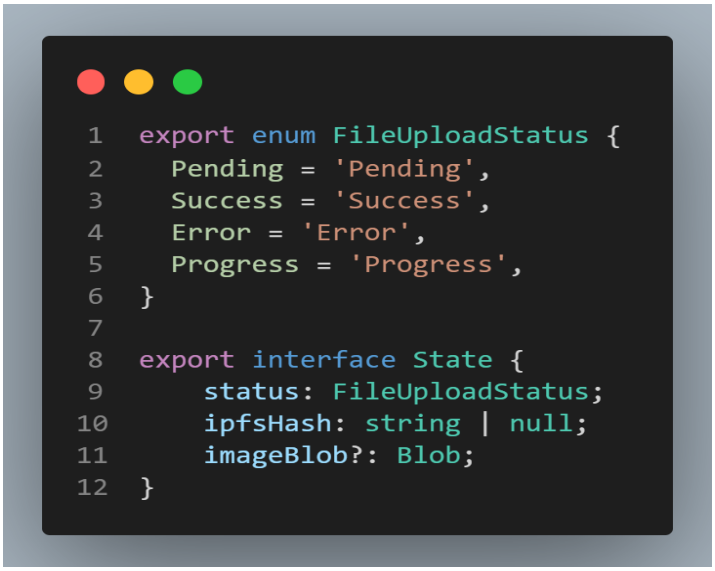
The Image Preview

The component's template includes the button 'Select Product Photo', which lets us preview the image before uploading it to IPFS. To handle the preview logic, we use the angular material reactive form. The image file is stored in the component variable `fileBlob: File`.



Upload Image Files to IPFS

To manage file uploads using NgRx, we create the following state interface:



```
export enum FileUploadStatus {
  Pending = 'Pending',
  Success = 'Success',
  Error = 'Error',
  Progress = 'Progress',
}

export interface State {
    status: FileUploadStatus;
    ipfsHash: string | null;
    imageBlob?: Blob;
}
```

To initiate the upload process, the user needs to click the button labeled 'Upload to IPFS', which becomes active after we choose an image file. This triggers the store action `uploadImage()` that carries the value of the variable `fileBlob: File` in its payload. The action is dispatched to the dedicated store effect `uploadImage$:`

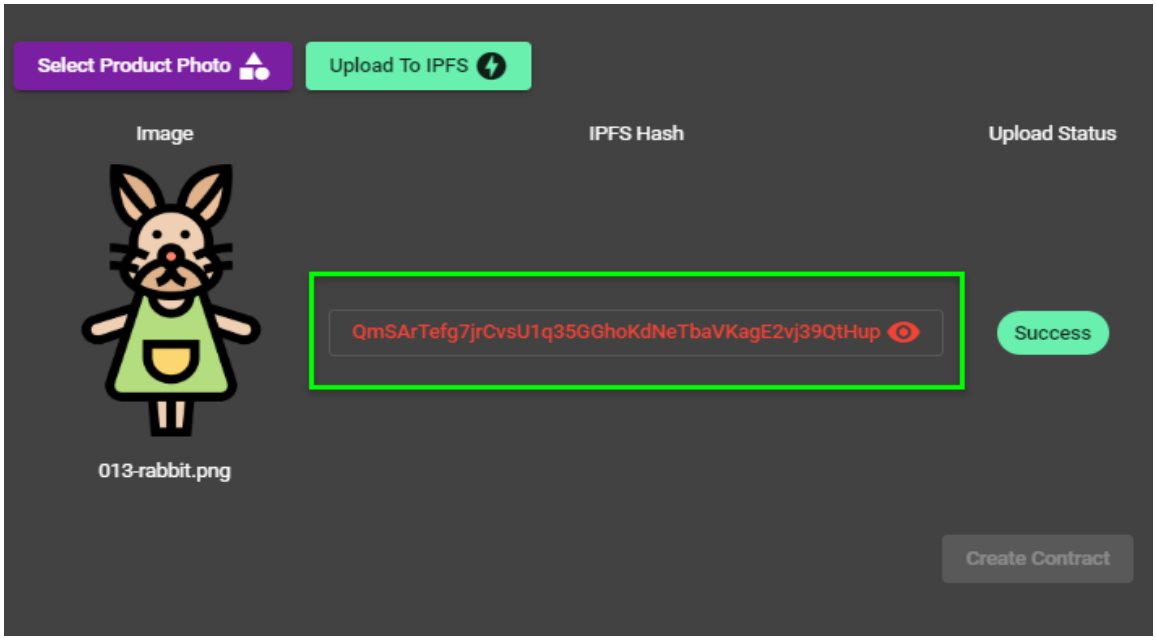
```
uploadImage$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(IpfsImageActions.uploadImage),
        map(action => action.file),
        exhaustMap((file) => {

          return this.ipfsSrv.addFile(file).pipe(
            tap(ipfsHash => console.log(`IPFS file hash: ${ipfsHash}`)),
            map(ipfsHash => IpfsImageActions.uploadImageSuccess({ ipfsHash })),
            catchError((err: Error) =>
              of(this.handleError(err), IpfsImageActions.uploadImageFail())
            )
          );
        })

      ));
```

Let's take a closer look at what this effect is doing. We use the `exhaustMap()` operator to receive a `File` object from the action payload. The advantage of using this RxJS mapping operator is that it will ignore subsequential file upload requests while the current one is still ongoing. We then pass this value to

the `addFile()` method in the `IpfsDaemonService`. Upon successful execution of this method, we use the `map()` operator to project the received hash value to the `uploadImageSuccess()` action.

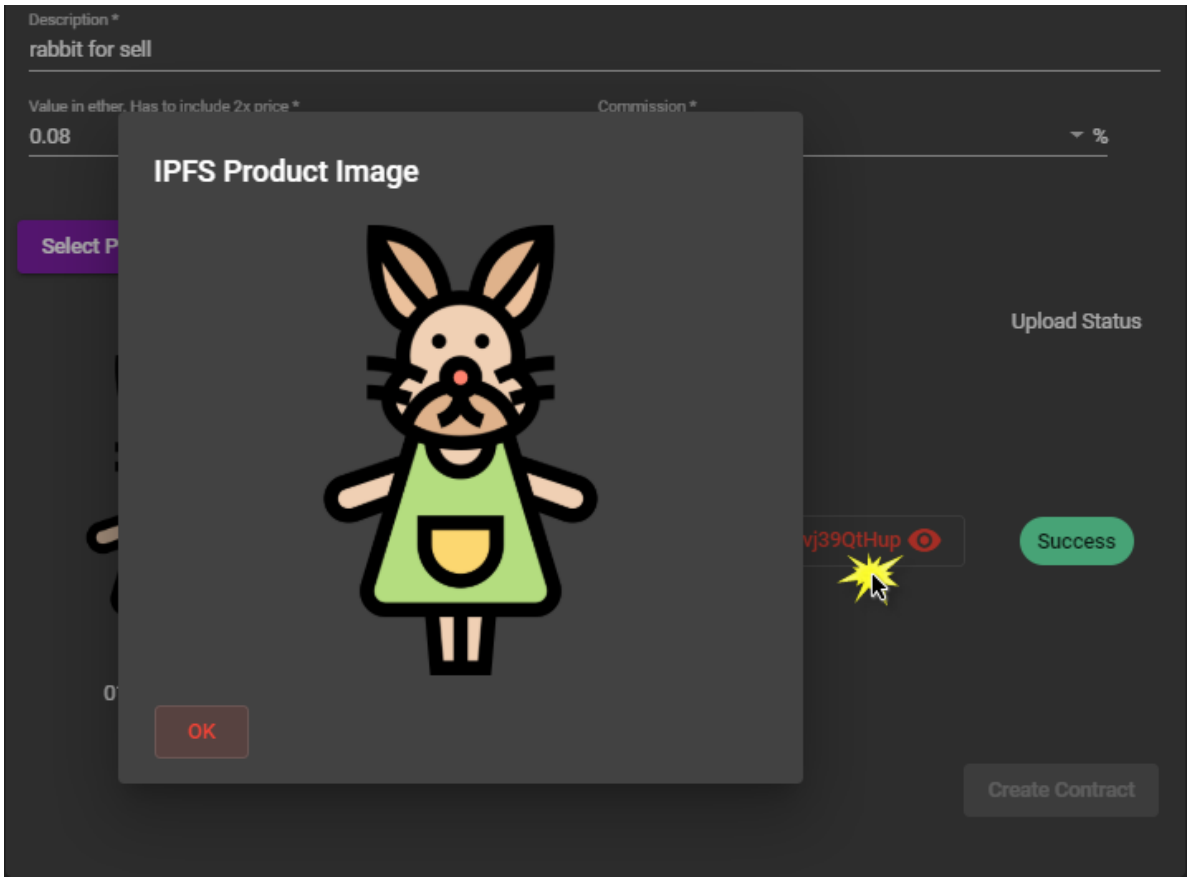
The action will trigger the following chain of events managed by our feature store:

1. The action carrying the IPFS image hash payload is dispatched to the corresponding reducer
2. The reducer will update the state properties `ipfsHash` and `status` with new values
3. The store selectors that observe these state changes will update the component template accordingly



Retrieving Images from IPFS

Once the product image is uploaded on the IPFS network, we can confirm that we can retrieve it using the hash. We have a button in our template component that becomes visible and displays the value of the hash emitted by `ipfsHash$`. When a user clicks on it,



it will activates the material dialog defined in the component `ShowIpfsImageComponent`:

```
import { ChangeDetectionStrategy, Component, Inject, OnInit, ViewChild ,
    ElementRef } from '@angular/core';
import { MatDialogRef } from '@angular/material';
import { windowRefToken } from '../../../core/services/tokens';
import { Observable } from 'rxjs';
import { withLatestFrom, map, tap, filter, take } from 'rxjs/operators';
import { Store, select } from '@ngrx/store';
import * as fromPurchaseContract from '../../store/reducers';
import * as IpfsActions from '../../store/actions/ipfs-product-image.actions';

@Component({
    changeDetection: ChangeDetectionStrategy.OnPush,
    selector: 'app-show-ipfs-image',
    templateUrl: 'show-ipfs-image.component.html',
    styleUrls: ['show-ipfs-image.component.css']
})
export class ShowIpfsImageComponent implements OnInit {
    @ViewChild('ipfsImage') imageRef: ElementRef;
    image$: Observable<Blob>;
```

```
  constructor(
    private store$: Store<fromPurchaseContract.AppState>,
    public dialogRef: MatDialogRef<ShowIpfsImageComponent>,
    @Inject(windowRefToken) private windowRef: Window
  ) { }

  ngOnInit() {
    this.image$ = this.checkStore().pipe(
      tap((blob) =>
        this.imageRef.nativeElement.src = this.windowRef.URL.createObjectURL(blob)
      )
    );
  }

  checkStore(): Observable<Blob> {
    return this.store$.pipe(
      select(fromPurchaseContract.getImageBlob),
      withLatestFrom(this.store$.pipe(select(fromPurchaseContract.getIpfsHash))),
      tap( ([image, ipfsHash]) => {
       if (!image) {
         this.store$.dispatch(IpfsActions.downloadImage({ipfsHash}));
       }
      }),
      map(([image, ipfsHash]) => image),
      filter(image => !!image),
      take(1)
    );
  }
}
```

Inside the component, we declare the method `checkStore()` that returns the blob observable. The method checks the `getImageBlob` selector for the state property `imageBlog`. If this value hasn't set up yet, the method will dispatch the `downloadImage()` action and broadcast it to the effect `downloadImage$`:

```
downloadImage$ = createEffect(
  () =>
    this.actions$.pipe(
      ofType(IpfsImageActions.downloadImage),
      map((action) => action.ipfsHash),
      switchMap((ipfsHash: string) =>
        this.ipfsSrv.getFile(ipfsHash).pipe(
```

```
          map((image: Blob) => IpfsImageActions.downloadImageSuccess({ image })),
          catchError((err: Error) =>
            of(this.handleError(err), IpfsImageActions.downloadImageError())
          )
        )
      )
    )
);
```

The current value of the hash property is pulled from the store and passed as a parameter into the service method `getFile()`. The method gets resolved with the `image: Blob` value retrieved from the IPFS node. We then dispatch it back to the store. The reducer updates the state property `imageBlob` accordingly. The new value will be immediately picked up by the `getImageBlob` selector in the component and passed down through the pipes first using the `checkStore()` method and then the `tap()` operator. In the operator `tap()` we are using the template reference variable `#ipfsImage` to hookup the blob image into the template `<img>` native element.

Part 11: Creating a Purchase Contract

In the previous section, we discussed how to employ the IPFS to store the product image. In this part, we will focus on creating the Purchase Contract instances. We are planning to use the NgRx entity state adapter to manage a collection of the purchase contract entities:

```
npm install @ngrx/entity --save
```

We'll also bundle all related functionality in our lazy-loaded feature module `MarketPlaceModule`.

Building the Smart Contract Service

Let's create a brand new service that will be responsible for communication with the `FleaMarketFactory` smart contract. Start by defining the Injectable contract token:

```
import { Injectable } from '@angular/core';
import { Contract } from 'ethers';
import { EthersWeb3Token } from './ethers-web3-token';
import { MarketPlaceAnchorModule } from '../../market-place-anchor.module';
import { environment } from 'src/environments/environment';

const FLEA_MARKET_CONTRACT_ADDRESS = environment.fleaMarketContractAddress;
const abi = [
    'event LogCreatePurchaseContract(address sender, address contractAddress)',
    'event LogRemovePurchaseContract(address sender, bytes32 key)',
    'function createPurchaseContract(bytes32 key, string description,
       string ipfsImageHash, uint256 commissionRate)
       payable returns(bool createResult)',
    'function getContractCount() view returns(uint contractCount)',
    'function getContractKeyAtIndex(uint index) view returns(bytes32 key)',
    'function getContractByKey(bytes32 key) view returns(address contractAddress)',
    'function contractName() view returns(string contractName)',
```

```
    'function removeContractByKey(bytes32 key) returns(bool result)'
];

@Injectable({ providedIn: MarketPlaceAnchorModule })
export class FleaMarketContractToken extends Contract {
    constructor(provider: EthersWeb3Token) {
      super(FLEA_MARKET_CONTRACT_ADDRESS, abi, provider.getSigner());
    }
}
```

Here we assign the variable `FLEA_MARKET_CONTRACT_ADDRESS` the address of the deployed `FleaMarketFactory` contract, which we stored in the `src/environments` area. We are also taking advantage of the Human-Readable ABI of the ethers.js library since we only have to specify the functions and events signatures. By having the contract address and the ABI definition, we can now identify our smart contract on the Ethereum blockchain.

Now that we have defined the injectable contract token, we can pass it in the `FleaMarketContractService` constructor:

```
import { Injectable } from '@angular/core';
import { MarketPlaceAnchorModule } from '../market-place-anchor.module';
import { FleaMarketContractToken } from './tokens/flea-market-contract-token';
import { Observable, from, of, forkJoin } from 'rxjs';
import { map, tap, switchMap, mergeMap, exhaustMap } from 'rxjs/operators';
import { ethers, utils } from 'ethers';
import { PurchaseWidgetModel } from '../models';

@Injectable({ providedIn: MarketPlaceAnchorModule })
export class FleaMarketContractService {
    constructor(private contractToken: FleaMarketContractToken) {
    }

    public createPurchaseContract(product: any): Observable<string> {

      const commission = Math.floor(parseFloat(product.commission) * 100);
      const bytes32Key = utils.formatBytes32String(product.productKey);
      const wei = utils.parseEther(product.etherValue);

      // based on https://docs.ethers.io/ethers.js/html/cookbook-contracts.html
      // Call the contract method, getting back the transaction tx
      const token =
```

```
      this.contractToken.createPurchaseContract(bytes32Key, product.description,
        product.ipfsHash, commission, {
          value: wei
      });
    return from(token)
      .pipe(
        switchMap((tx: any) => {

          console.log('Transaction', tx);
          // Wait for transaction to be mined
          // Returned a Promise which would resolve to the TransactionReceipt
          // once it is mined.
          return from(tx.wait()).pipe(
            tap((txReceipt: any) => console.log('TransactionReceipt: ', txReceipt)),

            // The receipt will have an "events" Array, which will have
            // the emitted event LogCreatePurchaseContract
            map(txReceipt => txReceipt.events.pop()),
            map(txEvent => txEvent.args.contractAddress),
            tap(address => console.log('address: ', address)));
        }));
  }
...

}
```

To avoid a circular dependencies warning, for both contract token and contract service we specify the `providedIn` dependency injection to a lazy "dummy" module `MarketPlaceAnchorModule` instead of the feature module `MarketPlaceModule`.

The service method `createPurchaseContract()` is responsible for creating `SafeRemotePurchase` smart contract entities defined in the `SafeRemotePurchase.sol`. One of the features of the ethers.js API is that the transaction we send to a blockchain node becomes available to us even before it is mined. While waiting for a transaction to be mined, we may choose to implement some other side effects, such as to store some transaction details in an external database. Once a new contract has been created and mined, we should receive the transaction receipt `txReceipt`, which contains the last emitted event `LogCreatePurchaseContract`. We can use this event object to retrieve the contract address value passed to it as the second parameter: `txEvent.args.contractAddress`.

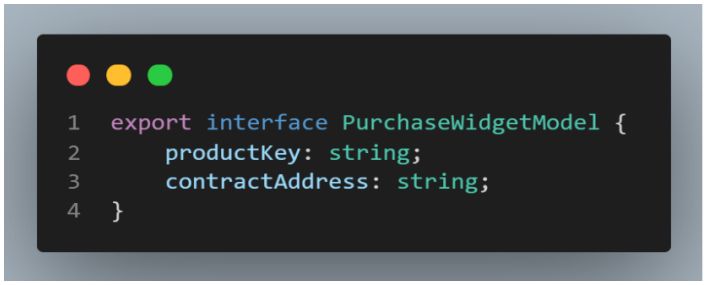
Define The Entity State

We manage the collection of the purchase contract entities with the following entity state adapter:

```
import { createEntityAdapter, EntityAdapter, EntityState } from '@ngrx/entity';
import { createReducer, on } from '@ngrx/store';
import { PurchaseWidgetModel, PurchaseContractModel } from '../../models';
import { PurchaseContractActions } from '../actions';

export interface State extends EntityState<PurchaseWidgetModel> {
    loaded: boolean;
    selectedPurchaseContract: PurchaseContractModel;
}

export function sortByKey(a: PurchaseWidgetModel, b: PurchaseWidgetModel)
    : number {
      return a.productKey.localeCompare(b.productKey);
}

export const adapter: EntityAdapter<PurchaseWidgetModel> =
    createEntityAdapter<PurchaseWidgetModel>({
      selectId: (product: PurchaseWidgetModel) => product.productKey,
      sortComparer: sortByKey,
});

export const initialState: State = adapter.getInitialState({
      loaded: false,
      selectedPurchaseContract: null
});

export const reducer = createReducer(
    initialState,
    on(
        PurchaseContractActions.loadProductsSuccess,
        (state, { products }) => adapter.addAll(products, {
          ...state,
          loaded: true,
          selectedPurchaseContract: null
        })
    ),
    on(
        PurchaseContractActions.removePurchaseContractSuccess,
```

```
      (state, { key }) => adapter.removeOne(key, {
        ...state,
        selectedPurchaseContract: null
      })
    ),
      /**
      * The addOne function provided by the created adapter
      * adds one record to the entity dictionary
      * and returns a new state including that records if it doesn't
      * exist already. If the collection is to be sorted, the adapter will
      * insert the new record into the sorted array.
      */
    on(
      PurchaseContractActions.createPurchaseContractSuccess,
      (state, { product }) => adapter.addOne(product, state)),
    on(
      PurchaseContractActions.loadPurchaseContractSuccess,
      (state, { contract }) => ({
        ...state,
        selectedPurchaseContract: contract,
    })),

);
```

We also specify the entity unique identifier and the default entity sort order. The entity model is defined as:



```
export interface PurchaseWidgetModel {
    productKey: string;
    contractAddress: string;
}
```

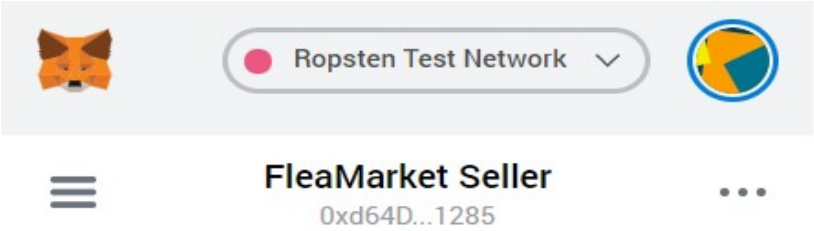
We then hook up the entity state into the feature state `PurchaseContractState`:

```
import {createSelector, createFeatureSelector, Action, combineReducers}
    from '@ngrx/store';
import * as fromRoot from '../../../core/store/reducers';
import * as fromIpfs from './ipfs-product-image.reducer';
import * as fromProducts from './purchase-contract.reducer';
```

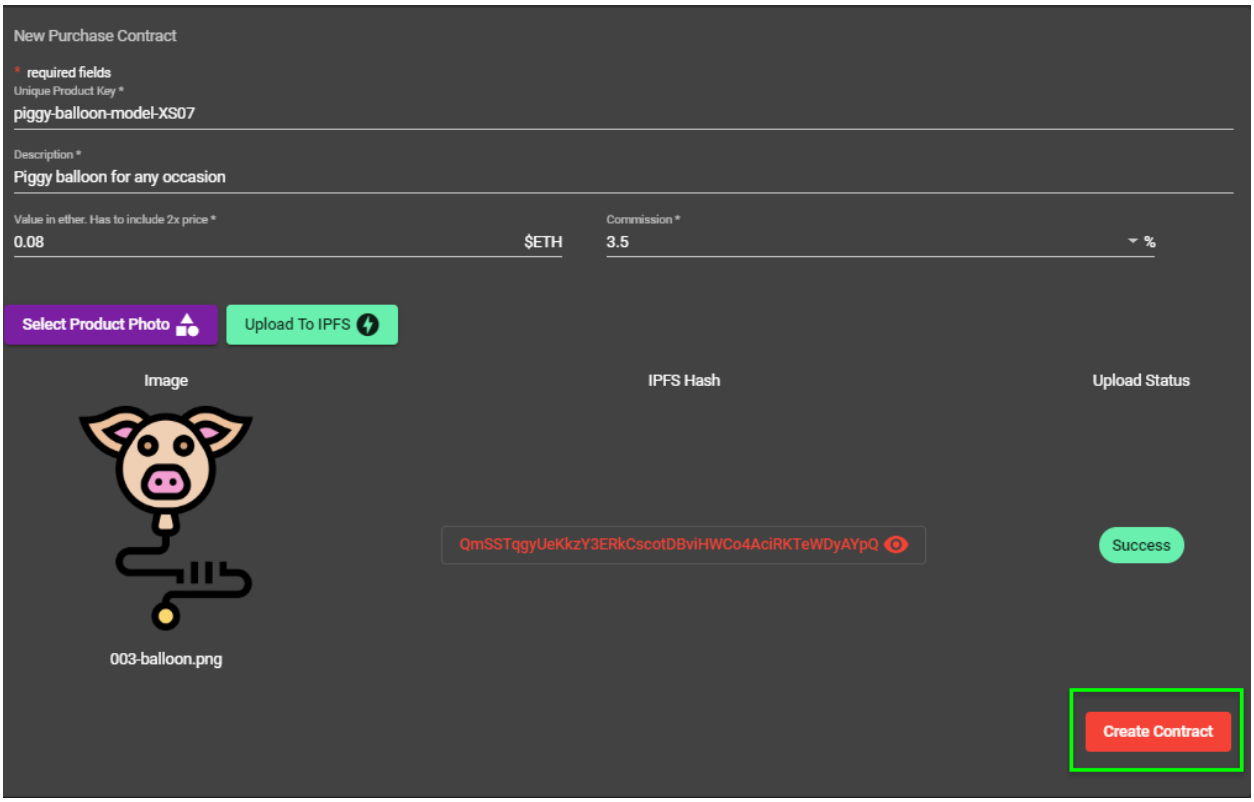
```
export interface PurchaseContractState {
    ipfs: fromIpfs.State;
    products: fromProducts.State;
}

export interface AppState extends fromRoot.AppState {
    purchaseContract: PurchaseContractState;
}

export function reducers(state: PurchaseContractState | undefined, action: Action) {
    return combineReducers({
      ipfs: fromIpfs.reducer,
      products: fromProducts.reducer
    })(state, action);
}
```

Creating a New Purchase Contract in Actions

Let's switch to the MetaMask account that uses the Seller's wallet.



To wire up the new purchase contract logic, we need to fill in the details of a new product into the Angular material reactive form:



Notice, that the amount of ETH entered by the seller has to be two times the purchase price. In the example above, the purchase price is 0.04 ETH (0.04 x 2 = 0.08). We also selected the product image and uploaded it to the IPFS file system and received the corresponding hash code.

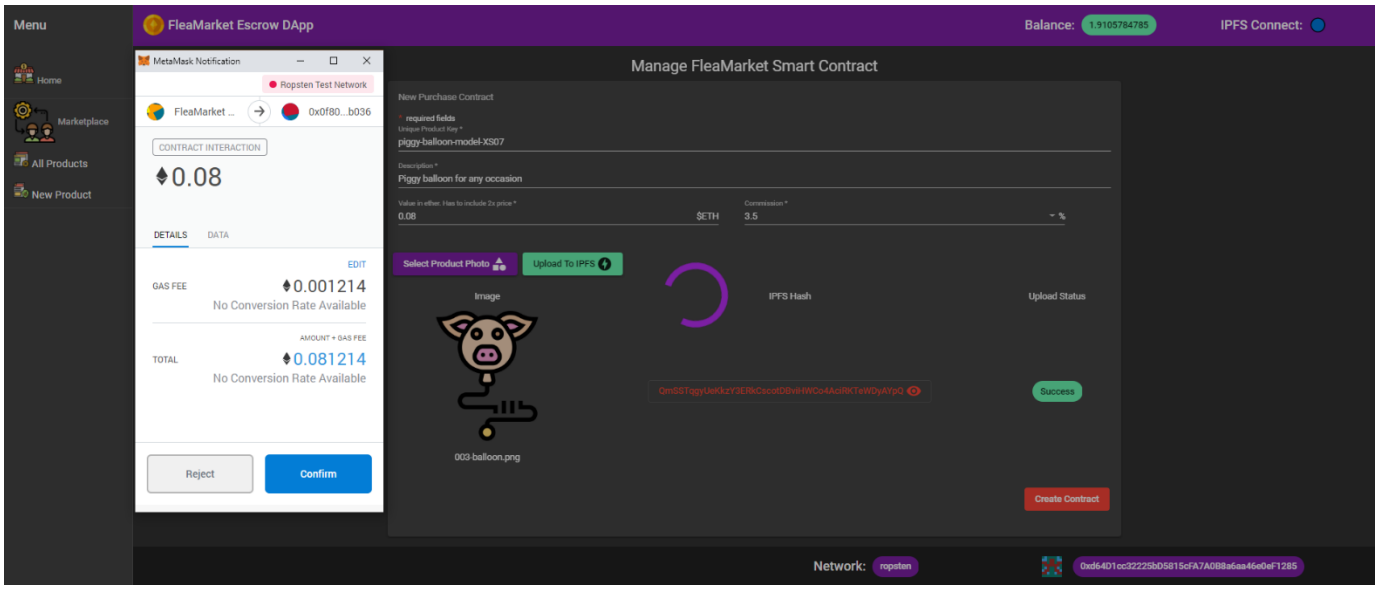
Once we satisfy the reactive form validation requirements, the button labeled 'Create Contract' becomes active, which allows us to trigger the store action that carries the reactive form fields values in its payload.

```
onCreate(): void {
    const { valid } = this.frmGroup;

    if (valid) {
      const { fileArg, ...model } = this.frmGroup.value;
      this.store$.dispatch(PurchaseContractActions.createPurchaseContract({ payload: model }));
    }

  }
```

The action is dispatched to the store effect `createProduct$`:

```
createProduct$ = createEffect(
    () => this.actions$.pipe(
      ofType(PurchaseContractActions.createPurchaseContract),
      map(action => action.payload),
      exhaustMap((payload) => {

        return this.fleaSrv.createPurchaseContract(payload).pipe(
          tap(address => console.log('Contract address: ', address)),
          switchMap((address: string) => {

            return [
              PurchaseContractActions.createPurchaseContractSuccess({
                product: {
                    productKey: payload.productKey,
                    contractAddress: address
                }
              }),
              // update ballance
              Web3GatewayActions.getBalance()];
          }),
          catchError((err: Error) =>
            of(this.handleError(err), SpinnerActions.hide(),
              Web3GatewayActions.getBalance())
          )
        );
      })

    ));
```

We use the `exhaustMap()` operator to retrieve the action's payload data object. The advantage of using this mapping operator is that it will ignore sequential requests while the current one is still ongoing. We then pass the payload as a parameter into the `createPurchaseContract()` service method. A call to this method will alter the state of the `FleaMarketFactory` smart contract. As a result, it will initiate a new transaction on the Ethereum blockchain and we have to pay the gas fee for it in ETH. At that moment, the MetaMask will pop up, asking to confirm the transaction:



Once we approve the transaction, we are waiting for successful contract execution. The method `createPurchaseContract()` gets resolved with a new `PurchaseWidgetModel` entity. We pipe this value to the `switchMap()` operator and dispatch it back to the store with the `createPurchaseContractSuccess()` action.

```
export const createPurchaseContractSuccess =
createAction('[PurchaseContract/Command] Create Purchase Contract Success', props<{
product: PurchaseWidgetModel}>());
```

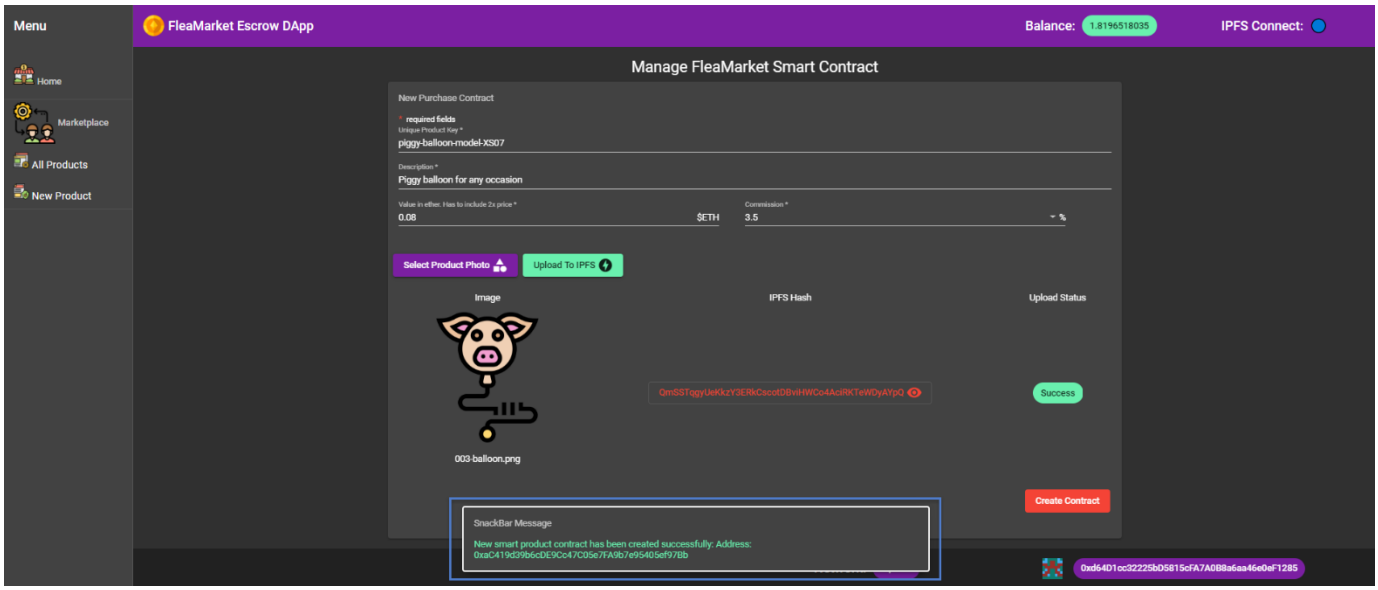
We also dispatch the `getBalance()` action to update the current account balance displayed on the toolbar to reflect the money spent on gas.

The action that carries the new purchase widget model continues the journey through the feature store and gets picked up by another store effects `showSnackbarOnCreateContract$`:

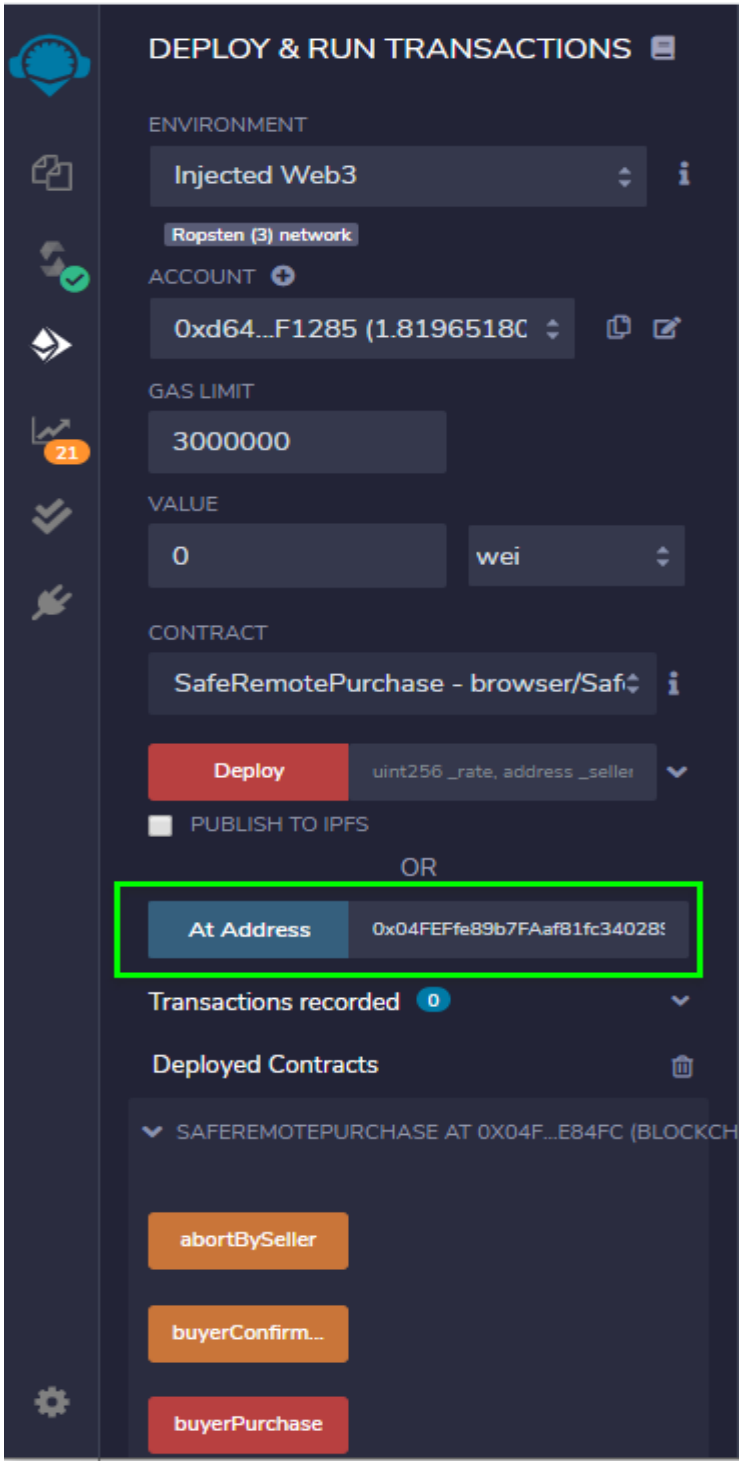
```
showSnackbarOnCreateContract$ = createEffect(() =>
    this.actions$.pipe(
      ofType(PurchaseContractActions.createPurchaseContractSuccess),
      map((payload) => {
        const msg: SnackBarInterface = {
```

```
            message: `New smart product contract has been created successfully:
                      Address: ${payload.product.contractAddress}`,
            color: AppearanceColor.Success
          };

          return SnackBarActions.open({ payload: msg });
        })
      )
);
```

It will pop up a success notification message that displays the address of a newly-created purchase contract:



The easy way to confirm that a new purchase contract has been deployed to the blockchain successfully is to open the Remix IDE, connect to the Robsten network, and enter the contract address in 'At Address' field:


DEPLOY & RUN TRANSACTIONS
ENVIRONMENT
Injected Web3
Ropsten (3) network
ACCOUNT
0xd64...F1285 (1.8196518C
GAS LIMIT
3000000
VALUE
0
wei
CONTRACT
SafeRemotePurchase - browser/Saf
Deploy
uint256 _rate, address _seller
PUBLISH TO IPFS
OR
At Address
0x04FEFfe89b7FAaf81fc34028
Transactions recorded
0
Deployed Contracts
SAFEREMOTEPURCHASE AT 0X04F...E84FC (BLOCKCH
abortBySeller
buyerConfirm...
buyerPurchase

Part 12: Purchase Contract Workflow

In the previous section, we discussed how to create a new instance of the purchase contract. In this part, we will walk through the code that lets us implement the following functionalities of our FleaMarket Escrow smart contract:

- Display the list of all purchase contract widgets
- Implement the search for contract
- Load the purchase contract
- Abort contract by the seller
- Remove contract by the seller
- Confirm purchase by the buyer
- Confirm delivery by the buyer
- Refunding the seller and the contract owner

Load Purchase Widget Collection

The feature-level `Routes` array for the lazy-loaded module `MarketPlaceModule` has the following context:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import * as fromContainers from './containers';
import * as fromComponents from './components';
import * as guards from './guards';

const routes: Routes = [
   {
     path: '',
     redirectTo: 'products',
```

```
      pathMatch: 'full',
    },
    {
      path: 'products',
      component: fromComponents.MarketPlaceHomeComponent,
      children: [
        {
          path: '',
          component: fromContainers.ViewProductCollectionComponent,
          canActivate: [guards.ProductsLoadedGuard],
          children: [
            {
              path: ':id',
              component: fromContainers.ViewPurchaseContractComponent,
            },
            {
              path: '',
              component: fromComponents.ProductDetailHomeComponent
            },
          ]
      },
      {
        path: 'make/new',
        component: fromContainers.NewPurchaseContractComponent,
        pathMatch: 'full',
      },

    ]
  }
];

@NgModule({
    imports: [
      RouterModule.forChild(routes),
    ],
    exports: [RouterModule]
})
export class MarketPlaceRoutingModule { }
```

It introduces the route-guard `ProductsLoadedGuard` that controls the navigation to the top-level route `/products`. The guard will wait for the collection of the purchase contract widgets to load from the blockchain by observing the `loaded` property of the entity state:

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { Store, select } from '@ngrx/store';
import { Observable, of } from 'rxjs';
import { take, tap, filter, switchMap, catchError } from 'rxjs/operators';
import { MarketPlaceAnchorModule } from '../market-place-anchor.module';
import * as fromStore from '../store/reducers';
import { PurchaseContractActions } from '../store/actions';

@Injectable({
    providedIn: MarketPlaceAnchorModule
})
export class ProductsLoadedGuard implements CanActivate {
    constructor(private store: Store<fromStore.AppState>) {}

    canActivate(): Observable<boolean> {
      return this.waitForProductsToLoad().pipe(
        switchMap(() => of(true)),
        catchError(() => of(false))
      );
    }

    waitForProductsToLoad(): Observable<boolean> {
      return this.store.pipe(
        select(fromStore.isProductsLoaded),
        tap(loaded => {
          if (!loaded) {
            this.store.dispatch(PurchaseContractActions.loadProducts());
          }
        }),
        filter(loaded => loaded),
        take(1)
      );
    }
}
```

If the value of `loaded` is false, the guard will dispatch the `loadProducts()` action to the store effect `loadProducts$`:

```
loadProducts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(PurchaseContractActions.loadProducts),
      switchMap(() =>
        this.fleaSrv.getPurchaseContractList().pipe(
```

```
        tap(products =>
          console.log('purchase contracts:', products)),
        map(products =>
          PurchaseContractActions.loadProductsSuccess({ products })),
        catchError((err: Error) =>
            of(this.handleError(err), SpinnerActions.hide()))
        )
      )
    ));
```

The effect then handles the call to the service method `getPurchaseContractList()` to fetch the collection data from the blockchain:

```
private widgetObservable = (id: number): Observable<PurchaseWidgetModel> =>
    from(this.contractToken.getContractKeyAtIndex(id)).pipe(
      switchMap(key => from(this.contractToken.getContractByKey(key)).pipe(
        map(address => {
          const widget: PurchaseWidgetModel = {
            productKey: utils.parseBytes32String(key as ethers.utils.Arrayish),
            contractAddress: address as string
          };
          return widget;
        })
      ))
    )

public getPurchaseContractList(): Observable<PurchaseWidgetModel[]> {
    return from(this.contractToken.getContractCount()).pipe(
      map((bigNumber: ethers.utils.BigNumber) => bigNumber.toNumber()),
      tap((contractCount: number) =>
        console.log('contractCount: ', contractCount)),
      switchMap((contractCount: number) => {
        if (contractCount === 0) {
          return of([]);
        } else {
          // we get array [0,1,....contractCount-1]
          const countArr: number[] = Array.from(Array(contractCount)).map((e, i) => i);
          const source = of(countArr);
          return source.pipe(
            mergeMap(ids => forkJoin(ids.map(this.widgetObservable)))
          );
        }
      })
```

```
    );
}
```

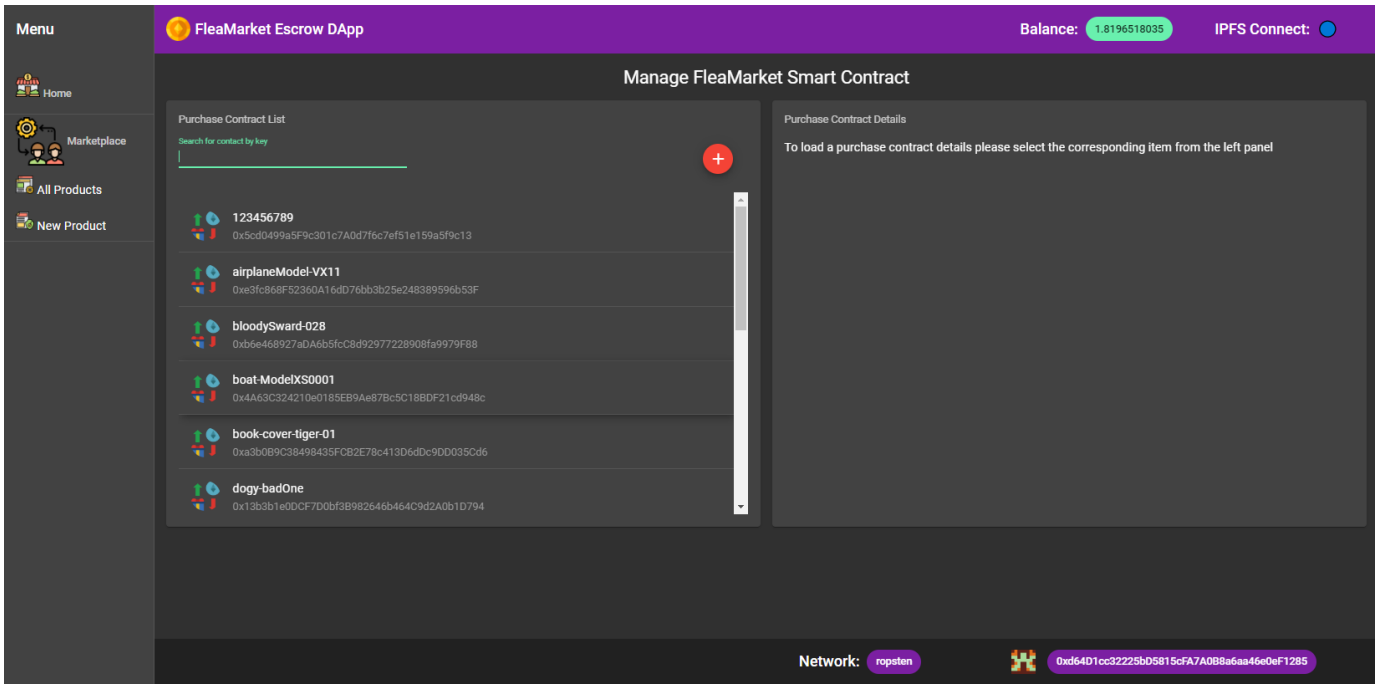
We call the smart contract method `getContractCount()` to retrieve the total amount of the purchase contract instances. Then, we use the combination of the operators `mergeMap()` and `forkJoin()` to execute multiple requests to the smart contract to retrieve the purchase widget collection.

Once the method `getPurchaseContractList()` gets resolved with the list of the `PurchaseWidgetModel` entities, we pass it into the payload of the `loadProductsSuccess()` action and dispatch it back to the entity store. The entity state reducer then calls the entity adapter method `addAll()` to replace the entity state with a new collection. The reducer will also change the state property `loaded` to `true` and permit access to the `/products` route.

Since the `/products` route becomes active, it will also activate the route component `ViewProductCollectionComponent`. In the component, we assign the `getAllProducts` store selector to the `products$` observable and implement the search by the contract key functionality:

```
ngAfterViewInit () {
    const products$ = this.store$.pipe(select(fromStore.getAllProducts));
    const filters$ = fromEvent(this.contractKey.nativeElement, 'keyup).pipe(
      map(event => this.contractKey.nativeElement.value),
      startWith(''),
      debounceTime(150),
      distinctUntilChanged());

    this.filteredProducts$ = combineLatest([products$, filter$]).pipe(
      map(([products, filterString]) =>
        products.filter(product =>
          product.productKey.indexOf(filterString) !== -1)
    );
    setTimeout(() => {
      this.contractKey.nativeElement.focus();
    }, 150);
}
```

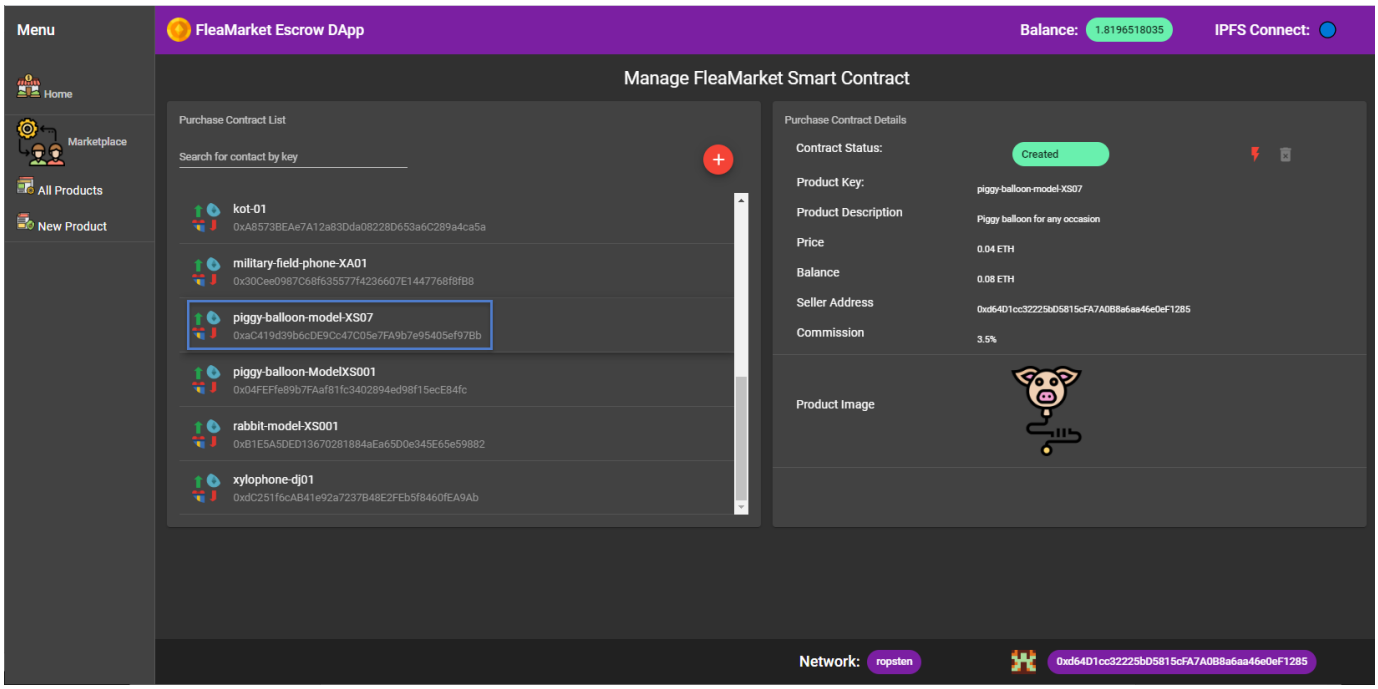
As a result, the component template should render the collection of our purchase widgets as follows:



Each item in the widget collection contains the key and the address of the purchase contract. When a new instance of the purchase contract is created, the new purchase widget model is added to the entity store. This will trigger the `getAllProducts` selector to emit a new value and update the content of the collection.

View Purchase Contract Details

When a user clicks an item in the widget list, it will activate the child route *`products/id`* where *`id`* is the unique product key we assigned to each widget. The child route will load the route component `ViewPurchaseContractComponent` with the corresponding purchase contract context:



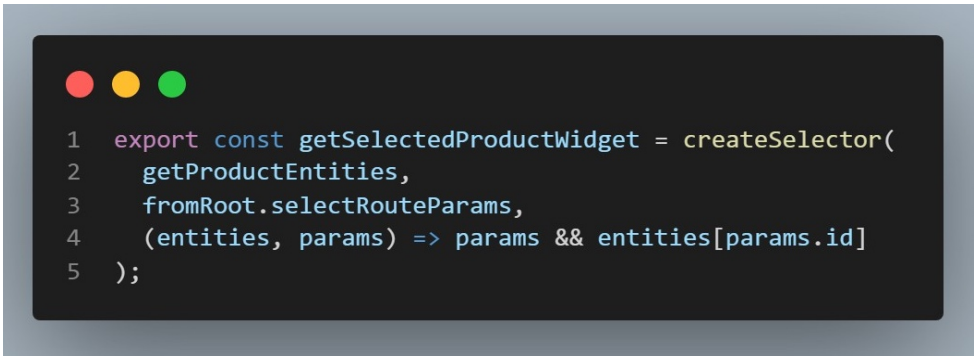
Let's overview the code in the `ViewPurchaseContractComponent` class that drives that piece of logic.

```
export class ViewPurchaseContractComponent implements OnInit, OnDestroy {
    selectedPurchaseContract$: Observable<PurchaseContractModel>;
    image$: Observable<Blob>;

    constructor(
      private store$: Store<fromStore.AppState>,
    ) { }

    ngOnInit() {
      this.selectedPurchaseContract$ = this.store$.pipe(
        select(fromStore.getSelectedProductWidget),
        filter(product => !!product),
        tap(product =>
          this.store$.dispatch(PurchaseContractActions.
            loadPurchaseContract({ address: product.contractAddress }))),
            // we switch from one observable to another
        switchMap(() =>
          this.store$.select(fromStore.getSelectedPurchaseContract)),
        filter(contract => !!contract),
    );

    this.image$ = this.store$.pipe(
      select(fromStore.getSelectedPurchaseContract),
```

```
      filter(contract => !!contract),
      tap(contract =>
       this.store$.dispatch(IpfsImageActions.
         downloadImage({ ipfsHash: contract.ipfsHash }))
       ),
      // we switch from one observable to another
      switchMap(() => this.store$.select(fromStore.getImageBlob)),
      filter(image => !!image)
    );
}
```

Here we declare two public observable fields: `selectedPurchaseContract$` and `image$`. We use the first observable to pull out the purchase contract properties from the Ethereum blockchain and the second observable to load the corresponding product image from the IPFS node.

In the `ngOnInit()` life-cycle hook we assign the `selectedPurchaseContract$` to the feature selector `getSelectedProductWidget`:



```
export const getSelectedProductWidget = createSelector(
  getProductEntities,
  fromRoot.selectRouteParams,
  (entities, params) => params && entities[params.id]
);
```

The `getSelectedProductWidget` is a combined selector that returns a specific member from the product widgets collection based on the product key which we specify in the route parameter. We then dispatch the widget contract address with the `loadPurchaseContract()` action to the store effect:

```
loadPurchaseContract$ = createEffect(
    () => this.actions$.pipe(
      ofType(PurchaseContractActions.loadPurchaseContract),
      map(action => action.address),
      switchMap(address => {

        return this.purchaseSrv.loadPurchaseContract(address).pipe(
          map(contract =>
            PurchaseContractActions.loadPurchaseContractSuccess({ contract })),
```

```
        catchError((err: Error) =>
          of(this.handleError(err), SpinnerActions.hide()))
      );
    })

));
```

Within the effect method, we use the `switchMap()` operator to receive a new inner observable of type `PurchaseContractModel`, that is returned from the `loadPurchaseContract()` method in the `PurchaseContractService`.

```
public loadPurchaseContract(contractAddress: string):
    Observable<PurchaseContractModel> {

    const contract: Contract =
      new ethers.Contract(contractAddress, this.abi, this.provider.getSigner());

    const crObservable: Observable<number | null> =
      from(contract.commissionRate()).pipe(
        map((commission: ethers.utils.BigNumber) => commission.toNumber()),
        // only account with deployer or seller can retrieve this value,
        // otherwise the contract will throw error.
        catchError((err: Error) => of(null))
    );

    return zip(
      from(contract.key()),
      from(contract.seller()),
      from(contract.buyer()),
      from(contract.owner()),
      from(contract.price()),
      from(contract.balanceOf()),
      from(contract.description()),
      from(contract.ipfsImageHash()),
      from(contract.state()),
      from(crObservable),
    )
      .pipe(
        map(([key, sellerAddress, buyerAddress, ownerAddress,
          weiPrice, weiBalance, description, ipfsHash, state, commission]) => {

          const product: PurchaseContractModel = {
            productKey: utils.parseBytes32String(key as ethers.utils.Arrayish),
```

```
            contractAddress,
            sellerAddress: sellerAddress as string,
            buyerAddress: (buyerAddress === ethers.constants.AddressZero) ?
              null : buyerAddress as string,
            ownerAddress: ownerAddress as string,
            price: utils.formatEther(weiPrice as ethers.utils.BigNumberish),
            balance: utils.formatEther(weiBalance as ethers.utils.BigNumberish),
            description: description as string,
            ipfsHash: ipfsHash as string,
            state: state as ContractState,
            commission: commission ? commission as number : null
          };

          return product;
        }),
      );
}
```

We are using the ethers.js library to retrieve the smart contract observable properties and apply the `zip()` operator to combine them in the observable object of type `PurchaseContractModel`. We then dispatch this object with the `loadPurchaseContractSuccess()` action to let the store reducer update the feature state property `selectedPurchaseContract`.

Back to the `ViewPurchaseContractComponent` component, we use the `switchMatch()` operator to observe the values returned from the feature selector `getSelectedPurchaseContract`, which emits a new value each time when the state property `selectedPurchaseContract` is changed.

Let's quickly review the code above for another component's observable `image$` that binds the purchase contract image to the template. Notice that the image isn't stored in the smart contract. We only store the corresponding IPFS hash value. The hash is used to retrieve the image context from a decentralized file system IPFS. We set it by observing the selected `PurchaseContractModel` object using the `select()` method and providing the `getSelectedPurchaseContract` selector function. Then, the IPFS hash gets extracted and dispatched with `download_image()` action to the feature effect `downloadImage$` like this:

```
downloadImage$ = createEffect(
    () =>
      this.actions$.pipe(
```

```
      ofType(IpfsImageActions.downloadImage),
      map((action) => action.ipfsHash),
      switchMap((ipfsHash: string) =>
        this.ipfsSrv.getFile(ipfsHash).pipe(
          map((image: Blob) => IpfsImageActions.downloadImageSuccess({ image })),
          catchError((err: Error) =>
            of(this.handleError(err), IpfsImageActions.downloadImageError())
          )
        )
      )
    )
  );
```

We are using the `switchMap()` operator to extract the IPFS hash value and call the `getFile()` method in `IpfsDaemonService` to retrieve the image Blob object from the IPFS node. We then dispatch the `downloadImageSuccess()` action to the store and provide the image Blob object as the payload. It will cause the store reducer to update the feature state and trigger the `selector()` function to emit a new value. Finally, we use the template reference variable `#ipfsImage` defined in the `PurchaseContractDetailComponent` to anchor the Blob image to its `src` property:

```
1  constructor(
2    @Inject(windowRefToken) private windowRef: Window
3  ) { }
4
5  ngOnChanges(changes: SimpleChanges) {
6
7    if (this.imageRef) {
8      this.imageRef.nativeElement.src = this.windowRef.window.URL.createObjectURL(this.image);
9    }
```

Aborting Contract

The seller may change his mind and decide to cancel a purchase contract. This is only allowed if the contract is in the state ***Created***. In this case, the amount of ETH staked by the seller will be refunded back to him. Let's take a look at what happened when the seller clicked on the power button.

The click event will dispatch the `abortSelectedPurchaseContract()` action to the store effect:

```
abortContract$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(PurchaseContractActions.abortSelectedPurchaseContract),
        withLatestFrom(
          this.store$.pipe(select(fromStore.getSelectedPurchaseContract))),
        switchMap(([action, contract]) => {

          const dialogConfig = new MatDialogConfig();
          dialogConfig.width = '420px';
          dialogConfig.disableClose = true;
          dialogConfig.autoFocus = true;
          dialogConfig.data = {
            title: 'Confirm Abort',
            content: `Are you sure you want to deactivate contract:
                                        ${contract.productKey}?`,
            output: contract.contractAddress
          };

          const dialogRef =
               this.dialog.open(ConfirmDialogComponent, dialogConfig);
          return dialogRef.afterClosed();
        }),
        filter(result => !!result),
        exhaustMap(result => concat(
          of(SpinnerActions.show()),
          this.purchaseSrv.abortPurchaseContract(result).pipe(
            tap(address =>
              console.log(`Successfully canceled contract: ${address}`)),
            concatMapTo(
              [PurchaseContractActions.abortSelectedPurchaseContractSuccess(),
              Web3GatewayActions.getBalance()]
            ),
            catchError((err: Error) =>
              of(this.handleError(err), SpinnerActions.hide(),
                Web3GatewayActions.getBalance())
              )
          ),
```

```
        of(SpinnerActions.hide()),
      ))
  ));
```

The effect then makes the call to the `abortPurchaseContract()` service method to initiate the smart contract transaction using ethers.js:

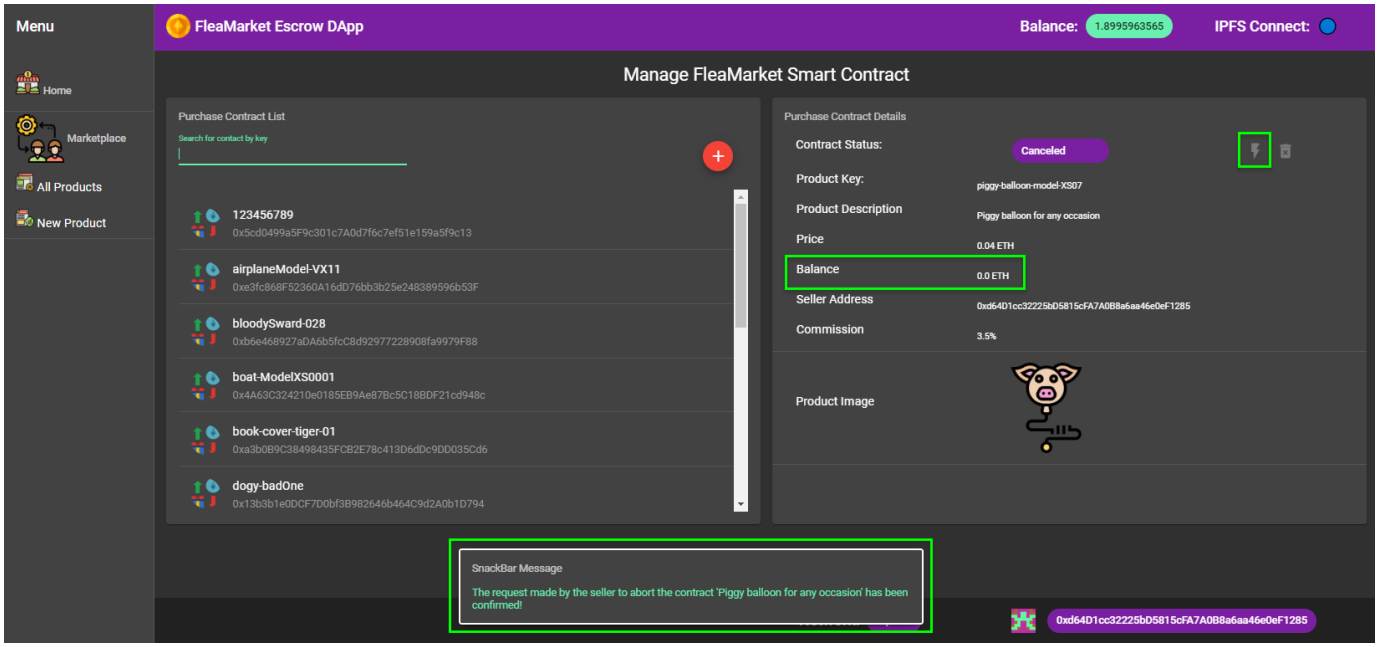
```
public abortPurchaseContract(contractAddress: string): Observable<string> {

    const contract: Contract = new ethers.Contract(contractAddress,
      this.abi, this.provider.getSigner());

    // Call the contract method, getting back the transaction tx
    const token = contract.abortBySeller();
    // 'from' operator can be used to conver a promise to an observable
    return from(token)
      .pipe(
        switchMap((tx: any) => {
          console.log('abortBySeller Tx:', tx);
          // Wait for transaction to be mined
          // Returned a Promise which would resolve to
          // the TransactionReceipt once it is mined.
          return from(tx.wait()).pipe(
            tap((txReceipt: any) => console.log('txReceipt: ', txReceipt)),
            // The receipt will have an "events" Array, which will have
            // the emitted event from the Contract.
            // The "LogCanceledBySeller" is the only event.
            map(txReceipt => txReceipt.events.pop()),
            tap(txEvent => console.log('event: ', txEvent.event)),
            mapTo(contractAddress),
          );
        }));
}
```

Upon successful execution of the transaction on the blockchain, the effect then broadcasts the `abortSelectedPurchaseContractSuccess()` action. This action is picked up by another non-dispatching effect `reload$`:

```
reload$ = createEffect(
    () =>
      this.actions$.pipe(
```

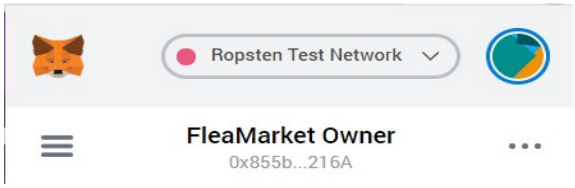
```
      ofType(
        PurchaseContractActions.abortSelectedPurchaseContractSuccess,
        PurchaseContractActions.confirmBuySuccess,
        PurchaseContractActions.confirmDeliverySuccess,
        PurchaseContractActions.releaseEscrowSuccess,
        PurchaseContractActions.withdrawByOwnerSuccess),
      withLatestFrom(
        this.store$.pipe(select(fromStore.getSelectedPurchaseContract))),
      tap(async ([action, contract]) => {

        // we need to reload on the same route
        // based on https://github.com/angular/angular/issues/13831
        this.router.routeReuseStrategy.shouldReuseRoute = ( ) => false;
        this.router.navigate(['/market-place/products', contract.productKey]);
      })
    ),
  { dispatch: false }
);
```

The sole purpose of this side effect is to force our application to reload the selected purchase contract from the blockchain and update the template. Finally, we display the SnackBar notification to the user. The item now shows the balance of 0.0 ETH and the status changed to ***Canceled***.

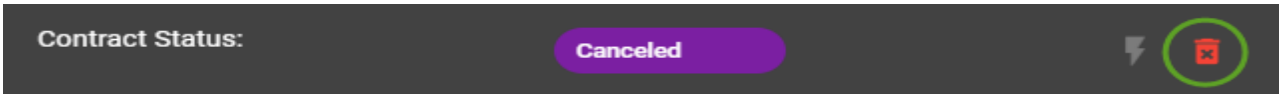


Removing Contract from Collection

After the contract has been canceled, the owner (the contract deployer) has an option to completely remove it from the product collection. Let's change our account on MetaMask to point to the owner's account:



When the owner clicks on the trash icon,



it'll dispatch the `removePurchaseContract()` action carrying the product key in its payload. Then, we follow the standard NgRx Redux pattern. We have a dedicated effect that is listening for this action:

```
removeProduct$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(PurchaseContractActions.removePurchaseContract),
        map(payload => payload.key),
        switchMap(key => {

          const dialogConfig = new MatDialogConfig();
          dialogConfig.width = '420px';
          dialogConfig.disableClose = true;
          dialogConfig.autoFocus = true;
          dialogConfig.data = {
            title: 'Confirm Remove',
            content: `Are you sure to remove contract ${key} from market?`,
            output: key
          };

          const dialogRef =
            this.dialog.open(ConfirmDialogComponent, dialogConfig);
```

```
      return dialogRef.afterClosed();
    }),
    filter(result => !!result),
    exhaustMap(result => concat(
      of(SpinnerActions.show()),
      this.fleaSrv.removePurchaseContract(result).pipe(
        tap(productKey =>
          console.log(`Contract has been removed: ${productKey}`)),
        concatMap(productKey =>
          [PurchaseContractActions.removePurchaseContractSuccess(
            { key: productKey }),
          Web3GatewayActions.getBalance()]
        ),
        catchError((err: Error) =>
          of(this.handleError(err), SpinnerActions.hide(),
            Web3GatewayActions.getBalance())
        ),
        of(SpinnerActions.hide()),
      ))
  ));
```

It will invoke the `removePurchaseContract()` method on the `FleaMarketContractService` and call the smart contract method `removeContractByKey()` using ethers.js to initiate the transaction on the blockchain:

```
public removePurchaseContract(productKey: string): Observable<string> {

    const bytes32Key = utils.formatBytes32String(productKey);
    const token = this.contractToken.removeContractByKey(bytes32Key);

    return from(token)
      .pipe(
        switchMap((tx: any) => {

          console.log('removeContractByKey Transaction', tx);
          // Wait for transaction to be mined
          // Returned a Promise which would resolve to
          // the TransactionReceipt once it is mined.
          return from(tx.wait()).pipe(
            tap((txReceipt: any) => console.log('TransactionReceipt: ', txReceipt)),
            // The receipt will have an "events" Array, which will have
            // the emitted event from the Contract.
            // The "LogRemovePurchaseContract(address sender, bytes32 key))
```

```
        // is the last event.
        map(txReceipt => txReceipt.events.pop()),
        tap(txEvent => console.log('txEvent: ', txEvent)),
        map(txEvent => {
          // retrieve the key parameter value from the event
          const key = txEvent.args.key;
          return utils.parseBytes32String(key as ethers.utils.Arrayish);
        }),
      );
    }));
}
```

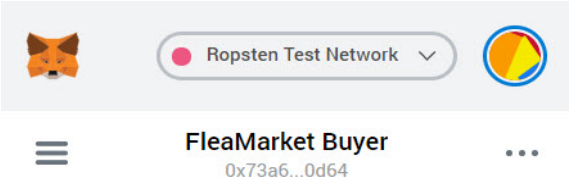
It then waits for the successful execution of the transaction and dispatches the `removePurchaseContractSuccess()` action to the store. As a result, the NgRx entity adapter will remove the corresponding `PurchaseWidgetMode` item from the collection in the entity state.

We also define another effect that is listening for the same action `removePurchaseContractSuccess()` to be dispatched and activate the navigate to the top-level feature route */market-place*:

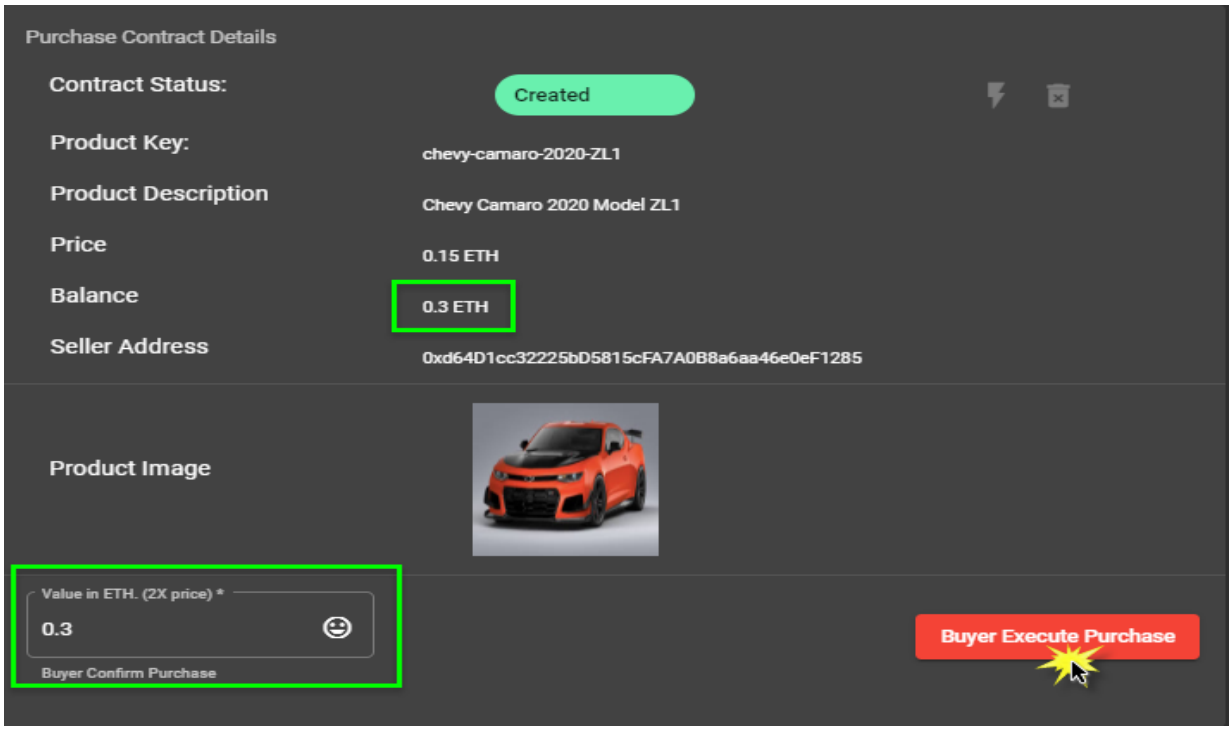
```
removeProductRedirect$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(PurchaseContractActions.removePurchaseContractSuccess),
        tap(_ => {
          this.router.navigate(['/market-place']);
        })
      ),
    { dispatch: false }
  );
```

Buyer Makes Purchase

Let's play as a buyer and switch to a MetaMask account that uses the buyer's wallet.



According to the logic of the smart contract, to purchase an item the buyer needs to deposit a double of the amount of ETH this item costs. The buyer then hits the Purchase button.



Here again, we leverage the powers of the NgRx Store. For that, we have a dedicated side effect that manages this part of purchase logic.

```
confirmBuy$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(PurchaseContractActions.confirmBuy),
        withLatestFrom(
          this.store$.pipe(select(fromStore.getSelectedPurchaseContract))),
        switchMap(([payload, contract]) => {

          const dialogConfig = new MatDialogConfig();
          dialogConfig.width = '420px';
          dialogConfig.disableClose = true;
          dialogConfig.autoFocus = true;
```

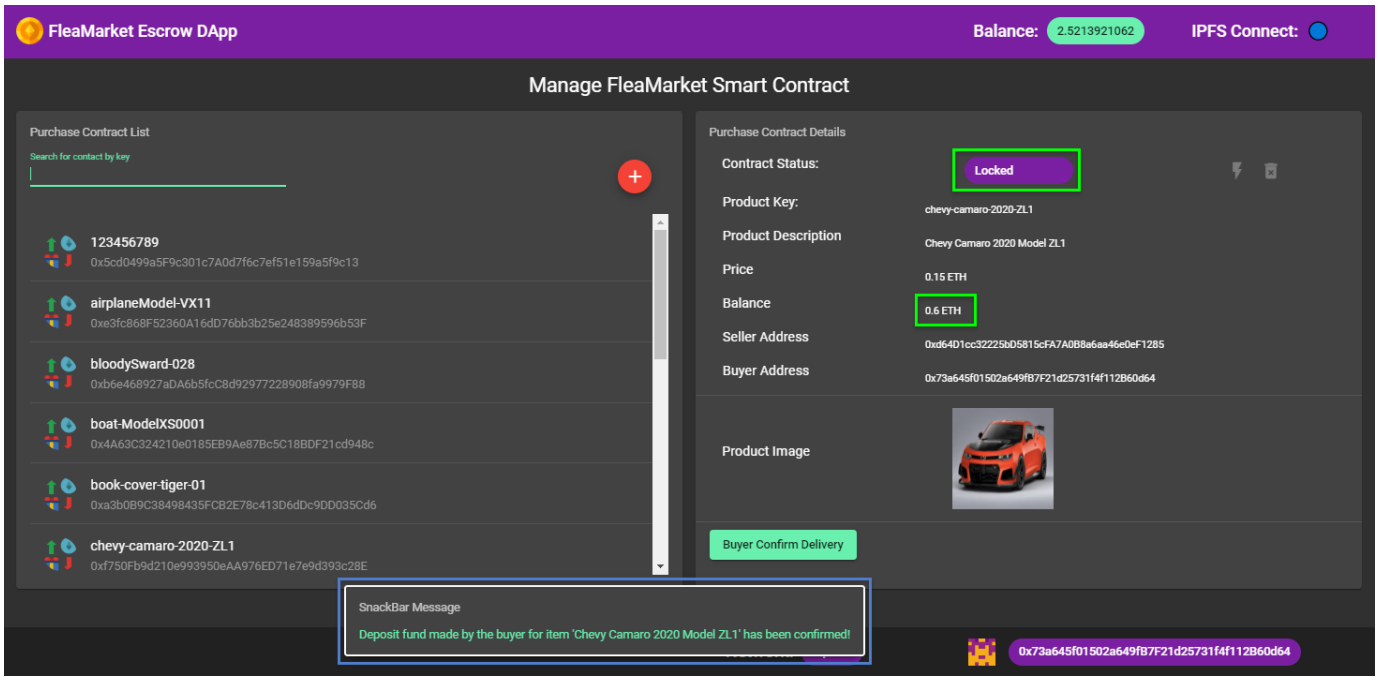
```
      dialogConfig.data = {
        title: 'Confirm Purchase',
        content: `Please confirm to deposit ${payload.eth} ETH into
                  the contract: ${contract.productKey}`,
        output: {
          address: contract.contractAddress,
          eth: payload.eth
        }
      };
      const dialogRef =
        this.dialog.open(ConfirmDialogComponent, dialogConfig);
      // Gets an observable that is notified
      //when the dialog is finished closing.
      return dialogRef.afterClosed();
    }),
    filter(result => !!result),
    exhaustMap(result => concat(

      of(SpinnerActions.show()),
      this.purchaseSrv.confirmPurchase(result.address, result.eth).pipe(
        tap(address =>
          console.log(`Purchase confirmed successfully for
                       the contract: ${address} `)),
        concatMapTo(
          [PurchaseContractActions.confirmBuySuccess(),
          Web3GatewayActions.getBalance()]
        ),
        catchError((err: Error) =>
          of(this.handleError(err), Web3GatewayActions.getBalance()))
      ),
      of(SpinnerActions.hide())
    ))
  ));
```

Upon closing the dialog that confirms our intention to purchase the item, we invoke the service method `confirmPurchase()`.

```
public confirmPurchase(contractAddress: string,
   etherValue: string): Observable<string> {

   const contract: Contract =
     new ethers.Contract(contractAddress, this.abi,
       this.provider.getSigner());
```

```
    const wei = utils.parseEther(etherValue);
    const token = contract.buyerPurchase({
      value: wei
    });

    return from(token).pipe(
      switchMap((tx: any) => {
        console.log('buyerConfirmPurchase Tx:', tx);
        // Wait for transaction to be mined
        // Returned a Promise which would resolve to
        // the TransactionReceipt once it is mined.
        return from(tx.wait()).pipe(
          tap((txReceipt: any) => console.log('txReceipt: ', txReceipt)),
          // The receipt will have an "events" Array, which will have
          // the emitted event from the Contract. The "LogPurchaseConfirmed"
          // is the last event.
          map(txReceipt => txReceipt.events.pop()),
          tap(txEvent => console.log('event: ', txEvent.event)),
          mapTo(contractAddress),

        );
      }));
}
```

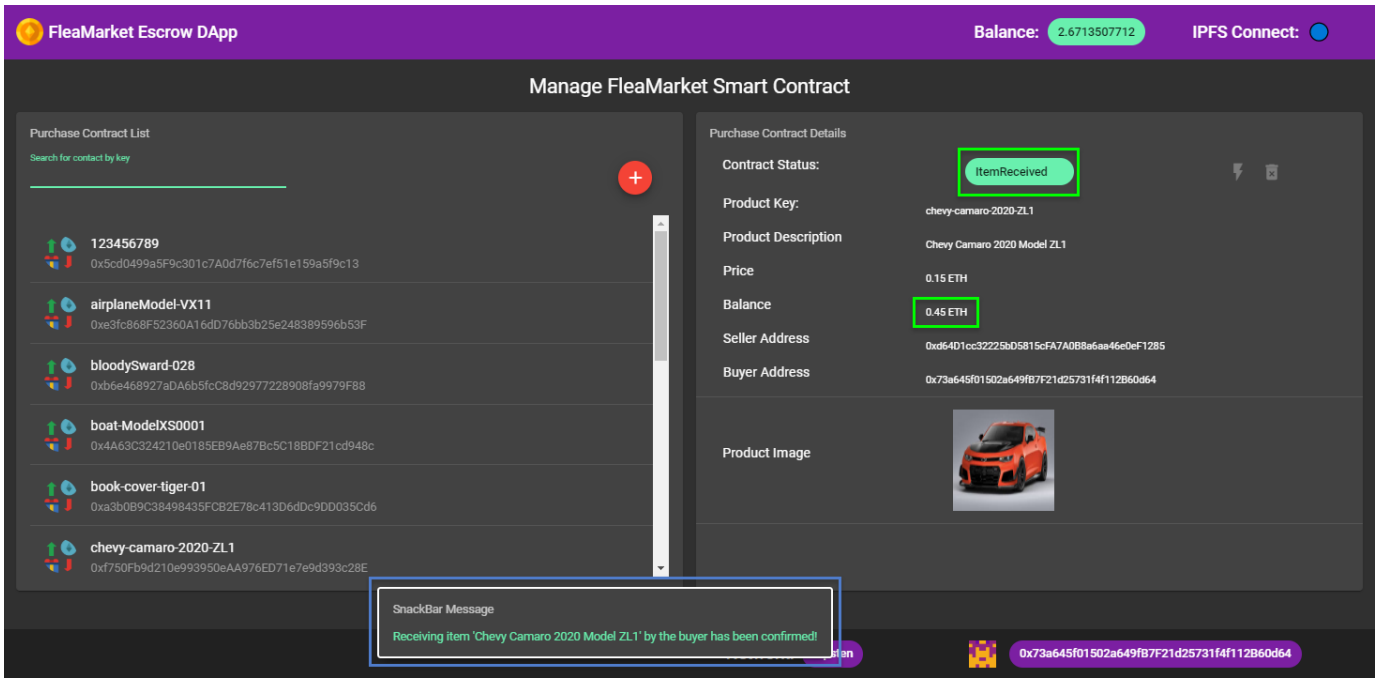
We send the transaction to deposit ethers by calling the `buyerPurchase()` method on the contract using the ethers.js library. Once we receive the transaction receipt from the blockchain, we return the control-flow logic to the effect and dispatch the `confirmBuySuccess()` action to refresh the contract's information on the page.



If everything goes as planned, we should see two important changes. The first change is the contract now shows a balance of ETH equals 4x the value of the product price. It is the 2x from the seller and 2x from the buyer. The second one is the status has been changed to ***Locked***. At this point, the smart contract no longer allows the buyer nor the seller to pull back from the deal.

Confirming Delivery

After a buyer has staked the right amount of ETH into the contract, the seller is obligated to deliver the purchased item. Upon receiving the item, the buyer confirms the delivery by clicking on the 'Confirm Delivery' button.



This will trigger the following chain of events. First, we dispatch the `confirmDelivery()` action to the store. Then, we use the `confirmDelivery$` effect.

```
confirmDelivery$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(PurchaseContractActions.confirmDelivery),
        withLatestFrom(this.store$.pipe(select
          (fromStore.getSelectedPurchaseContract))),
        switchMap(([payload, contract]) => {

          const dialogConfig = new MatDialogConfig();
          dialogConfig.width = '420px';
          dialogConfig.disableClose = true;
          dialogConfig.autoFocus = true;
          dialogConfig.data = {
            title: 'Confirm Delivery',
            content: `Are you sure you want to confirm that you
                     received the purchase item ${contract.description}`,
            output: contract.contractAddress
          };

          const dialogRef = this.dialog.open(ConfirmDialogComponent, dialogConfig);
          // Gets an observable that is notified when the dialog is finished closing.
          return dialogRef.afterClosed();
```

```
      }),
      filter(result => !!result),
      exhaustMap(result => concat(

        of(SpinnerActions.show()),
        this.purchaseSrv.confirmDelivery(result).pipe(
          tap(address => console.log
            (`Delivery confirmed successfully for the contract: ${address} `)),
          concatMapTo(
            [PurchaseContractActions.confirmDeliverySuccess(),
            Web3GatewayActions.getBalance()]
          ),
          catchError((err: Error) =>
            of(this.handleError(err), Web3GatewayActions.getBalance())
          )
        ),
        of(SpinnerActions.hide()),
      ))

    ));
```

From the side effect, we handle the call to the service method:

```
public confirmDelivery(contractAddress: string): Observable<string> {

    const contract: Contract = new ethers.Contract(contractAddress,
      this.abi, this.provider.getSigner());
    const token = contract.buyerConfirmReceived();

    return from(token).pipe(
      switchMap((tx: any) => {
        console.log('buyerConfirmReceived Tx:', tx);
        // Wait for transaction to be mined
        // Returned a Promise which would resolve to
        // the TransactionReceipt once it is mined.
        return from(tx.wait()).pipe(
          tap((txReceipt: any) => console.log('txReceipt: ', txReceipt)),
          // The receipt will have an "events" Array, which will have
          // the emitted event "LogReceivedByBuyer" from the contract
          map(txReceipt => txReceipt.events.pop()),
          tap(txEvent => console.log('event: ', txEvent.event)),
          mapTo(contractAddress),
        );
      }));
```

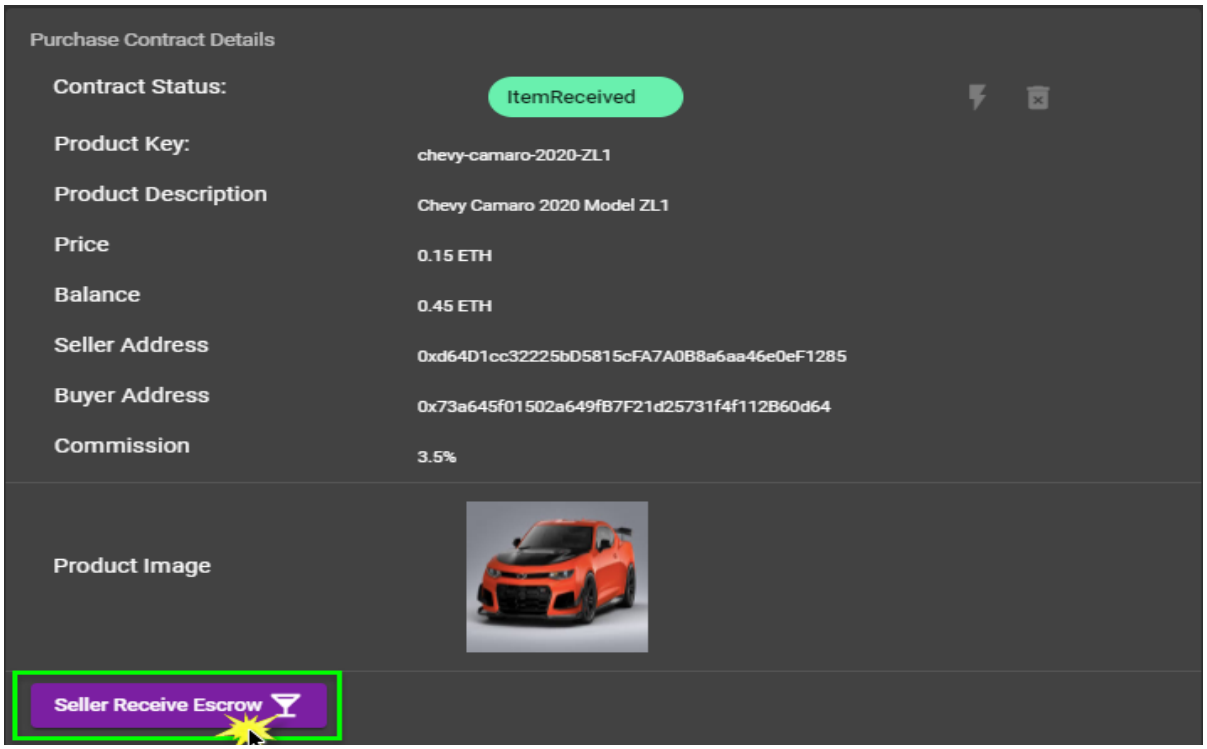
```
}
```

We execute the `buyerConfirmReceived()` method on the smart contract and wait for the contract to emit the transaction receipt. Finally, we dispatch the `confirmDeliverySuccess()` action to refresh the contract's properties on the component template.

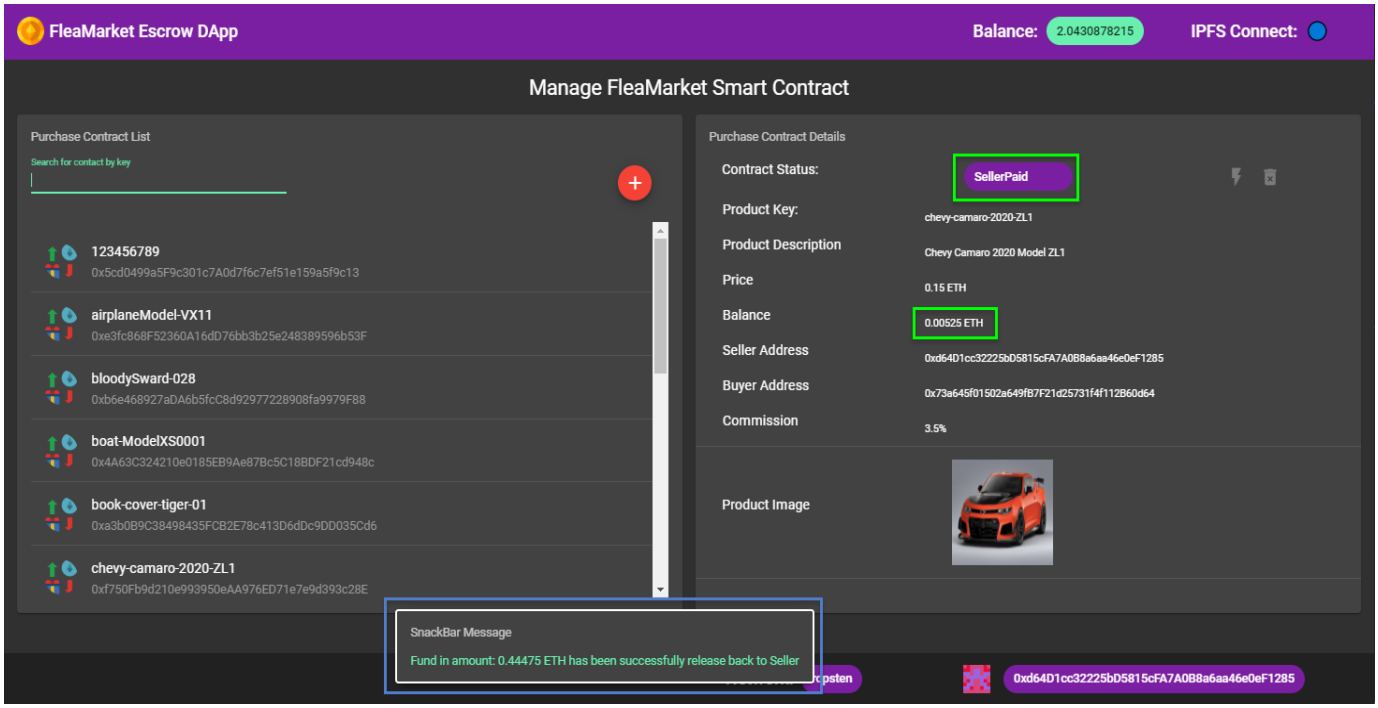
As we would expect, the purchase status has switched to the ***ItemReceived***. It also shows that the contract balance has been reduced by the amount of ETH of the item coast. This amount was served as collateral and held in the escrow by the smart contract. It has been now refunded back to the buyer by the built-in logic of the smart contract.

Refunding Escrow back to Seller and Owner

According to the business logic built into the smart contract, all collateral funds staked in the contract now could be refunded back to the seller and the contract owner.
The item in the ***ItemReceived*** status will have the button 'Seller Receive Escrow' or 'Owner Receive Commission' turned on depending on whether the seller or the contract owner load the item. For example, when we switch MetaMask account to the seller's account, it should look something like this:

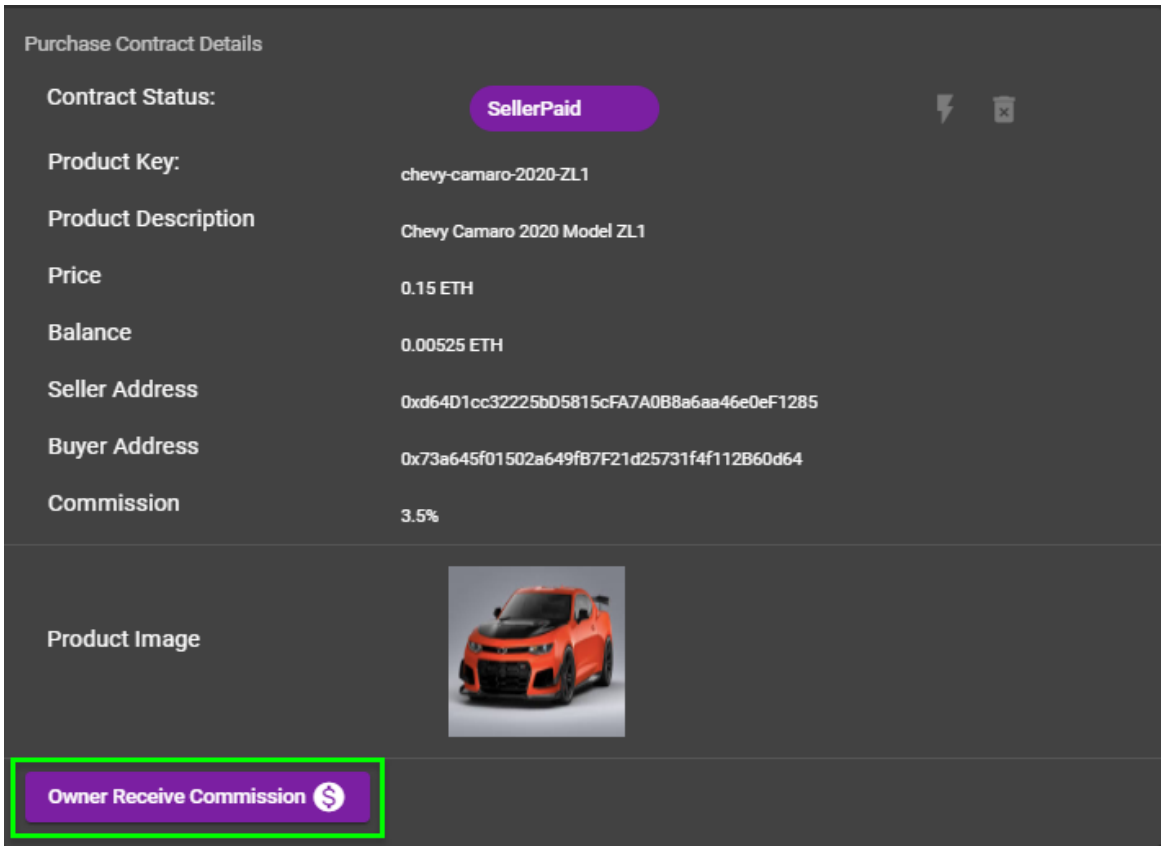


Now, the seller is ready to click the button. The code that runs this piece of logic is pretty much the same as it was discussed previously. We wait for the transaction to execute on the Blockchain. When it is done, we should see the snackbar confirmation pop-up and the balance changed:

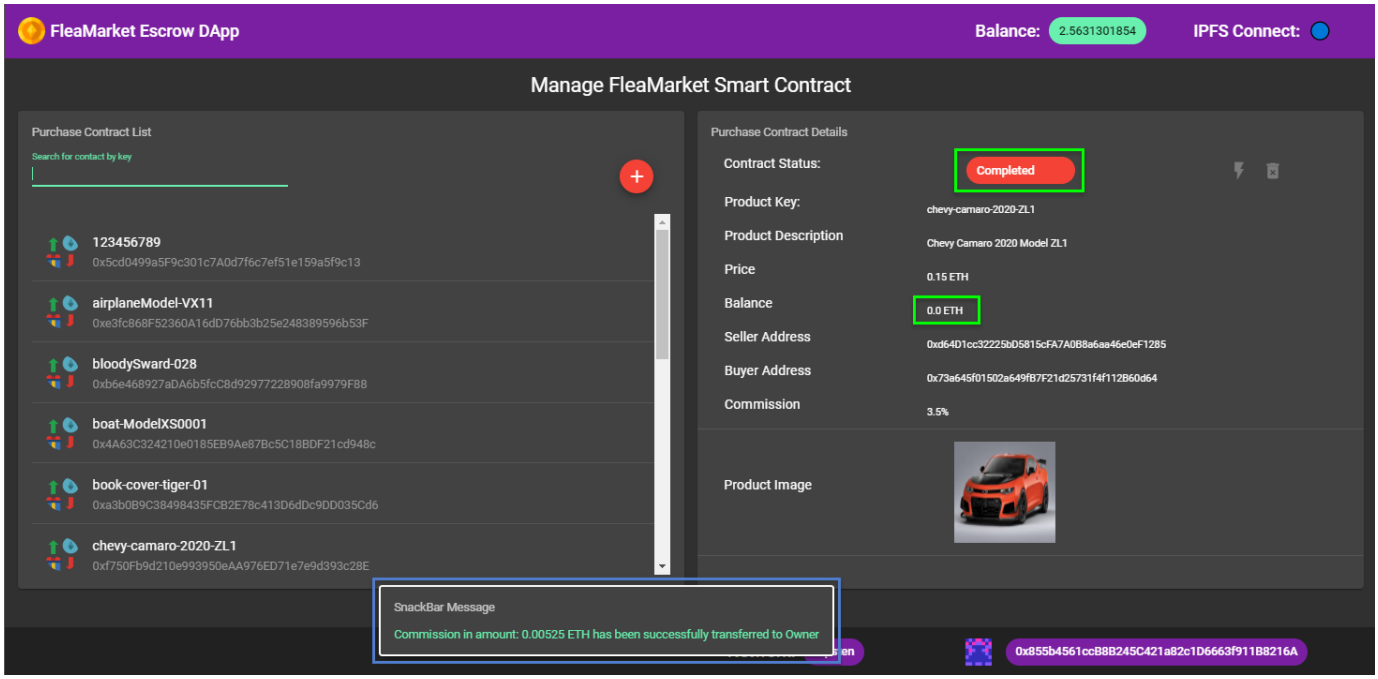


In the example above, the purchase screen now says ***SellerPaid*** and the smart contract has 0.00525 ETH balance left. Since the item cost 0.15 ETH and the seller has to pay commission in the amount of 3.5% of the selling price, the contract balance indeed should be equal to (0.15 x 3.5)/100 = 0.00525. The seller has been refunded back in the amount of 0.44475 ETH. Which is also true, because this value should be equal 3 times the item price minus the commission: (0.15 x 3) - 0.00525 = 0.44475.

Now, it is the owner's turn to receive the sales commission. Let's go back to MetaMask and switch the account to the owner's wallet. The button 'Owner Receive Commission' should become active.



Once the transaction to receive commission has been confirmed, we should see the updated status of the contract:



The item now says ***Completed***, balance is zero and The Smart Contract has **completed** its job.

Conclusion

This also completes our FleaMarket DApp running on Ethereum blockchain. It was a really enjoyable experience to build this application. It was fun to watch how different technologies come together to assemble a fully functional application running on Blockchain. Many different libraries and tools were involved like TypeScript, Angular, NgRx, ethers.js, IPFS, Solidity. It is worth noting that the Ethereum developer ecosystem grows and changes rapidly, with new tools emerging and old techniques becoming obsolete.

In summary, I would highlight the following two important ideas:

- In the core, the smart contract runs all the business logic to ensure that all parties (seller, buyer, and the owner), played their fair roles in the deal.

- The frontend application we built with Angular and it retrieves smart contract data from the Ethereum node with Ethereum client such ethers.js. We used RxJS to take advantage of Observables and treat the event handling and asynchronous calls on blockchain as observable streams. And NgRx gives us the ability to structure our code into well-organized components and manage our application state in sync with the smart contract state.

Appendix A

The smart contract test scenarios

```
var chai = require('chai');
chai.use(require('chai-as-promised')).should();

const BN = web3.utils.BN;
// Enable and inject BN dependency
chai.use(require('chai-bn')(BN));

var expect = chai.expect;
var assert = chai.assert;
var should = chai.should();

const { expectRevert, expectEvent } = require('@openzeppelin/test-helpers');
const { getCurrentTime } = require('./helpers/time');

const FleaMarketFactory = artifacts.require("../contracts/FleaMarketFactory.sol");
const SafeRemotePurchase = artifacts.require("../contracts/SafeRemotePurchase.sol");

// custom function to calculate amount of ether spent on the transaction
// here txInfo is the transaction results
async function getGasCoast(txInfo) {
     const tx = await web3.eth.getTransaction(txInfo.tx);
     const gasCost =
       (new BN(tx.gasPrice)).mul(new BN(txInfo.receipt.gasUsed));
     return gasCost;
}

contract("FleaMarketFactory", accounts => {

     const [deployer, seller, buyer, buddy] = accounts;
     const IPFS_HASH = "QmdXUfpqeGQyvJ6xVouPLR65XtNp63TUHM937zPvg9dFrT";

     // display three test accounts
     console.log(`deployer account: ${deployer}`);
```

```
console.log(`seller account: ${seller}`);
console.log(`buyer account: ${buyer}`);

describe('deployment of FleaMarketFactory contract', async () => {
  let factory;
  before(async () => {
    factory = await FleaMarketFactory.new();
  });

  beforeEach(async () => {
    const time = await getCurrentTime();
  console.log(`current time: ${time}`);
  });

  it("deployed successfully", async () => {

    const address = await factory.address;
    console.log(`contract address: ${address}`);

    //make sure the address is real
    assert.notEqual(address, 0x0);
    assert.notEqual(address, "");
    assert.notEqual(address, null);
    assert.notEqual(address, undefined);
  });

  it('it has the owner who is the deployer', async () => {
    const owner = await factory.owner()
    assert.equal(owner, deployer)
  })
})

describe('creating a new instance of SafeRemotePurchase contract',
  async () => {
let factory;

 beforeEach(async () => {
  const time = await getCurrentTime();
  console.log(`current time: ${time}`);
  factory = await FleaMarketFactory.new();
 });

 it('should create a valid product', async () => {
    const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
    const wei = web3.utils.toWei('1.4', 'Ether');
```

```
    const commission = new BN(350);

    const receipt = await factory.createPurchaseContract
      (bytes32Key, 'Tesla Cybertruck', IPFS_HASH, commission, {
        from: seller,
        value: wei
    });

    expect(await factory.getContractCount())
      .to.be.a.bignumber.that.equal(new BN(1));

    const address = await factory.getContractByKey(bytes32Key);
    expectEvent(receipt, 'LogCreatePurchaseContract', {
      sender: seller,
      contractAddress: address
      // key: bytes32Key
    });

    const logData = receipt.logs[2];
    const eventData = logData.args;
    assert.equal(web3.utils.hexToUtf8(eventData.key), 'teslaCybertruck-X01',
      "LogCreatePurchaseContract event logged did not have expected
      product key");
})

it('should not create a product for empty key', async () => {

    const bytes32Key = web3.utils.utf8ToHex('');
    const wei = web3.utils.toWei('1.4', 'Ether');
    const commission = web3.utils.toBN(350);

    await factory.createPurchaseContract(bytes32Key,
      'Tesla Cybertruck', IPFS_HASH, commission, {
        from: seller,
        value: wei
    }).should.be.rejected;
})

it('should not create a product with the same key', async () => {

    const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
    const wei = web3.utils.toWei('1.4', 'Ether');
    const commission = web3.utils.toBN(350);

    await factory.createPurchaseContract(bytes32Key,
```

```
    'Tesla Cybertruck', IPFS_HASH, commission, {
      from: seller,
      value: wei
  }).should.be.fulfilled;

  await factory.createPurchaseContract(bytes32Key,
    'Tesla Cybertruck II', IPFS_HASH, commission, {
      from: seller,
      value: wei
  }).should.be.rejected;
})

it('should not create product with zero commission', async () => {

  const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
  const wei = web3.utils.toWei('1.4', 'Ether');
  const commission = new BN(0);

  await factory.createPurchaseContract(bytes32Key,
    'Tesla Cybertruck', IPFS_HASH, commission, {
      from: seller,
      value: wei
  }).should.be.rejected;
})

it('should not able to create a product with zero purchase price',
  async () =>{

  const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
  const commission = new BN(350);

  await factory.createPurchaseContract(bytes32Key,
    'Tesla Cybertruck', IPFS_HASH, commission, {
      from: seller,
      value: 0
  }).should.be.rejected;
})

it('should not able to create a product having not even price', async () => {

  const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
  const commission = new BN(350);

  await factory.createPurchaseContract(bytes32Key,
    'Tesla Cybertruck', IPFS_HASH, commission, {
```

```
        from: seller,
        value: 31313131317
    }).should.be.rejected;
 })

})

describe('business logic for purchase and delivery the product', async () => {

  let product;

  const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
  const wei = web3.utils.toWei('1.4', 'Ether');
  const commission = web3.utils.toBN(350);

  beforeEach(async () => {
    const time = await getCurrentTime();
    console.log(`current time: ${time}`);

    const factory = await FleaMarketFactory.new();

    await factory.createPurchaseContract(bytes32Key,
      'Tesla Cybertruck', IPFS_HASH, commission, {
        from: seller,
        value: wei
    }).should.be.fulfilled;

    const address = await factory.getContractByKey(bytes32Key);
    //assert.notEqual(address, 0x0);
    address.should.not.equal(0x0);

    // get instance of the SafeRemotePurchase contract by address
    product = await SafeRemotePurchase.at(address);
 });

it('is a valid product', async () => {

  // validate key
  const key = await product.key();
  const keyAscii = web3.utils.hexToUtf8(key);
  keyAscii.should.equal('teslaCybertruck-X01');

  // validate seller
  expect(await product.seller()).to.equal(seller);
```

```
  // validate owner
  expect(await product.owner()).to.equal(deployer);

  // validate price
  const price = (new BN(wei)).div(new BN(2));
  expect(await product.price()).to.be.a.bignumber.that.equal(price);

  // validate ballance  -  balance has to be 2x price
  expect(await product.balanceOf()).to.be.a.bignumber.that.equal
    (price.mul(new BN(2)));

  // validate state - should be Created
  //note that Solidity enum are converted explicitly to uint ==> will be
  //retrieved from web3 as BN. the enum values start from 0.
  expect(await product.state()).to.be.a.bignumber.that.equal(new BN(0));

  // check for commission value
  expect(await product.commissionRate({
    from: deployer
  })).to.be.a.bignumber.that.equal(commission);

})

it('the buyer should be able to purchase and confirm delivery of product',
  async () => {

  // Buyer makes purchase (put 2x of price)
  let txInfo = await product.buyerPurchase({
    from: buyer,
    value: wei
  }).should.be.fulfilled;

  expectEvent(txInfo, 'LogPurchaseConfirmed', {
    sender: buyer,
    amount: wei   //could be an instance of BN or string
  });

  // validate buyer
  expect(await product.buyer()).to.equal(buyer);

  // validate state - should be Locked
  expect(await product.state()).to.be.a.bignumber.that.equal(new BN(1));

  // validate smart contract ballance
  // balance has to be 4x price  =
```

```
  (2x from the seller and 2x from the buyer)
  expect(await product.balanceOf())
    .to.be.a.bignumber.that.equal((new BN(wei)).mul(new BN(2)));

  // Buyer confirm delivery
  const buyerBalanceBefore = new BN(
    await web3.eth.getBalance(buyer)
  );

  const price = (new BN(wei)).div(new BN(2));
  txInfo = await product.buyerConfirmReceived({
    from: buyer
  }).should.be.fulfilled;

  expectEvent(txInfo, 'LogReceivedByBuyer', {
    sender: buyer,
    amount: price   // could be an instance of BN or string
  });

  const buyerBalanceAfter = new BN(
    await web3.eth.getBalance(buyer)
  );

  // calculate amount money the buyer spend on the transaction
  const gasCost = await getGasCoast(txInfo);

  // validate that the buyer gets his escrow money back
  expect(buyerBalanceAfter).to.be.a.bignumber.that.equal
    (buyerBalanceBefore.add(price).sub(gasCost));

  // validate state - should be ItemReceived
  expect(await product.state())
    .to.be.a.bignumber.that.equal(new BN(3));

  // validate smart contract ballance
  // balance has to be 3x price  = (1x went back to the buyer)
  expect(await product.balanceOf())
    .to.be.a.bignumber.that.equal(price.mul(new BN(3)));
})

it('the seller and then the deployer should be able to
    withdraw their money', async () => {

  // Buyer makes purchase (put 2x of price)
  await product.buyerPurchase({
```

```
  from: buyer,
  value: wei
})
// buyer confirm delivery
await product.buyerConfirmReceived({
  from: buyer
});

const sellerBalanceBefore = new BN(
  await web3.eth.getBalance(seller)
);

// seller withdraw his escrow money
let txInfo = await product.withdrawBySeller({
  from: seller
}).should.be.fulfilled;

expectEvent(txInfo, 'LogWithdrawBySeller', {
  sender: seller
});

const sellerBalanceAfter = new BN(
  await web3.eth.getBalance(seller)
);

// calculate amount money the seller spent on the transaction
let gasCost = await getGasCoast(txInfo);

const price = (new BN(wei)).div(new BN(2));
// calculate amount money the seller spent on the commission
const commissionCost = (price.mul(commission)).div(new BN(10000));

// validate the seller ballance after he withdraw escrow and
// purchased product money from the contract
expect(sellerBalanceAfter).to.be.a.bignumber.that.equal
  (sellerBalanceBefore.add(price.mul(new BN(3)))
  .sub(gasCost).sub(commissionCost));

// validate state - should be SellerPaid
expect(await product.state()).to.be.a.bignumber
  .that.equal(new BN(4));

// validate smart contract ballance
// balance has to have only the commission to be paid to the deployer
expect(await product.balanceOf())
```

```
    .to.be.a.bignumber.that.equal(commissionCost);

  const deployerBalanceBefore = new BN(
    await web3.eth.getBalance(deployer)
  );

  // deployer withdraw his commission
  txInfo = await product.withdrawByOwner({
    from: deployer
  }).should.be.fulfilled;

  expectEvent(txInfo, 'LogWithdrawByOwner', {
    sender: deployer,
    amount: commissionCost
  });

  const deployerBalanceAfter = new BN(
    await web3.eth.getBalance(deployer)
  );

  gasCost = await getGasCoast(txInfo);

  // validate the deployer ballance
  expect(deployerBalanceAfter).to.be.a.bignumber
    .that.equal(deployerBalanceBefore.add(commissionCost)
    .sub(gasCost));

  // validate state - should be Completed
  expect(await product.state()).to.be.a.bignumber
    .that.equal(new BN(6));

  // validate smart contract ballance => has to be zero
  expect(await product.balanceOf())
    .to.be.a.bignumber.that.equal(new BN(0));
})

it('the deployer and then the seller should be able
    to withdraw their money', async () => {

  // Buyer makes purchase (put 2x of price)
  await product.buyerPurchase({
    from: buyer,
    value: wei
  })
  // buyer confirm delivery
```

```
await product.buyerConfirmReceived({
  from: buyer
});

const deployerBalanceBefore = new BN(
  await web3.eth.getBalance(deployer)
);

// deployer withdraw his commission
let txInfo = await product.withdrawByOwner({
  from: deployer
}).should.be.fulfilled;

// calculate the commission
const price = (new BN(wei)).div(new BN(2));
const commissionCost = (price.mul(commission)).div(new BN(10000));

expectEvent(txInfo, 'LogWithdrawByOwner', {
  sender: deployer,
  amount: commissionCost
});

const deployerBalanceAfter = new BN(
  await web3.eth.getBalance(deployer)
);

let gasCost = await getGasCoast(txInfo);

// validate the deployer ballance
expect(deployerBalanceAfter).to.be.a.bignumber
  .that.equal(deployerBalanceBefore.add(commissionCost)
  .sub(gasCost));

// validate state - should be OwnerPaid
expect(await product.state()).to.be.a.bignumber.that.equal(new BN(5));

// validate smart contract ballance =>
// should be 3x price minus commission paid to the deployer
expect(await product.balanceOf()).to.be.a.bignumber
  .that.equal(price.mul(new BN(3)).sub(commissionCost));

const sellerBalanceBefore = new BN(
  await web3.eth.getBalance(seller)
);
```

```
  // seller withdraw his escrow money
  txInfo = await product.withdrawBySeller({
    from: seller
  }).should.be.fulfilled;

  expectEvent(txInfo, 'LogWithdrawBySeller', {
    sender: seller
  });

  const sellerBalanceAfter = new BN(
    await web3.eth.getBalance(seller)
  );

  // calculate amount money the seller spent on the transaction
  gasCost = await getGasCoast(txInfo);

  // validate the seller ballance after he withdraw escrow
  // and the purchased product money from the contract
  expect(sellerBalanceAfter).to.be.a.bignumber
    .that.equal(sellerBalanceBefore.add(price.mul(newBN(3)))
    .sub(gasCost).sub(commissionCost));

  // validate state - should be SellerPaid
  expect(await product.state()).to.be.a.bignumber.that.equal(new BN(6));

  // validate smart contract ballance
  // balance has to be zero
  expect(await product.balanceOf()).to.be.a.bignumber
    .that.equal(new BN(0));
})

it('the seller should be able to cancel a purchase contract
     and reclaim the escrow', async () => {

  const sellerBalanceBefore = new BN(
    await web3.eth.getBalance(seller)
  );

  txInfo = await product.abortBySeller({
    from: seller
  }).should.be.fulfilled;

  expectEvent(txInfo, 'LogCanceledBySeller', {
    sender: seller,
    amount: wei
```

```
    });

    const sellerBalanceAfter = new BN(
      await web3.eth.getBalance(seller)
    );

    // calculate amount money the seller spend on the transaction
    const gasCost = await getGasCoast(txInfo);

    // validate that the seller gets his escrow money back
    expect(sellerBalanceAfter).to.be.a.bignumber
      .that.equal(sellerBalanceBefore.add(new BN(wei))
      .sub(gasCost));

    // validate state - should be Canceled
    expect(await product.state()).to.be.a.bignumber
      .that.equal(new BN(2));

    // validate smart contract ballance - has to be 0
    expect(await product.balanceOf()).to.be.a.bignumber
      .that.equal(new BN(0));
  })
})

describe('some failure cases', async () => {

  let product;

  const bytes32Key = web3.utils.utf8ToHex('teslaCybertruck-X01');
  const wei = web3.utils.toWei('1.4', 'Ether');
  const commission = web3.utils.toBN(350);

  beforeEach(async () => {
    const time = await getCurrentTime();
    console.log(`current time: ${time}`);

    const factory = await FleaMarketFactory.new();

    await factory.createPurchaseContract(bytes32Key, 'Tesla Cybertruck',
       IPFS_HASH, commission, {
         from: seller,
         value: wei
    }).should.be.fulfilled;

    const address = await factory.getContractByKey(bytes32Key);
```

```
  //assert.notEqual(address, 0x0);
  address.should.not.equal(0x0);

  // get instance of the SafeRemotePurchase contract by address
  product = await SafeRemotePurchase.at(address);
});

it('the seller should not be able to cancel a purchase contract
    in the state other than Created', async () => {

    // Buyer makes purchase (put 2x of price)
    await product.buyerPurchase({
      from: buyer,
      value: wei
    })

    // the seller is trying to Abort contract - should be rejected
    await product.abortBySeller({
      from: seller
    }).should.be.rejected;
})

it('the buyer is trying to purchase a product with not enough ether',
    async() => {

    // Buyer makes purchase (must deposit 2x of price)
    await product.buyerPurchase({
      from: buyer,
      value: web3.utils.toWei('1', 'Ether')
    }).should.be.rejected;
})

it('someone else is trying to withdraw money
    after the buyer confirms the delivery', async () => {

    // Buyer makes purchase (put 2x of price)
    await product.buyerPurchase({
      from: buyer,
      value: wei
    })

    // buyer confirm delivery
    await product.buyerConfirmReceived({
      from: buyer
    });
```

```
        // request to withdraw came from the wrong account
        await product.withdrawBySeller({
          from: buddy
        }).should.be.rejected;
      })

      it('should reject if someone trying to view commission rate',
        async () => {

        // only deployer and seller allow
        await product.commissionRate({
          from: buddy
        }).should.be.rejected;
      })

      it('should reject if someone send ether to a purchase contract',
        async () => {

        await web3.eth.sendTransaction({ from: buddy, to: product.address,
          value: web3.utils.toWei('0.005', "ether") }).should.be.rejected;
      })

      it('should reject if someone try to withdraw ether
        from a purchase contract', async () => {

        // notice, we can not provide a contract type address
        // in the 'from' parameter
        // otherwise someone would able to withdraw money from the contract
        // If we do, we get the error =>
        // 'Error: Returned error: sender account not recognized'
        await web3.eth.sendTransaction({ from: product.address, to: buddy,
          value: web3.utils.toWei('0.00005', "ether")}).should.be.rejected;
       })
    })
})
```

References

1. Best Practices for Smart Contract Development, by Yos Riady
2. Escrow Service as a Smart Contract: The Business Logic, by Jackson Ng
3. Creating Smart Contracts with Smart Contract, by Jackson Ng
4. Breaking Changes to the MetaMask Inpage Provider, by Erik Marks
5. No Longer Injecting web3.js, by Bobby Dresser
6. Solidity CRUD- Epilogue, by Rob Hitchens
7. Solidity Security: Comprehensive list of known attack vectors and common antipatterns, by Adrian Manning
8. Common Smart Contract Vulnerabilities and How To Mitigate Them, by Yos Riady
9. The Encyclopedia of Smart Contract Attacks and Vulnerabilities, by Kaden Zipfel
10. Intro to Solidity Linting and Formatting, by Daniel Onggunhao
11. Deploying Smart Contracts with Truffle, by Sam Benemerito
12. Truffle: Smart Contract Compilation & Deployment, by Josh Cassidy
13. 5-minute guide to deploying smart contracts with Truffle and Ropsten, by Nicole Zhu
14. RPC Access to Ethereum with Infura, by Jackson Ng
15. Getting Started with Metamask!, by Markus Viola
16. Developing for Ethereum: Getting Started with Ganache, by Bruno Škvorc
17. Using Ganache with Remix and Metamask, by Kacharlabhargav
18. JavaScript — Unit Testing using Mocha and Chai, by NC Patro
19. The Ultimate Blockchain Programming Tutorial with Ethereum, Solidity & Web3.js, by Gregory McCubbin
20. Truffle: Testing your smart contract, by Josh Cassidy
21. Angular CLI: Getting Started Guide, by Amadou Sall
22. Expecting the Unexpected – Best practices for Error handling in Angular, by Michael Karen
23. Error Handling & Angular, by Aleix Suau
24. Angular NgRx Material Starter, by Tomas Trajan
25. NgRx: Action Creators redesigned, by Alex Okrushko
26. Announcing NgRx Version 8: @ngrx/data, creator functions, run-time checks, and isolated tests, by Tim Deschryver
27. NgRx + Loading Indicator, by Brian Love

28. Using Angular Material Spinner with CDK Overlay by Velen Aranha
29. NGRX Store: Understanding State Selectors, by Todd Motto
30. Handling Error States with NgRx, by Brandon Roberts
31. NgRx: How and where to handle loading and error states of AJAX calls?, by Alex Okrushko
32. Start using ngrx/effects for this, by Tim Deschryver
33. ethers.js — Version 4.0 Release, by RicMoo
34. Build an Ethereum DApp Using Ethers.js, by Mahesh Murthy
35. Ethers and Angular, by GrandSchtroumpf
36. Angular HttpClient Blob, by Brian Love
37. Creating a File Upload Component in Angular, by Lukas Marx
38. Using CanLoad guard to prevent or allow a module from loading in Angular, by Venod Vemuru
39. Managing File Uploads With NgRx, by Wes Grimes
40. IPFS and Angular 6, by GrandSchtroumpf
41. Ethereum + IPFS + React DApp Tutorial, by Alexander Ma
42. RxJS Observable interop with Promises and Async-Await, by Ben Lesh
43. Youtube On IPFS in 5 mins, by vasa
44. Total Guide To Angular 6+ Dependency Injection — providedIn vs providers:[ ], by Tomas Trajan
45. NGRX Store + Effects, by Todd Motto
46. 5 useful NgRx effects that don't rely on actions, by Ferdinand Malcher

Licenses

Icons made by Eucalyp from www.flaticon.com is licensed by CC 3.0 BY

Icons made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Icons made by photo3idea_studio from www.flaticon.com is licensed by CC 3.0 BY

Icons made by prettycons from www.flaticon.com is licensed by CC 3.0 BY

Lions in different actions illustration Free Vector by Freepik

Big thanks to Kire Thomas for helping with graphics and styling.

Architecture Ethereum DApp with Angular, Angular Material and NgRx

The book is a step-by-step guide demonstrating how to build a modern Ethereum blockchain DApp with Ethers.js and IPFS using Angular and NgRx. You'll learn first hand how to harness the power of these exciting technologies by building your complete application named FleaMarket that runs on Ethereum blockchain and implements the functionality of the Safe Remote Purchase contract. The entire application is built with the VS code. All libraries used in the source code are updated to the latest stable release.

Alex Yevseyevich is an application developer with more than 20 years experience. His main interest is in modern web development and blockchain smart contract technologies.

Email: alexanddanik@outlook.com
LinkedIn: www.linkedin.com/in/alex-tiger

Daniel Yevseyevich currently attends Nova Southeastern University as a Biology Major with a minor in Finance. He has a passion for programming and implementing it into the new cryptocurrency trend.

Email: DanielYevseyevich@gmail.com
LinkedIn: https://www.linkedin.com/in/dan-yevseyevich