<https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html>

Smart Contract Processing

스마트 컨트랙트는 블록체인 네트워크의 심장이다. PaperNet에서, commercial paper smart contract안에 있는 코드는 commercial paper의 유효한 상태와 paper의 상태를 전환시키는 트랜잭션 로직을 정의한다. 이 토픽에서 우리는 너희에게 commercial paper를 issuing하고 buying라고 redeeming하는 프로세스를 관리하는 현실세계의 스마트 컨트랙트를 어떻게 구현하는지를 보여줄 것이다.

다룰 것 :

-스마트 컨트랙트가 무엇인지, 이것이 왜 중요한지

-어떻게 스마트 컨트랙트를 정의하는지

-트랜잭션을 어떻게 정의하는지

-어떻게 트랜잭션을 구현하는지

-어떻게 스마트 컨트랙트에서 비즈니스 대상을 표현하는지

-어떻게 장부에 대상을 저장하고 수정하는지

Smart Contract

스마트 컨트랙트는 비즈니스 대상의 다양한 상태들을 정의하고 이 다양한 상태들 사이에서 특정대상을 옮기는 과정들을 관리한다. 스마트 컨트랙트는 건축가와 스마트 컨트랙트 개발자들이 블록체인 네트워크에서 협업하는 여러 조직에서 공유되는 주요 비즈니스 과정과 데이터를 정의하도록 허용하기 때문에 중요하다.

PaperNet 네트워크에서, 스마트 컨트랙트는 MagnetoCorp와 DigiBank같은 다양한 네트워크 참여자들에 의해 공유된다. 그들이 공동으로 같은 공유된 비즈니스 과정과 데이터를 구현하려면 동일한 버전의 스마트 컨트랙트가 네트워크에 연결된 모든 사용자들에 의해 사용되어야 한다.

Implementation Languages

지원되는 런타임은 두가지(Java Virtual Machine, Node.js)가 있다. 이것들은 JavaScript, TypeScript, Java 중 하나, 또는 이 지원되는 런타임들 중 하나 위에서 돌아갈 수 있는 다른 언어를 사용할 기회를 준다.

Java와 TypeScript에서, 주석 또는 데코레이션은 스마트 컨트랙트와 이것의 구조에 대한 정보를 제공하기 위해 사용된다. 이를 통해 더 풍부한 개발경험이 가능해진다. – 예를 들어, 저자 정보나 반환유형을 적용할 수 있다. JavaScript에서는, 규칙을 따라야 한다. 그러므로 그들은 자동적으로 결정될 수 있는 것에 대한 제한이 있다.

Contract class

paperNet commercial paper 스마트 컨트랙트의 사본은 단일 파일에 포함되어 있다.

* Papercontract.js – JavaScript version
* commercialPaperContract.java – Java version

당신은 파일경로로부터 이것이 MagnetoCorp의 스마트 컨트랙트 사본임을 알 수도 있다. MagnetoCorp와 DigiBank는 사용할 스마트 컨트랙트의 버전에 동의해야만 한다. 지금, 어떤 조직의 사본은 사용할지는 문제가 아니다. 다 같다.

파일의 맨 위에, commercial paper 스마트 컨트랙트의 정의를 볼 수 있다.

@Contract**(...)**

@Default

**public** **class** **CommercialPaperContract** **implements** ContractInterface **{...}**

commercialPaperContract 클래스는 commercial paper의 트랜잭션(issue, buy, redeem)의 정의들을 포함한다. 이 트랜잭션들은 commercial paper를 실재로 가져오고 그들의 생명주기를 통해 이동시킨다.

Java에서, 클래스는 반드시 @Contract(…) 주석과 함께 꾸며져야 한다. 이것은 컨트랙트에 라이센스나 저자 같은 추가적인 정보를 제공할 기회를 제공한다. @Default() 주석은 이 컨트랙트 클래스가 기본 컨트랙트 클래스임을 나타낸다. 컨트랙트 클래스를 기본 컨트랙트 클래스로 표시할 수 있는 것은 여러 컨트랙트 클래스를 가진 몇몇 스마트 컨트랙트에 유용하다.

**import** org.hyperledger.fabric.contract.Context**;**

**import** org.hyperledger.fabric.contract.ContractInterface**;**

**import** org.hyperledger.fabric.contract.annotation.Contact**;**

**import** org.hyperledger.fabric.contract.annotation.Contract**;**

**import** org.hyperledger.fabric.contract.annotation.Default**;**

**import** org.hyperledger.fabric.contract.annotation.Info**;**

**import** org.hyperledger.fabric.contract.annotation.License**;**

**import** org.hyperledger.fabric.contract.annotation.Transaction**;**

우리의 commercial paper 컨트랙트는 이 클래스들의 내장된 기능들을 사용할 것이다. 예를 들어, 자동 메소드 호출, 트랜잭션 마다의 문법, 트랜잭션 핸들러, 클래스 공유상태 등

constructor() {

**super**('org.papernet.commercialpaper');

}

자바 클래스에서는, @contract() 주석에 명시적 contract 이름을 지정할 수 있으므로 생성자가 비어있다. 만약 이것이 없다면, 클래스의 이름이 사용된다.

가장 중요하게도, org.papernet.commercialpaper는 매우 서술적이다. – 이 스마트 컨트랙트는 모든 paperNet 조직들에게 동의 된 commercial paper의 정의이다.

일반적으로 파일당 단 하나의 스마트 컨트랙트가 있을것이다. – 컨트랙트들은 서로 다른 생명주기들을 갖는 경향이 있으므로, 이들을 분리하는 것이 합리적이다. 그러나, 몇몇 경우에는, 여러 스마트 컨트랙트들은 애플리케이션에 대한 구문 도움말을 제공할 수 있다. 그러한 경우들에는, 스마크 컨트랙트들과 트랜잭션들은 명확해질 수 있다.

Transaction definition

클래스내에서 issue 메소드를 찾아라.

자바 주석 @Transaction은 이 메소드를 트랜잭션 정의로 표시하는데 사용된다. 이 함수는 이 컨트랙트가 commercial paper를 발행(issue)하도록 호출될 때마다 제어된다. 다음 거래를 통해 commercial paper 00001이 어떻게 만들어졌는지 상기시켜라.

Txn **=** issue

Issuer **=** MagnetoCorp

Paper **=** 00001

Issue time **=** 31 May 2020 09:00:00 EST

Maturity date **=** 30 November 2020

Face value **=** 5M USD

우리는 변수이름을 프로그래밍 스타일로 바꿨는데, 이 속성들이 어떻게 issue 메소드의 변수들에 매칭되는지 봐라.

Issue 메소드는 애플리케이션이 commercial paper를 발행(issue)하도록 요청할 때마다 컨트랙트에 의해 자동적으로 제어를 받는다. 트랜잭션 속성값들은 해당 변수들을 통해 메소드에 사용 가능해진다.

Issue 정의에서 추가적인 변수를 봤을지도 모른다. – ctx 이것은 트랜잭션 문법(transaction context)이라 불리고 항상 첫번째이다. 기본적으로, 이것은 트랜잭션 로직에 연관된 컨트랙트 별 및 트랜잭션 별의 정보를 유지 보수한다. 예를 들어, 이것은 장부 API에 대한 접근권한 뿐만 아니라 MagnetoCorp의 명시된 트랜잭션 식별자와 MagnetoCorp 발급 사용자의 디지털 인증서를 포함한다.

어떻게 스마트 컨트랙트가 기본 구현을 수용하지 않고 자체의 createContext() 메소드를 구현하여 기본 트랜잭션 문법을 확장시키는지 봐라.

이 확장된 문법이 디폴트에 paperList라는 맞춤속성을 더한다.

우리는 곧 ctx.paperList를 사용하여 모든 paperNet commercial paper를 저장하고 검색하는 방법을 볼 것이다.

스마트 컨트랙트 트랜잭션의 구조에 대한 당신의 이해를 강화하기 위해, buy와 redeem(다시 팜) 트랜잭션의 정의들을 찾고, 어떻게 그들이 대응되는 commercial paper 트랜잭션들에 매핑되는지 확인한다.

-The buy transaction:

Txn **=** buy

Issuer **=** MagnetoCorp

Paper **=** 00001

Current owner **=** MagnetoCorp

New owner **=** DigiBank

Purchase time **=** 31 May 2020 10:00:00 EST

Price **=** 4.94M USD

<Java>

@Transaction

**public** CommercialPaper **buy(**CommercialPaperContext ctx**,**

String issuer**,**

String paperNumber**,**

String currentOwner**,**

String newOwner**,**

**int** price**,**

String purchaseDateTime**)** **{...}**

-The redeem transaction:

Txn **=** redeem

Issuer **=** MagnetoCorp

Paper **=** 00001

Redeemer **=** DigiBank

Redeem time **=** 31 Dec 2020 12:00:00 EST

<Java>

@Transaction

**public** CommercialPaper **redeem(**CommercialPaperContext ctx**,**

String issuer**,**

String paperNumber**,**

String redeemingOwner**,**

String redeemDateTime**)** **{...}**

두 경우 모두에서, commercial paper 트랜잭션과 스마트 컨트랙트 메소드 정의 사이에 1:1 대응을 볼 수 있다.

Tracsaction logic

지금까지 컨트랙트들이 구성되고 트랜잭션들이 정의되었다. 이제 스마트 컨트랙트안의 로직에 집중하자.

-issue transaction

Txn **=** issue

Issuer **=** MagnetoCorp

Paper **=** 00001

Issue time **=** 31 May 2020 09:00:00 EST

Maturity date **=** 30 November 2020

Face value **=** 5M USD

이것은 issue 메소드가 제어를 통과하게 한다.

<Java>

@Transaction

**public** CommercialPaper **issue(**CommercialPaperContext ctx**,**

String issuer**,**

String paperNumber**,**

String issueDateTime**,**

String maturityDateTime**,**

**int** faceValue**)** **{**

System**.**out**.**println**(**ctx**);**

*// create an instance of the paper*

CommercialPaper paper **=** CommercialPaper**.**createInstance**(**issuer**,** paperNumber**,** issueDateTime**,** maturityDateTime**,**

faceValue**,**issuer**,**""**);**

*// Smart contract, rather than paper, moves paper into ISSUED state*

paper**.**setIssued**();**

*// Newly issued paper is owned by the issuer*

paper**.**setOwner**(**issuer**);**

System**.**out**.**println**(**paper**);**

*// Add the paper to the list of all similar commercial papers in the ledger*

*// world state*

ctx**.**paperList**.**addPaper**(**paper**);**

*// Must return a serialized paper to caller of smart contract*

**return** paper**;**

**}**

로직은 간단하다. 트랜잭션의 입력 변수를 취하고, 새로운 commercial paper인 paper를 만들고, 이것을 paperList를 사용하여 모든 commercial paper들의 리스트에 추가하고, 새로운 commercial paper를 트랜잭션의 응답으로 반환한다.

Commercial paper의 리스트에 대한 접근 권한을 제공하기 위해 paperList가 트랜잭션 문법으로부터 검색되는 방법을 봐라. Issue(), buy(), redeem()은 commercial paper의 리스트를 최신의 상태로 유지하기 위해 지속적으로 ctx.paperList에 재접속한다.

-buy transaction

<Java>

@Transaction

**public** CommercialPaper **buy(**CommercialPaperContext ctx**,**

String issuer**,**

String paperNumber**,**

String currentOwner**,**

String newOwner**,**

**int** price**,**

String purchaseDateTime**)** **{**

*// Retrieve the current paper using key fields provided*

String paperKey **=** State**.**makeKey**(new** String**[]** **{** paperNumber **});**

CommercialPaper paper **=** ctx**.**paperList**.**getPaper**(**paperKey**);**

*// Validate current owner*

**if** **(!**paper**.**getOwner**().**equals**(**currentOwner**))** **{**

**throw** **new** RuntimeException**(**"Paper " **+** issuer **+** paperNumber **+** " is not owned by " **+** currentOwner**);**

**}**

*// First buy moves state from ISSUED to TRADING*

**if** **(**paper**.**isIssued**())** **{**

paper**.**setTrading**();**

**}**

*// Check paper is not already REDEEMED*

**if** **(**paper**.**isTrading**())** **{**

paper**.**setOwner**(**newOwner**);**

**}** **else** **{**

**throw** **new** RuntimeException**(**

"Paper " **+** issuer **+** paperNumber **+** " is not trading. Current state = " **+** paper**.**getState**());**

**}**

*// Update the paper*

ctx**.**paperList**.**updatePaper**(**paper**);**

**return** paper**;**

**}**

어떻게 Paper.setOwner(newOwner)로 주인을 변경하기 전에 트랜잭션이 currentOwner를 확인하고 그 용지가 TRADING 되는지 봐라. 기본 흐름은 간단하다. – 몇몇 전제조건을 확인하고, 새로운 주인을 설정하고, 장부에 commercial paper를 업데이트하고, 트랜잭션의 응답으로 업데이트된 commercial paper를 반환한다.

Representing an Objeaddct

지금까지 CommercialPaper와 PaperList 클래스들을 활용하여 issue, buy, redeem 트랜잭션들을 정의하고 구현하는 방법들을 봤다. 이 틀래스들이 어떻게 작동하는지 보자.

CommercialPaper.java 파일에서 ConercialPaper 클래스를 찾아라.

@DataType**()**

**public** **class** **CommercialPaper** **extends** State **{...}**

이 클래스는 commercial paper 상태의 메모리 내 표현이 포함된다. createInstance 메소드가 제공된 매개변수들을 사용하여 새로운 commercial paper를 초기화하는 방법을 봐라.

**public** **static** CommercialPaper **createInstance(**String issuer**,** String paperNumber**,** String issueDateTime**,**

String maturityDateTime**,** **int** faceValue**,** String owner**,** String state**)** **{**

**return** **new** CommercialPaper**().**setIssuer**(**issuer**).**setPaperNumber**(**paperNumber**).**setMaturityDateTime**(**maturityDateTime**)**

**.**setFaceValue**(**faceValue**).**setKey**().**setIssueDateTime**(**issueDateTime**).**setOwner**(**owner**).**setState**(**state**);**

**}**

CommercialPaper paper **=** CommercialPaper**.**createInstance**(**issuer**,** paperNumber**,** issueDateTime**,** maturityDateTime**,**

faceValue**,**issuer**,**""**);**

issue 트랜잭션이 호출될 때마다, commercial paper의 새로운 상용메모리 인스턴스가 트랜잭션 데이터를 포함하면서 생성되는 방법을 봐라.

commercialPaper 클래스는 state 클래스를 확장한다. State는 상태에 대한 공통 추상화를 만드는 애플리케이션 정의 클래스이다. 모든 상태들은 복합 키, 직렬화 및 역직렬화 등을 나태내는 비즈니스 오브젝트 클래스가 있습니다. State는 우리가 장부에 하나 이상의 비즈니스 오브젝트를 저장할 때 우리의 코드를 더 읽기 쉽게 도와준다. State.js 파일에 state 클래스를 봐라.

페이퍼는 이것이 만들어졌을 때 자신의 키를 계산한다. – 이 키는 장부가 접속되었을 때 사용된다. 이 키는 issuer와 paperNumber의 조합으로 만들어진다.

constructor(obj) {

**super**(CommercialPaper.getClass(), [obj.issuer, obj.paperNumber]);

Object.assign(**this**, obj);

}

Paper는 paper 클래스가 아닌 트랜잭션에 의해 ISSUED 상태로 옮겨진다. 이것은 스마트 컨트랙트가 페이퍼의 생활주기를 관리하기 때문이다. 예를 들어, import 트랜잭션은 TRADING 상태에 있는 새로운 페이퍼의 세트를 직접 만든다.

CommercialPaper 클래스의 나머지는 간단한 도움 메소드를 포함한다.

getOwner() {

**return** **this**.owner;

}

이와 같은 메소드들이 스마트 컨트랙트에 의해 이것의 생명주기를 통해 commercial paper로 옮겨진 방법을 떠올려라. 예를 들어, redeem 트랜잭션에서 우리가 봤던 것.

**if** (paper.getOwner() **===** redeemingOwner) {

paper.setOwner(paper.getIssuer());

paper.setRedeemed();

}

Access the ledger

이제 paperlist.js 파일에서 PaperList 클래스를 찾아라.

**class** PaperList **extends** StateList {

이 유틸리티 클래스는 Hyperledger Fabric 상태 데이터베이스 안에 있는 모든 PaperNet commercial paper를 관리하는데 사용된다.

CommercialPaper 클래스처럼, 이 클래스는 애플리케이션-정의 StateList 클래스를 확장하여 상태들의 리스트에 대한 공동 추상화를 만든다. – 이 경우, PaperNet의 모든 commercial papers

addPaper() 메소드는 stateList.addState() 메소드에 대한 간단한 겉치장?이다.

async addPaper(paper) {

**return** **this**.addState(paper);

}

stateList.js 파일에서 어떻게 stateList 클래스가 장부에 commercial paper를 상태 데이터로 작성하기위해 Fabric API putState()를 사용하는지 볼 수 있다.

async addState(state) {

**let** key **=** **this**.ctx.stub.createCompositeKey(**this**.name, state.getSplitKey());

**let** data **=** State.serialize(state);

await **this**.ctx.stub.putState(key, data);

}

장부에 있는 모든 상태 데이터의 조각은 이 두 기본 요소를 요구한다.

* Key : key는 고정된 이름과 state의 key를 사용하여 createCompositeKey()로 만들어진다. 이름은 PaperList 객체가 구성될 때 할당되고 state.getSplitKey()는 각 상태의 유일한 상태의 키를 결정한다.
* Data : data는 간단하게 commercial paper의 상태의 일련화된 양식이며, state.serialize() 유틸리티 메소드를 사용하여 만들어진다. State 클래스는 JSON 을 사용하여 데이터를 직렬화 및 직렬화 해제를 하고, PaperList 객체가 구성될 때 State의 비즈니스 객체 클래스는 필요에 따라 (우리의 경우에는 CommercialPaper) 다시 세팅한다.

StateList는 개별의 상태나 상태들의 전체 리스트의 그 어떤 것도 저장하지 않는 것을 주목하라. – 이것은 모든 상태를 Fabric 상태 데이터베이스에 위임한다. 이것은 중요한 디자인 패턴으로 Hyperledger Fabric에서의 장부 MVCC 충돌 가능성을 줄인다.

StateList에 getState()와 updateState() 메소드들도 같은 방법으로 작동한다.

async getState(key) {

**let** ledgerKey **=** **this**.ctx.stub.createCompositeKey(**this**.name, State.splitKey(key));

**let** data **=** await **this**.ctx.stub.getState(ledgerKey);

**let** state **=** State.deserialize(data, **this**.supportedClasses);

**return** state;

}

async updateState(state) {

**let** key **=** **this**.ctx.stub.createCompositeKey(**this**.name, state.getSplitKey());

**let** data **=** State.serialize(state);

await **this**.ctx.stub.putState(key, data);

}

그들이 장부에 접속하기 위해 어떻게 Fabric APIs인 putState()와 getState()와 createCompositeKey()를 사용하는지 봐라.