Table of Contents

1. Introduction

블록체인의 세계에 오신 것을 환영합니다! 블록체인은 비트코인이 전자통화의 대표주자로 안착하기까지 핵심적인 역할을 담당해 왔습니다. 현재 블록체인 기술은 비트코인과는 별도로 최고의 데이터 안전성과 신뢰성이 요구되는 금융, 의료, 관공서 등의 분야에서 활용되거나 검토 중에 있으며, 블록체인을 기반으로 그동안 존재하지 않던 서비스들이나 참신한 아이디어로 무장한 스타트업들이 계속해서 생겨나고 있습니다.

만약 블록체인 기술을 활용할 만한 획기적인 아이디어가 있어도, 우선 블록체인과 비트코인에 대해 이해하고 공부하기 위한 많은 노력이 선행되어야 했습니다. 그리고 블록체인은 원래 범용적인 기술이 아니라 비트코인을 지탱하기 위해 만들어진 구조이므로 이를 응용하기 위해서는 추가적인 노력을 들여 기술적 장벽을 넘어야 했습니다.

코인스택은 이러한 어려움 없이 여러분의 서비스와 블록체인을 연결해 주는 징검다리 역할을 하기 위해 만들어졌습니다. 즉, 코인스택은 간편하게 블록체인 기반 서비스를 개발할 수 있도록 도와주는 플랫폼입니다. 실제로 블로코에서 서비스 중인 클라우드스탬프는 모두 코인스택 기반으로 개발되었으며, 현재 제1금융권 은행 및 대형 카드사를 비롯한 대기업, 한국거래소, 각종 스타트업들이 블록체인 기반 서비스를 제공하기 위해 코인스택을 사용하고 있습니다. 블로코가 무료로 제공하고 있는 클라우드 기반 코인스택 서비스는 과제를 수행하는 학생들, 개인 개발자들이 편리하게 이용하고 있습니다.

블록체인에 관심은 많은데 아직 잘 모르신다고요? 그렇다면 이 문서와 함께 코인스택 홈페이지에서 제공하는 블록체인 소개와 튜토리얼 등도 살펴보시기 바랍니다. 하지만 블록체인 전문가가 되지 않더라도 코인스택과 함께라면 얼마든지 블록체인 기반 서비스를 만들 수 있다는 사실을 잊지 마세요!

코인스택은 많은 개발자들에게 친숙한 프로그래밍 언어인 자바, 자바스크립트, HTML5 기반 SDK를 제공하며, HTML5 기반의 웹플레이그라운드 를 통하여 편리한 개발 환경 또한 제공하고 있습니다.

지금 당장 시작해 보세요! 그리고 필요한 API나 기능이 있다면 헬프데스크 또는 개발자 커뮤니티에 제안해 주시길 바랍니다.

문서의 구성

이 문서에서는 코인스택 개발과 관련된 모든 것을 얻을 수 있습니다. Object, API 등의 각 항목마다 우측 상단의 탭을 통해 코인스택에서 공식적으로 제공하는 모든 프로그래밍 언어별 예제를 편리하게 참조할 수 있습니다.

각 장의 소개

Quick Start Guide

코인스택 환경설정, 설치, 인증방법 등을 간략히 소개합니다.

Getting Started

코인스택 클라이언트의 기본적인 사용 예제를 살펴볼 수 있습니다.

Authentication

코인스택 클라이언트의 인증 및 권한 획득 방식과 절차를 상세히 소개합니다.

Coinstack and Public Bitcoin Blockchain

비트코인 블록체인을 기반으로 하는 코인스택의 기본 구성요소 및 코인스택에서 제공하는 개념과 기능을 소개합니다.

Coinstack Stamping

코인스택에서 제공하는 Document Stamping 기능에 대해 소개합니다.

Coinstack Multisig

코인스택에서 제공하는 멀티 시그니처 기능에 대해 소개합니다.

Coinstack Open Assets

코인스택에서 제공하는 Open Asset 프로토콜 호환 기능에 대해 소개합니다.

API Reference

코인스택 API에서 객체로 취급하는 데이터 모델 Object와 코인스택 API에 대한 상세한 설명 및 사용 방법, 프로그래밍 언어별 예제를 참조할 수 있습니다.

프로그래밍 언어 지원

코인스택의 모든 기능은 REST API로 제공되며, 아래 환경에서 SDK를 설치하여 사용할 수 있습니다.

플랫폼	다운로드
Java	Maven Repository
Node.js	NPM JS
Meteor	ATMOSPHERE
HTML5	Coinstack CDN

Copyright

Company Info.

주식회사 블로코 (www.blocko.io)

경기 성남시 분당구 성남대로 331번길 8(정자동) 킨스타워 16층

Contact

- 기술지원: 031-8016-6253
- 팩스: 050-8054-6253
- E-mail: support@blocko.io

매뉴얼 정보

- 소프트웨어 버전: 코인스택 3.5
- 매뉴얼 버전: 1.0.0
- 발행일: 2018-04-01

2. Quick Start Guide

블로코에서는 웹브라우저 상에서 설치 없이 빠르게 코인스택 SDK를 경험해 볼 수 있는 웹플레이그라운드를 제공하고 있습니다. 본 문서에서는 플레이그라운드를 통해 비트코인 블록체인의 상태를 조회하고, 주소 생성 및 잔고를 조회하는 방법을 소개합니다. 각 과정에 대한 상세한 설명은 Coinstack Documentation의 다른 챕터들을 참조하시기 바랍니다. 코인스택을 활용하여 멋진 서비스를 개발해 보시기 바랍니다.

웹플레이그라운드 접속

크롬이나 인터넷익스플로러 등 웹브라우저를 통해 코인스택 웹플레이그라운드(https://playground.blocko.io/)에 접속합니다. 코드들은 웹페이지 상의 콘솔창에 입력하거나, 웹페이지 하단의 예제코드 항에 존재하는 RUN 버튼을 누르면 실행 가능합니다.

클라이언트 객체 생성

테스트용 API Access Key와 Secret Key를 사용하여 CoinStack 객체를 생성합니다. 실 사용을 위한 코인스택 API KEY 발급은 4장 Authentication을 참고 바랍니다.

```
var client = new CoinStack('c7dbfacbdf1510889b38c01b8440b1', '10e88e9904f29c98356fd2d12b26de');
console.log(client);
```

> 결과값은 JSON으로 리턴됩니다.
>
> ```
> {
>    "apiKey":"c7dbfacbdf1510889b38c01b8440b1",
>    "secretKey":"10e88e9904f29c98356fd2d12b26de",
>    "protocol":"https://",
>    "endpoint":"mainnet.cloudwallet.io",
>    "isBrowser":true
> }
> ```

최신 블록체인 상태 정보 조회

블록체인의 상태를 조회하여 최신 블록의 블록번호와 해시값을 출력해 봅니다.

```
client.getBlockchainStatus(function(err, status) {
    console.log(status);
});
```

> 결과값은 JSON으로 리턴됩니다.
>
> ```
> {
>    "best_block_hash":"000000000000000002f28ea6b32950216406511722839dd47752f3b06e1d025",
>    "best_height":433116
> }
> ```

새 비공개키 생성

임의의 비공개키를 생성해 봅니다. 본 예제의 결과에 나온 Secret Key는 테스트 용이며, **추후 본인의 개인 계정을 생성하고 발급받은 Secret Key는 절대 타인에게 공개되어서는 안됩니다.**

```
var privateKey = CoinStack.ECKey.createKey();
console.log(privateKey);
```

> 결과값은 문자열로 리턴됩니다.
>
> ```
> L3nkFqH4n9xoYFvEmEyg54utGogNdz1WA4fqRohMJ8VgkXpRvGs1
> ```

비공개키에서 주소 생성

지갑 주소를 생성해 봅니다. 이 주소는 외부에 공개 가능하며, 이 주소를 통해 비트코인을 전송받을 수 있습니다.

```
var address = CoinStack.ECKey.deriveAddress(privateKey);
console.log(address);
```

> 결과값은 문자열로 리턴됩니다.
>
> ```
> 1MVMj4Gr9e7U5D4ZLignuNLpv2cyKiHa2x
> ```

주소 잔고 조회

현재 주소의 비트코인 잔액을 조회하고 사토시 단위로 출력해 봅니다. (1 사토시 = 0.00000001 비트코인, 1 비트코인 = 1억 사토시) 본 예제의 주소는 새로 생성한 주소이므로 잔액이 0 비트코인(BTC)임을 확인할 수 있습니다.

```
client.getBalance(address,
function(err, balance) {
    console.log(CoinStack.Math.toBitcoin(balance) + ' BTC');
});
```

> 결과값은 문자열로 리턴됩니다.
>
> ```
> 0 BTC
> ```

3. Getting Started

본 문서에서는 코인스택을 사용하기 위한 클라이언트의 초기 환경 설정, SDK 설치, 구동 테스트 등을 상세히 소개합니다. 무료로 공개된 코인스택 서버는 클라우드 상에서 운영되고 있습니다. 만약 구축형(On-Premise)으로 별도의 설치가 필요한 경우 블로코의 기술지원이나 영업 채널로 문의 바랍니다.

API KEY 발급

코인스택을 사용하기 위해서는 우선 API KEY가 준비되어야 합니다. 코인스택 대시보드에서 계정을 생성한 후 인증키를 발급받을 수 있습니다. 대시보드의 [키 발급] 화면에서 [새로운 키 발급] 버튼을 누르고 [API Key]를 선택하면 Access Key와 Secret Key가 생성됩니다. **Secret Key는 대시보드에서 다시 확인할 수 없으므로** 키를 발급받을 때 화면에 표시된 Secret Key를 반드시 다른 곳에 저장해 두어야 합니다.

코인스택 인증 방식에 대한 자세한 사항은 4장 Authentication을 참고 바랍니다.

> 현재는 코인스택 대시보드 서비스가 중지되었습니다. API Key 영역은 비워두거나 임의의 문자열을 넣어도 작동 합니다.

Java

1. 자바 SDK 설치

코인스택 자바 SDK를 사용하려면 JDK 1.6 또는 상위 버전의 JDK가 필요합니다. Oracle Java SE Downloads 페이지에서 최신 버전의 JDK를 다운로드 받을 수 있습니다. 자세한 다운로드 및 설치방법은 공식 페이지를 참조하시기 바랍니다. 개발 편의를 위하여 환경변수(JAVA_HOME, PATH)를 설정 할 경우 환경변수 설정 페이지를 참조하시기 바랍니다.

2. 코인스택 SDK 설치

코인스택 자바 SDK를 설치할 때는 Maven 같은 패키지 매니저를 추천합니다. 패키지 매니저를 사용하지 않고 직접 jar 파일을 추가해서 사용하려면 Maven 저장소에서 최신 버전의 SDK를 다운로드 받을 수 있습니다.

Maven 환경에서는 `pom.xml` 파일에 아래 내용을 추가해 줍니다.

```
<dependency>
    <groupId>io.blocko</groupId>
    <artifactId>coinstack</artifactId>
    <version>3.0.23</version>
</dependency>
```

3. 클라이언트 생성과 종료

앞서 발급받은 API KEY를 가지고 CoinStackClient를 생성합니다. CredentialsProvider는 코인스택 API KEY와 접속할 서버를 가리키는 EndPoint를 관리합니다. Endpoint에는 SDK를 통해 접근하고자 하는 네트워크 주소를 입력하는데, 코인스택이 제공하는 클라우드 서비스를 통해 비트코인 네트워크에 접속하고자 할 경우 코인스택 메인넷(https://mainnet.cloudwallet.io)을 사용합니다. 만약 비트코인 테스트 네트워크에 접속하려면 테스트넷(https://regtestnet.cloudwallet.io)을 사용합니다. 설치형인 경우 Endpoint 인터페이스를 구현하여 해당 서버 접속 정보를 입력한 후 사용합니다.

```
//import io.blocko.coinstack.*
//import io.blocko.coinstack.model.*

CoinStackClient client = new CoinStackClient(new CredentialsProvider() {
  @Override
  public String getAccessKey() {
    return "YOUR_COINSTACK_ACCESS_KEY";
  }
  @Override
```

```
  public String getSecretKey() {
    return "YOUR_COINSTACK_SECRET_KEY";
  }
}, Endpoint.MAINNET);
```

CoinStackClient의 사용을 마친 후에는 명시적으로 종료를 선언하여 리소스를 반환합니다.

```
client.close()
```

4. 구동 테스트

설치한 코인스택 SDK가 정상적으로 동작하는지 확인하기 위해 블록체인의 상태 정보를 조회해 봅니다. 이를 위해서는 getBlockchainStatus 메소드를 이용합니다. 상태 정보 객체에 getBestHeight 메소드를 사용하면 최신 블록 번호를 알 수 있고, getBestBlockHash 메소드는 최신 블록의 해시를 반환합니다. 블록의 해시는 블록의 정보를 조회할 때 ID로 사용할 수 있습니다.

```
BlockchainStatus status = client.getBlockchainStatus();
System.out.println("bestHeight: "+ status.getBestHeight());
System.out.println("bestBlockHash: "+ status.getBestBlockHash());
```

Node.js

1. 코인스택 패키지 설치

npm (Node Packaged Modules) 패키지 매니저를 사용하여 CoinStack SDK를 설치할 수 있습니다.

```
npm install coinstack-sdk-js
```

2. 클라이언트 생성과 종료

앞서 발급받은 API KEY를 사용하여 CoinStack 객체를 생성합니다.

```
var CoinStack = require('coinstack-sdk-js')

var accessKey = "";
var secretKey = "";
var endpoint = "testchain.blocko.io";
var protocol = "https";
var client = new CoinStack(accessKey, secretKey, endpoint, protocol);
```

3. 구동 테스트

다음과 같이 블록체인 상태 정보를 조회해 봅니다.

```
client.getBlockchainStatus(function(err, status) {
    console.log(status.best_height);
    console.log(status.best_block_hash);
});
```

Meteor

1. 코인스택 패키지 설치

atmosphere 패키지 매니저를 이용하여 CoinStack SDK를 설치합니다.

```
meteor add shepelt:coinstack
```

2. 클라이언트 생성과 종료

앞서 발급받은 API KEY를 가지고 CoinStack 객체를 생성합니다.

```
var accessKey = "";
var secretKey = "";
var endpoint = "testchain.blocko.io";
var protocol = "https";
var client = new CoinStack(accessKey, secretKey, endpoint, protocol);
```

3. 구동 테스트

다음과 같이 블록체인의 상태 정보를 조회해 봅니다.

```
var status = client.getBlockchainStatusSync();
console.log(status.best_height);
console.log(status.best_block_hash);
```

4. Authentication

1. 개요

코인스택 API를 사용하기 위해서는 코인스택 서버에 자격 증명(credential)을 제공해야 합니다. 자격 증명은 사용자의 요청이 변조되지 않았음을 보장하는 역할과, 각 사용자의 요청을 구별하는 역할을 수행합니다. 자격 증명 절차는 다음과 같습니다.

1. 대시보드에서 API 키 발급 받기
2. 직접 REST 호출을 원하는 경우 토큰 발급
3. 자신의 프로그램이나 환경 변수에 설정을 원할 경우 API키 발급
4. 발급 받은 자격증명을 아래의 설명에 따라 설정하여 코인스택 API 호출

2. 자격 증명 방식

자격 증명은 아래 두 가지 방식으로 서버에 전달됩니다.

2.1. 토큰

REST API를 직접 사용하는 경우에는 토큰을 HTTP 요청의 헤더에 포함하여 보내야 합니다. 이 경우 토큰이 평문으로 전달되기 때문에, TLS 같은 안전한 프로토콜을 권장합니다.

2.2. 서명

SDK를 사용하여 API를 호출하는 경우 API 키와 비밀키로 request에 대한 전자 서명을 HMAC-SHA256 방식으로 생성하여 전달합니다.

3. API 키 발급 및 설정

3.1. API 키 발급

위에서 설명한 바와 같이, 코인스택 API를 사용하기 위해서는 먼저 자격 증명이 필요합니다. 대시보드에서 API 키 또는 API 토큰을 발급 받을 수 있습니다. Java와 Node, Meteor에서는 Access Key와 Secret Key를 사용하며, curl 등을 통해 직접 REST API를 호출하는 경우에는 Access Token을 사용합니다.

3.2. API 키 설정

3.2.1. 환경 변수 사용

API 키 또는 API 토큰을 환경 변수에 설정하면 코드 내에 이 값들을 직접 입력하는 일을 피할 수 있습니다. 발급받은 Access Key를 COINSTACK_ACCESS_KEY_ID라는 환경 변수에, Secret Key를 COINSTACK_SECRET_ACCESS_KEY에 각각 할당합니다. 환경 변수를 설정하는 자세한 방법은 환경 변수 설정 페이지를 참조 바랍니다.

환경 변수를 설정하면, 다음과 같이 기본 생성자를 이용해 CoinStackClient 객체를 생성할 수 있습니다.

Java

```
import io.blocko.coinstack.*

CoinStackClient client = new CoinStackClient();
```

Node.js

```
var CoinStack = require('coinstack-sdk-js')
var client = new CoinStack();
```

Meteor

```
export COINSTACK_ACCESS_KEY="YOUR_COINSTACK_ACCESS_KEY";
export COINSTACK_SECRET_KEY="YOUR_COINSTACK_SECRET_KEY";
var client = new CoinStack();
```

3.2.2. 코드 내에 직접 입력

REST API

curl을 사용하여 REST API를 통해 키를 직접 입력, 호출하는 방법은 다음과 같습니다.

```
curl https://mainnet.cloudwallet.io/ \
    -H "apiKey: YOUR_API_TOKEN_KEY"
```

Java

다음과 같이 객체 생성자에 직접 키를 입력할 수도 있습니다.

```
//import io.blocko.coinstack.*
//import io.blocko.coinstack.model.*

CoinStackClient client = new CoinStackClient(new CredentialsProvider() {
            @Override
            public String getAccessKey() {
                return "발급받은 access key";
            }

            @Override
            public String getSecretKey() {
                return "발급받은 secret key";
            }
        }, Endpoint.MAINNET);
```

Node.js

```
var accessKey = "YOUR_COINSTACK_ACCESS_KEY";
var secretKey = "YOUR_COINSTACK_SECRET_KEY";
var client = new CoinStack(accessKey, secretKey);
```

Meteor

```
var accessKey = "YOUR_COINSTACK_ACCESS_KEY";
var secretKey = "YOUR_COINSTACK_SECRET_KEY";
var client = new CoinStack(accessKey, secretKey);
```

5. Coinstack Basic

1. Blockchain과 Block

블록체인은 분산 데이터베이스 기술의 하나로, 운영자나 사용자라 할지라도 저장된 데이터를 임의로 조작하지 못하도록 고안되었습니다. 일반적인 데이터베이스와는 달리 블록체인은 자발적으로 참여하는 수 십, 수 백, 또는 수 천 개의 노드들에 의하여 운영되며, 이를 제어하기 위한 중앙 관리자가 존재하지 않습니다. 블록체인은 거래(transaction)들이 저장되는 거대한 분산 장부로서, 거래는 블록(block)에 담겨서 노드간에 공유되고 이 블록들이 사슬처럼 연결된 구조가 바로 블록체인입니다.

1.1. Blockchain 상태 조회

앞서 Getting Started 장에서 사용한 getBestBlockHash 메소드를 통해 블록 체인의 최신 상태 정보를 조회할 수 있습니다.

JAVA

```
String[] blockchainStatus = client.getBlockchainStatus();
```

REST API

```
YOUR_API_TOKEN_KEY="YOUR_API_TOKEN_KEY"

curl https://mainnet.cloudwallet.io/blockchain \
    -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 결과값은 JSON으로 반환됩니다.
>
> ```
> {
>   "best_block_hash": "000000000000000007e5724849ad23a76aa9db734e9671b2b08c075ab017fb6c",
>   "best_height": 370353
> }
> ```

Node.js

```
client.getBlockchainStatus(function(err, status) {
    console.log(status.best_block_hash, status.best_height);
});
```

Meteor

```
// server side
var status = client.getBlockchainStatusSync()
console.log(status.best_block_hash, best_height)

// client side
client.getBlockchainStatus(function(err, status) {
    console.log(status.best_block_hash, status.best_height);
});
```

1.2. Block 조회

특정 블록의 정보를 얻기 위해서는 블록의 ID인 블록 해시 값이 필요합니다. CoinStackClient 객체의 getBlock 메소드를 호출하면 입력한 블록 ID에 해당하는 블록의 정보가 담겨 있는 Block 객체를 반환합니다. Block 객체의 getBlockId, getParentId, getHeight 메소드를 통해 각각 블록 해시, 이전 블록의 해시, 블록 번호를 확인할 수 있습니다.

JAVA

```
Block block = client.getBlock("BLOCK_ID");
System.out.println("blockId: " + block.getBlockId());
System.out.println("parentId: " + block.getParentId());
System.out.println("height: " + block.getHeight());
System.out.println("time: " + block.getBlockConfirmationTime());
```

> 수행 결과는 다음과 같습니다.

```
blockId: 00000000000000000017363cc0f6562f4e4552b94032c49ce37f5fb1309062fc6
parentId: 00000000000000000428ec876f074bc7b34824bb11f8b00ec764622b3f54861d
height: 433254
time: Fri Oct 07 15:25:34 KST 2016
```

REST API

```
BLOCK_ID = "BLOCK_ID"

curl https://mainnet.cloudwallet.io/blocks/$BLOCK_ID \
    -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 수행 결과는 다음과 같습니다.

```
{
  "block_hash": "000000000000000035bce787da99a937602b7b47367c1b4b026149eb699df72",
  "height": 433254,
  "confirmation_time": "2016-10-07T06:25:34Z",
  "parent": "00000000000000000428ec876f074bc7b34824bb11f8b00ec764622b3f54861d",
  "transaction_list": [
    "d855dea8da20fad1b6f0d1654f4fea0d933809f315216e4ea60b984ee947faed",
    "b613a39216eab6af00f29ef584bd4dd057497d4b8410a0dc97651d1a18b44878",
    "...."
  ]
}
```

Node.js

```
client.getBlock("BLOCK_ID", function(err, result) {
    console.log('result', result);
});
```

Meteor

```
// server
var result = client.getBlock("BLOCK_ID") {
    console.log(result.block_hash, result.height);
});

// client
client.getBlock("BLOCK_ID", function(err, result) {
    console.log('result', result);
});
```

2. Addresses

블록체인 서비스나 비트코인을 사용하기 위해서는 주소를 생성해야 합니다. 주소는 비트코인 네트워크 상에서 자신의 신원을 나타냅니다. 아래 설명할 개인키와 달리, 비트코인 주소는 다른 사람들과 공유하여 자신의 주소로 비트코인을 받기도 하고 주소에 담겨있는 비트코인을 다른 주소로 보내기도 합니다. 블록체인 응용 서비스에서는 좀 더 다양하게 주소를 활용하기도 하는데, TSA(Time Stamp Authority)라 불리는 문서 증명의 경우 이 주소를 문서 hash값을 담는 주소로 활용합니다. 개인 인증으로 활용할 경우는 이 주소를 인증서의 보관 장소로 활용할 수 있습니다. 비트코인 주소는 ECDSA(Eliptic Curve Digital Signature Algorithm)를 기반으로 하고 있으며, 그 주소 자체가 암호화 성격을 가지고 있으므로 다양한 분야에 적용할 수 있습니다.

2.1. Address 생성

비트코인 주소를 생성하기 위해서는 ECDSA 알고리즘 기반의
공개키와 개인키를 먼저 생성해야 합니다. 생성 순서는 다음과 같습니다.

> Random Number Generation -> 개인키 생성 -> 공개키 생성 -> 비트코인 주소 생성

코인스택에서는 사용자가 위 과정을 이해하고 직접 수행할 필요 없이 손쉽게 주소를 생성할 수 있는 API를 제공합니다.

2.1.1. 개인키 생성

1에서 2^256 사이의 숫자 중 무작위로 정수 하나를 고르게 되는데 이것이 바로 개인키가 됩니다.
개인키는 보통 Base58Check으로 인코딩하여 사용하는데, 이것을 비트코인에서는 Private Key WIF
(wallet import format)라고 부릅니다. WIF 형으로 변화하면 그 길이가 짧아지고 혼동 하기 쉬운 문자열
(1 또는 i)이 제거되어 편리한 형태가 됩니다.

개인키는 유출되지 않도록 주의하여야 합니다. 개인키가 유출되면 내 지갑의 비트코인을 도난당할 수 있습니다.

다음은 코인스택이 제공하는 개인키 생성 메소드입니다. 이처럼 코인스택을 사용하면 임의의 새로운 개인키를
빠르고 간편하게 생성할 수 있습니다.

Java

```
// create a new private key
String newPrivateKeyWIF = ECKey.createNewPrivateKey();
System.out.println("private key: " + newPrivateKeyWIF);
```

> 결과값은 문자열로 반환됩니다.
>
> ```
> private key: L3nWhxhA68enYiXa2D1r4dj6bGDqXECEDoTgsgd4smBf8xrtnFED
> ```

2.1.2. 공개키 및 비트코인 주소 생성

공개키는 ECDSA 함수 중 하나인 secp256k1에 개인키를 입력값으로 한 결과 좌표 (x,y)로 표현됩니다.
우리가 사용하는 공개키의 형식은 이 좌표값 x와 y를 연결한 후 접두어를 붙인 형태입니다.

Java

```
// derive a public key
String newPublicKey = Hex.encodeHexString(ECKey.derivePubKey(newPrivateKeyWIF));
System.out.println("public key: " + newPublicKey);
```

> 결과값은 문자열로 반환됩니다.
>
> ```
> public key: 04ed0f040740a2d2e60a9910b3bd94c92217d387a6bec8e9c78e246b78160fa64e9a4f1a82ec07ed7a9070cb26e32c3f770d31783a8
> f8456113c2d76f95d1166c2
> ```

비트코인 주소는 이 공개키를 RIPEMD160(SHA256(K)) 함수로 다시 인코딩하여 생성합니다. 다음은 코인스택을 사용하여 개인키에서
비트코인 주소를 생성하는 예시입니다.

Java

```
// derive an address
String your_wallet_address = ECKey.deriveAddress(newPrivateKeyWIF);
System.out.println("address: " + your_wallet_address);
```

> 결과값은 문자열로 반환됩니다.
>
> ```
> address: 1MG4Y8JwqrMyNHpot9xvj62v12aPwuBB6W
> ```

2.4. Address Balance 조회

주소는 비트코인을 담고 있거나 OP_RETURN data를 담고 있습니다. 흩어져 있는 UTXO(Unspent output)의 비트코인 수량을 합산하여 보여주는 기능은 다음과 같습니다. (e.g. 0.0001 BTC = 10000 satoshi)

Java

```
// get a remaining balance
long balance = client.getBalance(your_wallet_address);
System.out.println("balance: " + balance);
```

> 결과값은 사토시 단위이며 Long 타입으로 반환됩니다.
>
> ```
> balance: 566368820
> ```

REST API

```
YOUR_WALLET_ADDRESS="YOUR_WALLET_ADDRESS"

curl https://mainnet.cloudwallet.io/addresses/$YOUR_WALLET_ADDRESS/balance \
    -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 결과값은 사토시 단위이며 JSON 형식으로 반환됩니다.
>
> ```
> {"balance":566368820}
> ```

Node.js

```
coinstackclient.getBalance("YOUR_WALLET_ADDRESS", function(err, balance) {
    console.log(balance);
});
```

Meteor

```
// server
var balance = coinstackclient.getBalanceSync("YOUR_WALLET_ADDRESS");
console.log(balance);

// client
coinstackclient.getBalance("YOUR_BLOCKCHAIN_ADDRESS", function(err, balance) {
    console.log(balance);
});
```

2.5. Address별 Transaction 조회

앞서 살펴본 바와 같이 주소는 비트코인을 담고 있거나 OP_RETURN 데이터를 담고 있습니다. 다음은 특정 주소에서 일어난 비트코인 거래 내역 또는 OP_RETURN 데이터의 내용을 조회하는 기능입니다.

Java

```
// print all transactions of a given wallet address
String[] transactionIds = client.getTransactions(your_wallet_address);
System.out.println("transactions");
for (String txId : transactionIds) {
  System.out.println("txIds[]: " + txId);
}
```

> 결과 값은 문자열의 배열로 반환됩니다.
>
> ```
> transactions
> 4fe42d98bdede1eedb504c48a7670b18b1d3691ae140d313e529ab92d53c7aa0
> ed7b57daba7621e726eab433823faa107cc20fdfe6c53c322288cb4161d850c1
> ```

REST API

```
curl https://mainnet.cloudwallet.io/addresses/YOUR_WALLET_ADDRESS/history \
 -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 결과 값은 JSON 형식으로 반환됩니다.
>
> ```
> ["4fe42d98bdede1eedb504c48a7670b18b1d3691ae140d313e529ab92d53c7aa0",
> "ed7b57daba7621e726eab433823faa107cc20fdfe6c53c322288cb4161d850c1",
> ...
> ]
> ```

Node.js

```
coinstackclient.getTransactions("YOUR_WALLET_ADDRESS", function(err, result) {
    console.log(result);
});
```

Meteor

```
// server
var result = coinstackclient.getTransactionsSync("YOUR_WALLET_ADDRESS");
console.log(result);


// client
coinstackclient.getTransactions("YOUR_WALLET_ADDRESS", function(err, result) {
    console.log(result);
});
```

2.6. Address별 unspent outputs 조회

블록체인은 주소 별로 잔고를 관리하고 있는 것이 아니라 단순히 거래 내역을 기록하고 있을 뿐입니다. 자신의 주소에 해당하는 Transaction output은 두 가지 종류로 나눌 수 있는데, 한 가지는 Spent ouput, 다른 한 가지는 일반적으로 UTXO라 불리는 Unspent output입니다. 새로운 거래를 만들기 위해서는 이 UTXO를 input으로 입력해야 하는데, 코인스택에서는 이를 위해 특정 주소와 연관된 UTXO들을 조회하는 기능을 제공합니다.

Java

```
//print all utxos
Output[] outputs = client.getUnspentOutputs(your_wallet_address);
System.out.println("unspent outputs");
for (Output utxo: outputs) {
    System.out.println(utxo.getValue());
}
```

> 결과값은 Output 객체의 배열로 반환됩니다. 예제에서는 Output이 가진 잔고를 출력합니다.
>
> ```
> unspent outputs
> 7259033307
> 299990000
> 961900000
> ```

REST API

```
curl https://mainnet.cloudwallet.io/addresses/YOUR_WALLET_ADDRESS/unspentoutputs \
 -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 결과값은 JSON 형식으로 반환됩니다.
>
> ```
> [
>  {
>  "transaction_hash": "f2020daaf62e03deb2c6f8a94988a6cca3d66273dc6fa2169e88f02b288520ea",
>  "index": 1,
>  "value": "7259033307",
>  "script": "76a914a13ca67f70cc4afefd4057f87d1e433dab05a20788ac",
>  "confirmations": 2509
> ```

```
  },
  {
  "transaction_hash": "9b30059da6517a08034c0faf3bd347a7a24189285fa80ede850ecc552d165620",
  "index": 1,
  "value": "299990000",
  "script": "76a914a13ca67f70cc4afefd4057f87d1e433dab05a20788ac",
  "confirmations": 99
  },
  {
  "transaction_hash": "9ac9ab9285ebb5107bfc2e086d35c3ebefd9a899722039fac667e6ece3b20c90",
  "index": 1,
  "value": "961900000",
  "script": "76a914a13ca67f70cc4afefd4057f87d1e433dab05a20788ac",
  "confirmations": 91
  }
 ]
```


Node.js

```
coinStackClient.getUnspentOutputs("YOUR_WALLET_ADDRESS", function(err, result) {
    console.log('result', result);
});
```

Meteor

```
// server
var result = coinstackclient.getUnspentOutputsSync("YOUR_WALLET_ADDRESS");
console.log('result', result);

// client
coinstackclient.getUnspentOutputs("YOUR_WALLET_ADDRESS", function(err, result) {
    console.log('result', result);
});
```

3. Transactions

3.1. 트랜잭션 생성

트랜잭션을 이용하여 특정 블록체인 주소로 비트코인을 전송하거나, 데이터를 저장할 수 있습니다.

3.1.1. Unspent Output 조회

새로운 트랜잭션을 만들기 위해서는 일단 input으로 사용할 unspent output을 조회하여 사용 가능한 잔고를 확인해야 합니다. SDK에 포함된 transaction builder를 이용하면 트랜잭션 생성시 자동으로 unspent output을 조회하여 최적의 unspent output을 선택해 줍니다.

3.1.2. 트랜잭션 생성 및 서명

3.1.1.1. 일반 트랜잭션

일반적으로 트랜잭션은 내 주소의 잔여 비트코인을 타 주소로 전송하기 위해 사용됩니다. 이 경우 비트코인을 전송받을 주소와 전송할 금액, 사용할 unspent output을 지정해야 합니다. (unspent output은 SDK가 자동적으로 지정해 줍니다.) 그리고 개인키로 서명하여 지정한 unspent output의 소유권을 증명합니다.

java

```
// create a target address to send
String toPrivateKeyWIF = ECKey.createNewPrivateKey();
String toAddress = ECKey.deriveAddress(toPrivateKeyWIF);

// create a transaction
long amount = io.blocko.coinstack.Math.convertToSatoshi("0.0002");
long fee = io.blocko.coinstack.Math.convertToSatoshi("0.0001");

TransactionBuilder builder = new TransactionBuilder();
builder.addOutput(toAddress, amount);
builder.setFee(fee);
```

```
// sign the transaction using the private key
String rawSignedTx = client.createSignedTransaction(builder, "YOUR_PRIVATE_KEY");
System.out.println(rawSignedTx);
```

> 결과값은 문자열로 반환됩니다.
>
> ```
> 01000000 01 be66e10da854e7aea9338c1f91cd489768d1d6d7189f586d7a3613f2a24d5396 00000000 8c 49 3046022100cf4d7571dd47a4d47
> f5cb767d54d6702530a3555726b27b6ac56117f5e7808fe0221008cbb42233bb04d7f28a715cf7c938e238afde90207e9d103dd9018e12cb7180e 0
> 1 41 042daa93315eebbe2cb9b5c3505df4c6fb6caca8b756786098567550d4820c09db988fe9997d049d687292f815ccd6e7fb5c1b1a9113799981
> 8d17c73d0f80aef9 ffffffff 01 23ce010000000000 19 76 a9 14 a2fd2e039a86dbcf0e1a664729e09e8007f89510 88 ac 00000000
> ```

Node.js

```
var txBuilder = coinstackclient.createTransactionBuilder();
txBuilder.addOutput("TO_ADDRESS", CoinStack.Math.toSatoshi("0.0001"));
txBuilder.setInput("YOUR_WALLET_ADDRESS");
txBuilder.setFee(CoinStack.Math.toSatoshi("0.0001"));

txBuilder.buildTransaction(function(err, tx) {
    tx.sign("YOUR_PRIVATE_KEY")
    var rawSignedTx = tx.serialize()
    console.log(rawSignedTx)
});
```

Meteor

```
// server
var txBuilder = coinstackclient.createTransactionBuilder();
txBuilder.addOutput("TO_ADDRESS", CoinStack.Math.toSatoshi("0.0001"));
txBuilder.setInput("YOUR_WALLET_ADDRESS");
txBuilder.setFee(CoinStack.Math.toSatoshi("0.0001"));

var tx = coinstackclient.buildTransactionSync(txBuilder);
tx.sign("YOUR_PRIVATE_KEY");
var rawSignedTx = tx.serialize();
console.log(rawSignedTx)

// client
var txBuilder = coinstackclient.createTransactionBuilder();
txBuilder.addOutput("TO_ADDRESS", CoinStack.Math.toSatoshi("0.0001"));
txBuilder.setInput("YOUR_WALLET_ADDRESS");
txBuilder.setFee(CoinStack.Math.toSatoshi("0.0001"));

txBuilder.buildTransaction(function(err, tx) {
    tx.sign("YOUR_PRIVATE_KEY");
    var rawSignedTx = tx.serialize();
    console.log(rawSignedTx)
});
```

비트코인 클라이언트에 따라 다르지만, 너무 적은 액수의 비트코인을 전송하는 트랜잭션은 dust threshold라는 정책에 의해 전송이 거부당할 수 있습니다.

Coinstack SDK의 transaction builder를 사용하는 경우 너무 적은 액수의 비트코인을 송금하려고 하면 SDK에서 오류가 발생합니다. 현재 코인스택 SDK에 기본으로 설정된 최소 전송 가능 액수는 5460 사토시이며, allowDustyOutput(true) 함수를 사용하면 이 설정을 비활성화하여 더 적은 양의 비트코인도 전송할 수 있습니다.

그러나 코인스택 SDK와는 별개로 각 블록체인 서버에 상이한 dust threshold 정책이 존재할 수 있으므로 접속하시는 서버의 정책 또한 확인하여 사용하시기 바랍니다. 현재 코인스택에서 제공하는 비트코인 클라우드 서비스의 dust threshold는 546 사토시이며, 구축형 모델에서는 이 값을 변경할 수 있습니다.

```
builder.allowDustyOutput(true);
```

3.1.1.2. Data 트랜잭션

비트코인을 전송하는 대신 OP_RETURN 이라는 output script를 사용하여 트랜잭션에 데이터를 저장할 수 있습니다.
코인스택 SDK에 포함된 transaction builder에서 제공하는 인터페이스를 사용하면 블록체인에 데이터를 손쉽게 저장할 수 있습니다.
코인스택에서 제공하는 stamping 및 Open Assets 관련 기능은 바로 이 OP_RETURN을 이용하여 구현되었습니다.

```
builder.setData("DATA_AT_OP_RETURN".getBytes());
```

3.2. 트랜잭션 전송

서명된 트랜잭션은 이제 네트워크에 전송될 수 있습니다. 만약 전송 과정에서 문제가 발생하거나, 트랜잭션 자체에 오류가 있으면 관련된 오류 코드로 원인을 확인할 수 있습니다. 하지만 트랜잭션 자체에 오류가 없는 경우에도 블록체인 네트워크의 특성상 일시적으로 트랜잭션이 거부되는 경우가 있습니다. 이 때는 네트워크에 성공적으로 전파될 때까지 재전송하면 됩니다.

Java

```
// send the signed transaction
client.sendTransaction(rawSignedTx);
```

Node.js

```
txBuilder.buildTransaction(function(err, tx) {
    tx.sign("YOUR_PRIVATE_KEY")
    var rawSignedTx = tx.serialize()
    // send transaction
    coinstackclient.sendTransaction(rawSignedTx, function(err) {
        if (null != err) {
            console.log("failed to send tx");
        }
    });
});
```

Meteor

```
// server
try {
    // send tx
    coinstackclient.sendTransactionSync(rawSignedTx);
} catch (e) {
    console.log("failed to send tx");
}

// client
txBuilder.buildTransaction(function(err, tx) {
    tx.sign("YOUR_PRIVATE_KEY");
    var rawSignedTx = tx.serialize();
    // send tx
    coinstackclient.sendTransaction(rawSignedTx, function(err) {
        if (null != err) {
            console.log("failed to send tx");
        }
    });
});
```

3.3. 트랜잭션 조회

생성한 트랜잭션의 ID를 조회하는 방법은 다음과 같습니다.

Java

```
String transactionId = TransactionUtil.getTransactionHash(rawSignedTx);
```

Node.js

```
var transactionId = CoinStack.Transaction.fromHex(rawSignedTx).getHash();
```

Meteor

```
var transactionId = CoinStack.Transaction.fromHex(rawSignedTx).getHash();
```

트랜잭션이 네트워크에 성공적으로 반영된 경우, 다음과 같이 해당 ID로 트랜잭션 정보를 조회할 수 있습니다.

Java

```
// print transaction
Transaction tx = client.getTransaction("YOUR_TRANSACTION_ID");
System.out.println(tx.getConfirmationTime())
```

> 결과 값은 Transaction 객체로 반환됩니다. 예제에서는 Transaction이 확정된 시간을 출력합니다.
>
> ```
> Mon Oct 10 18:13:34 KST 2016
> ```

REST API

```
curl https://mainnet.cloudwallet.io/transactions/YOUR_TRANSACTION_ID \
    -H "apiKey: $YOUR_API_TOKEN_KEY"
```

> 결과값은 JSON으로 반환됩니다.
>
> ```
> {
>  "transaction_hash": "8c14f0db3df150123e6f3dbbf30f8b955a8249b62ac1d1ff16284aefa3d06d87",
>  "block_hash": [
>  {
>  "block_hash": "000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506",
>  "block_height": 100000
>  }
>  ],
>  "coinbase": true,
>  "inputs": [
>  {
>  "transaction_hash": "0000000000000000000000000000000000000000000000000000000000000000",
>  "output_index": -1,
>  "address": [
>  ],
>  "value": ""
>  }
>  ],
>  "outputs": [
>  {
>  "index": 0,
>  "address": [
>  "1HWqMzw1jfpXb3xyuUZ4uWXY4tqL2cW47J"
>  ],
>  "value": "5000000000",
>  "script": "41041b0e8c2567c12536aa13357b79a073dc4444acb83c4ec7a0e2f99dd7457516c5817242da796924ca4e99947d087fedf9ce467cb
> 9f7c6287078f801df276fdf84ac",
>  "used": false
>  }
>  ],
>  "time": "2010-12-29T11:57:43Z",
>  "broadcast_time": "0001-01-01T00:00:00Z",
>  "addresses": [
>  "1HWqMzw1jfpXb3xyuUZ4uWXY4tqL2cW47J"
>  ]
> }
> ```

Node.js

```
coinstackclient.getTransaction("YOUR_TRANSACTION_ID", function(err, result) {
    console.log('result', result);
});
```

Meteor

```
// server
var result = coinstackclient.getTransactionSync("YOUR_TRANSACTION_ID");
console.log(result.transaction_hash, result.inputs);

// client
coinstackclient.getTransaction("YOUR_TRANSACTION_ID", function(err, result) {
    console.log('result', result);
});
```

6. Coinstack Stamping

블록체인은 그 불가역성(한번 기록되면 삭제되거나 변경되지 않음)을 활용하여 여러 분야에 적용할 수 있습니다. 특히 문서 진위 확인 서비스(Document Stamping)에서 이 블록체인 기술이 각광을 받고 있습니다. Coinstack은 앞서 Transactions에서 설명한 바와 같이 Data output을 활용한 트랜잭션을 생성할 수 있습니다. 본인이 직접 데이터 트랜잭션을 생성하여 문서의 지문값을 등록할 경우 다음과 같은 일련의 작업들을 수행해야 합니다.

1. 본인의 비트코인 주소 생성
2. 주소에 수수료로 쓰일 비트코인 전송
3. 문서의 Hash값 추출(ex. SHA256)
4. 문서의 Hash값을 추가한 data output 생성
5. data output을 추가한 트랜잭션 생성
6. 트랜잭션을 블록체인 네트워크에 Broadcasting
7. 본인 주소의 Balance를 확인하여 수수료가 충분한지 계속 모니터링

하지만 단순히 문서의 지문값(Hash value)만을 빠르고 간단히 블록체인에 등록하고자 할 때는 이 모든 작업이 부담이 될 수 있습니다. Coinstack은 트랜잭션 level에서 데이터를 등록하는 기능을 추상화하여 위 작업을 간편하게 수행할 수 있는 Document Stamping 기능을 제공합니다.

1. Stamp document

SHA-256 해시를 블록체인에 기록하도록 요청합니다. 차후 요청 결과를 확인할 수 있는 stamp ID를 반환합니다. 단순히 문자열을 Coinstack에 전달하는 것만으로 번거로운 하위 작업들을 대신 처리하고 문서의 지문값을 블록체인에 등록합니다.

```
String privateKeyWIF = "YOUR_PRIVATE_KEY";
String message = "Hello, world";
byte[] data = message.getBytes();
String hash = Codecs.SHA256.digestEncodeHex(data);

String stampId = coinStackClient.stampDocument(privateKeyWIF, hash);
System.out.println(stampId);
```

> 결과값은 String 으로 리턴됩니다.
>
> ```
> "7f902baec17633d12fb70892698157f595682910c96e3ee44cbdc3e2545d6665-2"
> ```

2. Get stamp status

Stamp 요청 상태를 확인합니다. Stamp가 저장된 transaction hash와 output, confirmation 상태를 반환합니다.

```
Stamp stamp = coinStackClient.getStamp("YOUR_STAMP_ID_VALUE");
```

> 결과값은 Stamp object 로 리턴됩니다.
>
> ```
> Stamp stamp = coinStackClient.getStamp("YOUR_STAMP_ID_VALUE");
>
> System.out.println(stamp.getHash());
>
> if("YOUR_STAMP_HASH".equals(stamp.getHash())){
>     System.out.println("Right");
> } else {
>     System.out.println("Wrong");
> }
> ```

7. Coinstack Multisig

1. 개요

비트코인 계정은 공개 키 암호 방식으로 관리되므로 안전하고 편리하지만, 개인키를 분실하는 경우 해당 주소에 가지고 있는 비트코인을 사용할 수 없게 된다는 위험성이 있습니다. 실제로 상당량의 비트코인이 개인키의 분실로 인해 특정 주소에서 더이상 사용되지 못하고 잠들어 있을 것으로 추정되고 있습니다. 또한 허술한 관리로 인해 개인키가 유출되는 경우 비트코인을 쉽게 도난당할 수 있습니다.

만약 비트코인의 특정 주소에 여러 개의 키를 설정할 수 있고, 그 중 몇 개의 키가 있어야 해당 주소의 비트코인을 사용할 수 있게 한다면 단 한 개뿐인 개인키의 분실이나 유출만으로 비트코인을 사용할 수 없게 되거나 도난당하는 문제를 완화할 수 있습니다.

예를 들어 2개의 키를 설정하고 그 중 하나만으로도 특정 주소의 비트코인을 사용할 수 있게 해 두면, 사용자는 2개의 키를 각각 다른 공간에 보관할 수 있습니다. 만약 한 개의 키를 잃어버리더라도 별도로 보관하고 있던 다른 키를 사용하여 해당 비트코인을 사용할 수 있게 됩니다.

또 다른 활용은 3개의 키를 설정하고 그 중 두 개가 있어야 비트코인을 사용할 수 있게 하는 방식입니다. 3개의 키 중 하나는 거래소나 지갑 같은 서비스 회사에서 관리하게 하면 사용자는 항상 두 개의 키를 준비해야 하는 불편 없이 서비스를 이용할 수 있습니다. 하지만 평소 사용하던 키를 분실하더라도 별도로 보관하던 키를 이용해 비트코인을 안전하게 옮길 수 있고, 서비스 회사 입장에서는 키가 하나밖에 없기 때문에 사용자를 속일 수 없으며, 해킹을 당하더라도 사용자의 지갑을 안전하게 보호할 수 있습니다.

이처럼 특정 비트코인 주소에 n개(n>1)의 개인 키를 설정해 두고, 해당 주소의 비트코인을 사용할 때는 그 중 m개(n>=m>=1)의 개인 키가 필요하도록 설정하는 방식을 Multisig라고 합니다.

2. Multisig Funding

1. P2SH 주소 생성
2. P2SH 주소에 Bitcoin 전송하는 Transaction 생성

Multisig 주소에 Bitcoin을 전송하기 위해서는 우선 Multisig 주소를 생성해야 합니다. 일반적으로 많이 쓰이는 Multisig 주소라는 용어는 사실 P2SH 주소입니다. P2SH 의 Pubkey script는 일반적으로 다음 형태를 가집니다.

```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
```

위 Pubkey script의 input으로 사용되는 redeemScript는 다음과 같은 구성 요소를 가집니다.

```
Pubkey script: <m> <A pubkey> [B pubkey] [C pubkey...] <n> OP_CHECKMULTISIG
```

m of n multisig의 경우에 n개의 pubkey와 이 트랜잭션을 사용하기 위해 필수적으로 제출해야 하는 signature 수 m을 input으로 가지는 것을 알 수 있습니다.

다음은 Coinstack에서 제공하는 Multisig 주소 관련 함수입니다.

```
//Redeemscript 생성
    public void testRedeemScript() throws UnsupportedEncodingException, DecoderException {
        String publickey1 = "04a93d29a957d3e0064f6fde7a6c296e1ab2643f877d98ca5f52bdb9df43d3f70dfc040ee212b9bae63b2cf266ca677d3
1d72db53d1e928851d131d1cb9d9bde25";
        String publickey2 = "04c30518fb1d5f0e96ba9f262f31260c8ce02e322494436e4ccb0981ba687a1a2626ab30911ddad22023bcbdac1329fc7
3e999ff2836c608e9c8e405f4e6c0b615";
        String publickey3 = "04486a7c8754177b488982dcb13bd37fa6005a439dce55101c35ba3c5f60928036920e72a59429f0425967b735a1c52c9
d2018a5ef4f63b8a8473ded4a29d5bad0";
        List<byte[]> pubkeys = new ArrayList<byte[]>(3);

        pubkeys.add(Hex.decodeHex(publickey1.toCharArray()));
        pubkeys.add(Hex.decodeHex(publickey2.toCharArray()));
        pubkeys.add(Hex.decodeHex(publickey3.toCharArray()));

        String redeemScript = MultiSig.createRedeemScript(2, pubkeys);
    }
```

public key 3개와 이 UTXO를 사용하기 위해 2개의 pubkey를 제출해야 하는 redeemscript를 생성하는 예시입니다.

```
//  P2SH 주소 생성
```

```
    public static String createAddressFromRedeemScript(String redeemScript, boolean isMainNet) throws MalformedInputException
{
        Script redeem = null;
        String from = null;
        try {
            redeem = new Script(Hex.decodeHex(redeemScript.toCharArray()));
        } catch (ScriptException e) {
            throw new MalformedInputException("Invalid redeem script", "Parsing redeem script failed");
        } catch (DecoderException e) {
            throw new MalformedInputException("Invalid redeem script", "Parsing redeem script failed");
        }
        from = createAddressFromRedeemScript(redeem, isMainNet);
        return from.toString();
    }
```

Redeemscript를 이용하여 P2SH 주소를 생성하는 함수입니다.

최종적으로 생성되는 P2SH 주소는 일반적으로 사용되는 P2PKH 주소와 동일하게 사용할 수 있습니다. 즉, P2SH 주소에 Bitcoin을 전송하면 Multisig Transaction을 생성한 것이고 이 주소를 계속적으로 Funding을 하거나 Spending을 하면서 사용할 수 있습니다. Funding은 단순히 이 주소에 Bitcoin을 보내기만 하면 되고 Spending은 m개(m of n Multisig)의 Private key로 서명한 signature script가 제출되어야만 가능합니다.

3. Multisig Spending

위 장에서 Funding한 Multisig transaction UTXO를 사용하기 위해서는 m개의 Private key가 필요합니다.
m개의 private key를 array list로 전달하고 수신 주소를 입력하면 Multisig spending transaction을 생성할 수 있습니다.

```
@Test
    public void testMultiSigTransaction() throws Exception {
        String privateKey1 = "5Jh46BhaJJeiW8C9tTz6ockGEgYnLYfrJmGnwYdcDpBbAvWvCbv";
        // String privateKey2 = "5KF55BbKeZZqmAmpQAovn7KoBRjVdW4UN9uPGZoK1y9RrkPhnhA";
        String privateKey3 = "5JSad8KB82c3XW69e1hz8g1YFFts4GTdjHuWHkh4d4A8MZWw12N";
        String redeemScript = "52410468806910b7a3589f40c09092d3a45c64f1ef950e23d4b5aa92ad4c3de7804ed95f0f50aca9ae928fb6e00223f
ad667693bf3e2b716dd6c9d474ad79f5b7a107e410494bae4aa9a4c2ca6103899098ca0867f62ca24af02fee2d6473a698d92fbc8c449aa2e236c0684ebb9e
0fbb23d847d4624fd8ca4a1fdc940df432c6e312e18e84104cbc882d221f567005ea61aa45d5414f25371472f6ab5973e13a39a9edc26359b6980aa4f6f34c
ea62e82bbe13adc7fde9fc26bba2be2e7c5f8011a68bea39bae53ae";

        List<String> prikeys = new ArrayList<String>();
        prikeys.add(privateKey1);
        prikeys.add(privateKey3);

        String to = "1F444Loh6KzUQ8u8mAsz5upBtQ356vN95s";
        long amount = Math.convertToSatoshi("0.0039");
        long fee = Math.convertToSatoshi("0.0001");

        TransactionBuilder builder = new TransactionBuilder();
        builder.addOutput(to, amount);
        builder.setFee(fee);

        String signedTx = coinStackClient.createMultiSigTransaction(builder, prikeys, redeemScript);

        assertNotNull(signedTx);

        coinStackClient.sendTransaction(signedTx);
    }
```

4. Multisig Partial Sign

위에 설명한 Multisig Spending은 실제 use case에는 그 쓰임이 한정적일 수 밖에 없습니다. 설명에서 보듯이 제출하는 Private key들을 모두 알고 있어야 하고 그것을 이용하여 한번에 sign을 해야만 Spending이 가능하기 때문입니다. 하지만, 실제 사용에서는 개인들 간에, 또는 기관과 개인, machine과 개인 등등이 각각 private key를 나눠 갖고 자신이 가진 private key를 자신만이 보관하고 유지해야 하는 경우가 대부분입니다. 따라서, 자신이 가지고 있는 private key로 sign을 추가하고 incomplete한 transaction을 또 다른 개인에게 전달할 수 있는 기능이 필요합니다. 이것이 Multisig partial sign입니다.

```
@Test
    public void testPartialSignTransaction() throws Exception {
        String privateKey3 = "5JiqywVBWDphZbR2UWnUtj3yTX52LmGhnBy8gGED7GDdxzPuRaZ";
        String privateKey2 = "5J4ZadEdMs3zaqTutP1eQoKnCGKSYrUgkPwBZrk3hNmFiz7B6Ke";
```

```
        //String privateKey1 = "5JKhaPecauUSKKZTJ2R8zhNZqFxLSDu3Q5dPU3ijSqkf2WGVekn";
        String redeemScript = "524104162a5b6239e12d3d52f2c880555934525dbb014dae7165380f77dcbf58b121b8033f59a1f7a4dcea589fc4405
ac756542dfa393d53f7a559038f59b8d1084de541046a8fca1041f6ecf55aaa4e431b6c4ee72b51492330e777f2967697eb633e277eabf5d6e2ab3132b218a
2d03b013ac90a80a4a2b5a27d1fa2a78cccad64d43b6f4104e850211b270fe7c97335411fcb774f6c7af0a8dd2e3360ba577e0c2979c51a375f5c256e2c870
1d1b9777c15b7fc8b42af435977fe338e4a4e19683c884ad0fd53ae";
        String to = "1F444Loh6KzUQ8u8mAsz5upBtQ356vN95s";
        long amount = Math.convertToSatoshi("0.0001");
        long fee = Math.convertToSatoshi("0.0001");

        TransactionBuilder builder = new TransactionBuilder();
        builder.addOutput(to, amount);
        builder.setFee(fee);

        String signedTx = coinStackClient.createMultiSigTransactionWithPartialSign(builder, privateKey3, redeemScript);

        signedTx = coinStackClient.signMultiSigTransaction(signedTx, privateKey2, redeemScript);

        assertNotNull(signedTx);

        coinStackClient.sendTransaction(signedTx);
    }
```

위 예제는 2 of 3 Multisig spending transaction을 partial sign을 통해 생성하는 예제입니다.
createMultiSigTransactionWithPartialSign 메소드는 bitcoin 수량, 수수료, 수신자 주소와 같은 Transaction 생성을 위한 기본적인 정보를 활용하여 incomplete transaction을 만들어 냅니다. 실제로 이 메소드를 활용한 결과값 transaction을 broadcast할 수 없습니다. 왜냐하면 최소 요구 signature 수량이 2인데 privateKey3을 활용하여 1개의 signature만을 추가한 transaction이기 때문입니다. 이 임시 Transaction을 완료하기 위해서는 1개의 signature를 추가해야 하는데 이때 사용하는 메소드가 signMultiSigTransaction입니다. 만약 signature 최소 요구 수량을 만족하면 완성된 transaction이 return되고 여전히 부족하면 1개의 signature가 추가된 incomplete transaction이 return됩니다.

5. Assets Multisig Sign

다음 장에서 설명할 Open Assets Protocol 또한 Multisig Spending 의 적용이 가능합니다. Assets 의 소유자와 Assets 를 관리하는 시스템이 서로 권한을 나눠 운영 할 수 있게 됩니다. 1 of 2 Multisig 를 사용하면 운영 시스템에서의 Assets 에 대한 관리가 가능하며 Assets spending transaction 에 대한 서명 주체 또한 명확하게 구분됩니다. 아래 예제와 같은 방법으로 Assets 에 Multisig 를 적용할 수 있습니다.

```
@Test
    public void testTransferAssetMultisig() throws Exception {
        String privateKey1 = "5JpxHsHWAFDJmGhQTypRf62s4UQYf59a2BnzDD2P7YxhXywraNF";
        String privateKey2 = "5JMH6KfwymcUXo1FgLJS2hC8NwicLWwTMVdHkiEgkwzvuymbast";
        String redeemScript = "5241047f78d43ff2db4c6b5e46bbc7ee6a9a253b1e66c2aece8422bd1412c6997acd8cf8561dc5ff3dbaa8a4edd94cd
270204db833e0db3d6ee53276736b30184fe57741049de5e58cb6f2a3e059958a69934618b6d02277926b2d3fcf5ad8b387ffc54119ff3bd8369c893455ff7
f06e1304252ccd19efb3198f1f8feb94710173123fcdd4104bc61f87c32d0262c81a0fe95979fc54133c1c97f1da7882e586728fa0e7743863841a0a09e02a
c1e13ac022e0d66ac84a42aae90c9ef96866e663234dceab49353ae";

        List<String> prikeys = new ArrayList<String>();

        prikeys.add(privateKey1);
        prikeys.add(privateKey2);

        String assetID = "AKdiAtZGMuEUnZXm9tA3TPN6YJHzUarxRn";
        long assetAmount = 114;
        String to = "akDQMunRtK15fVpBR83Jnv4ezQLtyerTJTB";
        long fee = Math.convertToSatoshi("0.0001");
        String rawTx = coloringEngine.transferMultisigAsset(prikeys, redeemScript, assetID, assetAmount, to, fee);
        assertNotNull(rawTx);
        System.out.println(rawTx);
        assertNotNull(TransactionUtil.getTransactionHash(rawTx));
        coinstackClient.sendTransaction(rawTx);
    }
```

8. Coinstack Open Assets

비트코인은 블록체인을 활용한 대표적인 어플리케이션입니다. 블록체인의 공공 장부의 성격을 활용하여 비트코인은 현재까지 가장 안전한 전자 화폐로서 자리매김하고 있습니다. 하지만 블록체인 상에서 비트코인만을 거래할 수 있는 것은 아닙니다. Open Assets 프로토콜은 블록체인 상에서 자신만의 화폐나 어떠한 종류의 자산이라도 발행하고 유통시킬 수 있도록 해줍니다. 예를 들어 지역 화폐, 주식, 회사의 쿠폰이나 바우쳐 등을 발행할 수 있습니다. Open Assets은 비트코인 프로토콜 위에 Layer를 두어 구현되어 있기 때문에 비트코인의 안전성을 그대로 상속 받습니다. 비트코인이 해결한 double spending(이중 지출)의 문제를 Open Assets 또한 가지고 있지 않기 때문에 자산이 이중으로 발행되거나 유통하는 과정에서 자산이 감소하거나 증가하는 문제 또한 없습니다.

1. 개요

Open Assets은 비트코인 프로토콜을 사용하여 고객이 맞춤형 자산을 발행하거나 유통시킬 수 있습니다. 일반적인 Open Assets 트랜잭션은 비트코인 트랜잭션과 그 모습은 같지만 비트코인 트랜잭션의 output 중 Open Assets만이 가지고 있는 특수한 output, 즉, Marker output을 가집니다.다시 말하면 비트코인 트랜잭션 중에 Marker output을 가진 트랜잭션이 Open Assets 트랜잭션입니다. Marker output은 사실 OP_RETURN으로 시작하는 일반적인 비트코인 data output입니다. 다만 Marker output은 자신만이 가진 고유 ID값으로 시작하게 됩니다(OAP Marker, 0x4f41). 따라서 Marker output은 다음과 같은 데이터로 시작하게 됩니다.

OP_RETURN PUSHDATA OAP_MARKER

여기에 버젼 정보(0x0100)와 Asset count(현재 트랜잭션에서 몇 개의 asset output이 있는지), Asset 수량 리스트, 그리고 메타데이터(Asset에 관한 부가 정보)가 붙게 됩니다.

필드	설명	크기
OAP Marker	이 트랜잭션이 Open Asset이라는 표시, 0x4f41	2 bytes
version number	Open Asset 프로토콜 버젼 번호, 0x0100	2 bytes
Asset quantity count	asset quantity list 필드의 아이템 갯수를 var-integer로 표현	1-9 bytes
Asset quantity list	모든 output에 대응하는 Asset 수량을 LEB128 인코딩하여 표현, 0보다 큰 정수의 리스트이며 순서는 output의 순서와 일대일로 대응(Marker output은 제외)	가변적
Metadata length	Metadata 필드의 길이를 var-integer로 인코딩	1-9 bytes
Metadata	현재의 트랜잭션과 관련한 Meta 데이터	가변적

2. Open Assets Address & Asset ID

자신의 Open Asset을 생성하기 위해서는 우선 개인키 생성이 필요합니다. 이 개인키에서 유추한 공개키를 인코딩하면 비트코인 주소가 되는데 Open Asset만의 주소 인코딩 규칙으로 인코딩을 하면 Open Asset 주소가 생성됩니다. 따라서 하나의 개인키에서 비트코인 주소와 Open Asset주소, 이렇게 두 개의 주소가 유추가 됩니다. 코인스택에서는 다음과 같은 주소 생성 기능을 제공합니다.

```
public static String deriveAssetAddressFromPrivateKey(String privateKey)
```

생성한 개인키를 넣으면 해당하는 Open Asset 주소를 리턴받을 수 있습니다. 또한 비트코인 주소를 넣으면 해당하는 Open Asset 주소를 얻을 수 있습니다.

```
public static String deriveAssetAddressFromBitcoinAddress(String bitcoinAddress)
```

반대로 Open Asset 주소를 넣었을 때 비트코인 주소를 리턴하는 메소드는 다음과 같습니다.

```
public static String deriveBitcoinAddressFromAssetAddress(String assetAddress)
```

Open Assets은 자신이 원하는 어떠한 형태의 자산도 발행할 수 있습니다. 발행한 Asset을 구별하기 위해서는 Asset ID가 필요한데, Asset ID는 일반적으로 발행 Transaction의 첫 번째 input의 script를 hash해서 만들어 내고 그 첫 글자는 'A'로 시작하게 됩니다.

`RIPEMD160(SHA256(첫번째 input script))`

코인스택에서 제공하는 Asset ID 생성 함수는 다음과 같습니다.

```
public static String createAssetID(byte[] inputScript)
```

3. Open Assets Issuance

다음은 코인스택에서 제공하는 Open Assets 발행 메소드입니다.

```
    @Test
    public void testIssueAsset() throws Exception {
        // 잔고가 있는 Private Key
        String privateKeyWIF = "KyJwJ3Na9fsgvoW2v4rVGRJ7Cnb2pG4yyQQvrGWvkpuovvMRE9Kb";
        // Private Key -> Address -> Asset Address 로 변환
        String toAssetAddress = Util.deriveAssetAddressFromAddress(ECKey.deriveAddress(privateKeyWIF));

        // 필요한 Asset 발행량 설정
        long assetAmount = 100000;
        long fee = Math.convertToSatoshi("0.0002");
        String rawTx = coloringEngine.issueAsset(privateKeyWIF, assetAmount, toAssetAddress, fee);

        assertNotNull(rawTx);

        // 발행 트랜잭션 전송
        coinStackClient.sendTransaction(rawTx);
    }
```

Open Assets 발행을 위해서 input으로 발행 주소가 유추된 개인키, Asset 수량, Open Asset 수령 주소, 비트코인 수수료를 입력합니다. 여기서 사용하는 개인키는 Open Assets의 발행 주소가 유추되었던 원본 개인키를 사용해야 하며 추가 발행을 할 때에는 오직 이 개인키만을 사용해야 합니다.

4. Open Assets Transfer

다음은 코인스택에서 제공하는 Open Asset 전송 메소드입니다.

```
    @Test
    public void testTransferAsset() throws Exception {
        String privateKeyWIF = "KztgqWTCKS6dxuUnKcJnyiHtUqJ78k91P8Rn9oNrFLhgnRh3wiiE";

        String assetID = "AKJFoih7ioqPXAHgnDzJvHE8x2FMcFerfv";
        long assetAmount = 30;
        String to = "akFNUeHPC59mrBw3E57bRjgKTdUZeMxeLur";
        long fee = Math.convertToSatoshi("0.0002");
        String rawTx = coloringEngine.transferAsset(privateKeyWIF, assetID, assetAmount, to, fee);

        assertNotNull(rawTx);

        coinStackClient.sendTransaction(rawTx);
    }
```

Open Asset 전송 메소드는 transferAsset 메소드입니다. Input 파라미터로 Asset 소유자의 개인키, Asset ID, 전송하고자 하는 Asset의 수량, 수신자의 Asset 주소, 비트코인 수수료를 넣습니다. 이때 사용하는 개인키는 Asset 소유자의 주소입니다. 따라서, 최초 발행한 Asset 발행인의 개인키일 필요는 없습니다. 앞서 말했듯이 발행인의 개인키는 Asset을 추가로 발행할 때 필요합니다. 특정 Asset 주소에 두 가지 종류 이상의 Asset이 수신되어 있을 수 있으므로 보내고자 하는 Asset ID를 반드시 명시해야 합니다.

9. Coinstack Smart Contract

1. 개요

스마트 컨트랙트라는 개념은 1990년대까지 거슬러 올라갑니다. 컴퓨터 과학자이자, 법학자이자 암호 전문가인 Nick Szabo는 당시 디지털 계약, 디지털 통화에 관한 연구 및 기고를 꾸준히 하고 있었는데, 그가 제시했던 Bit Gold 체계를 살펴 보면 상당 부분 비트코인과 유사하여 그의 탁월한 선견지명을 보여 줍니다. 이 때문에 사토시 나카모토의 정체가 그가 아닐까 하는 의심도 있었습니다.

Nick Szabo는 국제적 규모의 비지니스 환경에서 비용의 혁명이 지속적으로 발생했다고 이야기합니다. 처음에는 교통, 다음에는 제조, 그리고 최근에는 통신 비용이 극적으로 줄어들었음을 제시하며 다음 단계는 '관계' 비용의 차례라고 주장합니다. 지역별로 상이한 사법적 관할, 보안, 신용 문제를 해결하기 위해 대부분의 초국가적 기업들은 많은 비용을 들이고 있습니다. 이 비용을 재정의하면 '관계'를 설정하고, 유지하며, 그 영향력을 보장하는 데 드는 비용이라고 할 수 있습니다.
인터넷과 컴퓨터 네트워크가 지배하는 현재에도 우리는 이 관계를 관리하는 데 여전히 종이의 세상에 머물고 있습니다. 종이에 씌어진 계약서, 법 조항으로 우리의 관계를 설정하며, 이를 유지하기 위해 비싸면서도 비효율적인 법조계 시스템에 의존해야 하고, 약속의 이행을 위한 소송을 통해 공권력의 투입을 요청해야 할 때도 있습니다.

기업 환경에서의 이러한 관계를 표현하는 가장 전통적이고 형식적인 개념은 바로 '계약'입니다. Nick Szabo는 컴퓨터의 기능에 주목하여 이 계약을 알고리즘적으로 표현할 수 있으리라 보았습니다. 그렇게 된다면 서로 다른 언어의 미묘한 뉘앙스 차이 또는 조항 자체의 애매모호함이 야기하는 분쟁의 요소를 획기적으로 줄일 수 있고, 계약의 강제적 이행도 보장될 수 있을 것입니다. 하지만 이 아이디어가 실현 가능하려면 이를 기록하고 실행하고 공유할 신뢰할 수 있는 '분산 컴퓨팅 환경' 및 알고리즘으로 표현된 계약의 강제적 이행을 위해 실질적 가치를 내포하는 '디지털 자산'이 필요하다고 보았습니다.

비트코인은 블록체인을 통해 디지털 자산으로서의 가치를 증명해 보였으나, 제한된 표현력으로 인해 일반화된 분산 컴퓨팅에는 활용할 수 없었습니다. 이에 이더리움의 창시자 Vitalik Buterin은 이를 확장하여 디지털 자산이라는 속성에 추가로 스마트 컨트랙트라는 명확한 프로그램 코드로 표현되는 계약 수행 방안을 제안함으로써 일반화된 분산 어플리케이션 플랫폼으로써의 가능성을 보여주었습니다.

블로코에서도 자체적으로 비트코인 프로토콜을 확장하여 스마트 컨트랙트 플랫폼을 구축, 제공하고 있으며 이를 통해 다양한 비지니스 로직을 블록체인 상에서 구현할 수 있도록 지원하고 있습니다.

코인스택은 스마트 컨트랙트의 프로그래밍 언어로 스크립트 언어인 Lua를 사용하고 있습니다. 스마트 컨트랙트는 컨트랙트 별로 키-밸류 쌍의 상태 정보를 저장하고 있고 모든 노드는 블록체인을 통해 동일한 컨트랙트를 단계적으로 수행함으로써 항상 동일한 상태 정보를 유지 합니다. 스마트 컨트랙트는 정의, 수행, 조회의 세단계로 나뉩니다. 정의 단계에는 Lua언어로 스마트 컨트랙트 코드를 작성, 블록체인 상에 배포하는 단계이고, 수행 단계는 앞서 정의한 코드 내의 함수를 실제로 인자값을 넣어 호출하여 상태값을 변경하는 단계입니다. 마지막으로 조회 단계는 수행 결과가 반영된 현재 상태값을 읽어오는 단계입니다.

이 장에서는 각각의 단계에서 코인스택 SDK를 사용하는 구체적인 방법을 기술하고 분산화된 어플리케이션(Distributed Application, DApp)을 구축하는 방안을 제시합니다.

2. Programming Language

코인스택은 가벼운 스크립트 언어인 Lua를 스마트 컨트랙트 언어로 사용합니다. 블로코에서는 개발의 편의를 위해 코인스택 코드라는 Lua 개발 및 컨트랙트 배포 지원도구(IDE)를 제공합니다. 코인스택 코드에 대해서는 본 문서의 범위에서 벗어나므로 별도로 문의해 주시기 바랍니다.

다음은 블록체인 상태 저장소에 키-밸류 값을 저장하고 값을 읽어오는 Lua로 작성된 간단한 코인스택 스마트 컨트랙트의 예제입니다.

```
local system = require("system");

-- 상태 저장소에 키-밸류 저장
function set(key, value)
    system.setItem(key, value);
end

-- 상태 저장소에서 키에 해당하는 값 반환
function get(key)
    return system.getItem(key);
end

-- 현재 블록의 해시값을 출력. 출력 결과는 서버 로그에 찍힘
function printBlock()
```

```
    system.print(system.getBlockhash());
end
```

블록체인 함수 내부에서 사용자의 접근을 제어하거나 블록체인 상에 데이터를 읽고 쓰기 위해서는 블로코에서 제공하는 시스템 패키지를 사용해야 합니다. 위의 예제에서는 system 패키지의 setItem, getItem 함수를 통해서 코인스택 내부의 키-밸류 저장소에 접근하여 데이터를 영속적으로 저장하는 함수를 컨트랙트 사용자에게 제공합니다. 또한 간단한 디버그 메시지를 print 함수를 통해 노드의 로그파일에 출력 할 수 있도록 하였습니다.

시스템 패키지의 기능은 아래와 같이 한정적인데 이는 허용되지 않은 기능을 사용함으로써 시스템 상에 오류 혹은 노드간 데이터 불일치 (e.g. 파일 삭제, 랜덤 사용)를 사전에 방지하기 위함입니다. 같은 이유로 외부 패키지의 사용을 제한하고 있습니다.

시스템 패키지는 컨트랙트 코드 내에서 다음과 같이 require 명령어를 통해 사용 가능합니다.

```
local system = require("system");
```

시스템 패키지에서 제공하는 기본 함수 목록과 그에 대한 설명은 다음과 같습니다.

함수명	인자	반환값	기능
print	(string) message		노드상에서 디버그 메시지를 출력
setItem	(string) key, (any) value		값을 블록체인에 저장. Query문에는 사용 불가
getItem	(string) key	(any) value	저장된 값을 반환
getSender		(string) address	스마트 컨트랙트를 호출한 주소 반환
getCreator		(string) address	스마트 컨트랙트를 정의한 주소 반환
getBlockhash		(string) hash	호출 시점 스마트 컨트랙트가 포함된 블록의 해시 반환
getBlockheight		(number) height	호출 시점 스마트 컨트랙트가 포함된 블록의 높이 반환
getTimestamp		(number) timestamp	UNIX 타임스탬프를 반환
getContractID		(string) id	스마트 컨트랙트 ID를 반환
getTxhash		(string) hash	스마트 컨트랙트가 포함된 트랜잰션의 해시 반환
getNode		(string) id	코인스택 노드 식별자를 반환. 설정 파일에서 nodeid 값에 해당

3. Built-in Functions

Lua는 언어 자체적으로 유용한 함수 및 기본 패키지 제공합니다. 문자열 관리 함수 등 유용한 함수를 제공하므로 이것들을 활용하여 쉽게 스마트 컨트랙트를 작성 할 수 있습니다. 이에 대한 상세한 문법, 설명, 기본 내장 함수와 패키지는 Lua Reference Manual을 참조하시기 바랍니다.

Lua Smart Contract 는 코인스택에서 수행되기 때문에 Input/Output 을 포함한 OS 에 관련된 함수는 안정성 및 보안을 위해 제공되지 않습니다.

다음은 사용할 수 없는 기본 함수 목록입니다.

> print, dofile, loadfile, module

다음의 사용할 수 없는 기본 패키지 목록입니다.

> coroutine, io, os, debug

`string, math, table` 패키지는 사용이 가능합니다. 단, `math` 패키지의 `random, randomseed` 함수는 사용할 수 없습니다.

4. Smart Contract Definition

스마트 컨트랙트 코드를 사용하기 위해서는 코인스택 코드 IDE 혹은 직접 SDK를 사용하여 블록체인 상에 작성한 스마트 컨트랙트를 등록, 배포해야 합니다. 이러한 과정은 스마트 컨트랙트 정의 (smart contract definition)라고 불립니다.

다음은 코인스택 SDK를 사용하여 블록체인에 스마트 컨트랙트를 등록하는 예제입니다.

JAVA

```
CoinStackClient client = new CoinStackClient("CONNECTION_INFO");
String contractAddress = "ADDRESS_FOR_SMARTCONTRACT"; //컨트랙트로 사용할 주소
String privateKey = "MY_PRIVATEKEY"; //컨트랙트로 사용할 주소의 개인키
String definition = "SMARTCONTRACT_CODE"; //앞서 정의한 예제 코드로 대체

// SmartContract Builder Instance 생성
LuaContractBuilder builder = new LuaContractBuilder();
// 사용할 Contract 주소 설정. Transaction을 Sign할 개인키의 주소와 동일해야 함
builder.setContractId(contractAddress);
// 정의에 사용할 Lua Contract 소스코드 설정
builder.setDefinition(definition);
// 트랜잭션 수수료 설정, 트랜잭션의 크기에 따라 증가시켜야 함
builder.setFee(10000);
// 트랜잭션 빌드
String rawTx = builder.buildTransaction(client, ContractPk);
// 트랜잭션 해시값 추출
String txHash = TransactionUtil.getTransactionHash(rawTx);
// 트랜잭션 전송
client.sendTransaction(rawTx);

// 블록에 트랜잭션이 포함 될 때까지 대기
while(true) {
  Thread.sleep(1000);
  // 블록 포함 여부 검사. 블록에 포함이 되어야만 스마트컨트랙트 수행 가능
  if (client.getTransaction(txHash).getBlockHeights().length != 0)
    System.out.println("Smart Contract Definition is Committed: "+ txHash);
}
```

Node.js

```
var CoinStack = require('coinstack-sdk-js');
var client = new CoinStack("", "", "127.0.0.1:8080", "http");
var contractAddress = "ADDRESS_FOR_SMARTCONTRACT"; //컨트랙트로 사용할 주소
var privateKey = "MY_PRIVATEKEY"; //컨트랙트로 사용할 주소의 개인키
var definitionCode = "SMARTCONTRACT_DEFINITION_CODE"; //앞서 정의한 예제 코드로 대체

var builder = client.createLuaContractBuilder();
builder.setInput(contractAddress);
builder.setContractID(contractAddress);
builder.setFee(fee);
builder.setDefinition(definitionCode);

builder.buildTransaction(function (err, tx) {
  if (err) {
    console.log("Failed to create a transaction: ", err);
    return;
  }
  try {
    tx.sign(privatekey);
    var rawTx = tx.serialize();
    var txHash = tx.getHash();
    client.sendTransaction(rawTx, function (err) {
      if (err) {
        console.log("Failed to send transaction: ", err);
        return;
      }
    });
    client.getTransaction(txHash, function(err, result) {
      console.log(result);
    });
  } catch (e) {
    console.log(e);
  }
});
```

코인스택 스마트 컨트랙트는 기존의 비트코인 표준 프로토콜을 확장한 자체적인 프로토콜을 가지므로 이를 따르는 트랜잭션 생성을 지원하기 위해 LuaContractBuilder라는 클래스를 제공합니다. LuaContractBuilder는 스마트 컨트랙트의 정의 및 수행시 사용하며 이를 위해 Transaction을 만들어 보내야 하므로 큰 틀에서 TransactionBuilder와 형태가 유사합니다.

먼저 생성자를 통해 인스턴스를 생성하고 난 뒤에 setContractId(String contractID)라는 함수로 스마트 컨트랙트로 사용할 주소를 지정합니다. 이 주소는 고유한 스마트 컨트랙트의 ID로 사용되어, 이후 다른 사용자들도 이 주소를 대상으로 스마트 컨트랙트 트랜잭션을 수행하거나 상태값을 조회해 볼 수 있습니다. 따라서 기존에 다른 응용에서 사용하던 주소 혹은 마이닝 보상을 받는데 사용하던 주소를 사용하기 보다는 새로운 주소를 만들고 컨트랙트 정의 트랜잭션을 수행할 정도로 소량의 잔고를 충전하여 사용하시기를 권고합니다.
타 사용자가 컨트랙트를 재정의 하는 것을 방지하기 위해 특정 주소로의 스마트 컨트랙트 정의(define)는 해당 주소의 개인키 소유자만 가능합니다.

컨트랙트 정의를 위한 트랜잭션을 생성할 때는 setDefinition(String contractCode)에 스마트 컨트랙트 코드를 문자열로 할당합니다. 그리고 최종적으로 트랜잭션을 빌드하고 보내게 되면, 해당 코드는 트랜잭션의 op_return nullData 영역에 스마트 컨트랙트 프로토콜에 맞추어 담겨 블록체인을 통해 모든 노드에 전파되게 됩니다.

만약에 코드의 크기가 커서 Serialize한 트랜잭션의 크기가 1kb를 넘는다면 수수료를 기본 트랜잭션 수수료인 10000 사토시에 더해 트랜잭션 사이즈에 따라 (기본값은 10000 사토시/1kb)더 넣어야 합니다.

스마트 컨트랙트 정의 트랜잭션은 트랜잭션의 수행순서를 보장하기 위해 맴풀이 아닌 블록에 담겨져 있는 상태여야만 작동합니다. 트랜잭션이 실제 블록에 포함되어있는지의 여부는 위의 예제와 같이 getTransaction(String transactionHash)로 트랜잭션을 읽어 온 후, int[] getBlockHeights() 함수를 호출하여 최소 한개 이상의 값이 존재하는지 확인하면 됩니다. getBlockHeights()의 결과값이 배열인 이유는 트랜잭션이 브랜치가 생겨 여러 블록에 포함되는 경우, 여러 높이가 존재할 수 있기 때문입니다.

5. Smart Contract Execution

정의 후 블록에 포함된 스마트 컨트랙트는 수행 코드를 통해서 호출하고 그 결과를 블록체인에 반영할 수 있습니다. 한번 블록체인에 반영된 수행 결과는 모든 노드가 동일하며 지속적으로 유지됩니다. 스마트 컨트랙트를 수행하기 위해서는 정의한 컨트랙트 내의 함수를 호출하는 수행 코드를 작성해야 합니다. 다음은 앞서 정의한 코드를 수행하여 사용자가 설정한 키에 특정 값을 저장하는 예제 코드입니다.

```
call("set", "Key1", 365);
```

스마트 컨트랙트 수행 코드를 작성한 뒤에는 코인스택 코드를 통해서 실행하거나, SDK를 통해서 직접 수행 트랜잭션을 생성, 전송하여 실행할 수 있습니다. 다음은 SDK를 이용해 스마트 컨트랙트를 수행하는 예제 코드입니다.

JAVA

```
CoinStackClient client = new CoinStackClient("CONNECTION_INFO");
String contractAddress = "ADDRESS_FOR_SMARTCONTRACT"; //수행할 대상 스마트 컨트랙트 주소
String privateKey = "MY_PRIVATEKEY"; //나의 개인키
String executionCode = "SMARTCONTRACT_EXECUTION_CODE"; //앞서 정의한 예제 코드로 대체

LuaContractBuilder builder = new LuaContractBuilder();
 // 이전에 정의한 Smart Contract Address 할당
builder.setContractId(contractAddress);
// 수행할 함수 및 인자의 코드를 빌더에 설정
builder.setExecution(executionCode);
builder.setFee(10000);
String rawTx = builder.buildTransaction(client, privateKey);
client.sendTransaction(rawTx);
```

Node.js

```
var CoinStack = require('coinstack-sdk-js');
var client = new CoinStack("", "", "127.0.0.1:8080", "http");
var contractAddress = "SMARTCONTRACT_ADDRESS";
var executionCode = "SMARTCONTRACT_EXECUTION_CODE";

var builder = client.createLuaContractBuilder();
builder.setInput(contractAddress);
builder.setContractID(contractAddress);
builder.setFee(10000);
builder.setExecution(executionCode);
builder.buildTransaction(function (err, tx) {
```

```
  if (err) {
    console.log("Failed to create a transaction -", err);
    return;
  }

  try {
    tx.sign(privatekey);
    var rawTx = tx.serialize();
    var hash = tx.getHash();
    client.sendTransaction(rawTx, function (err) {
      if (err) {
        console.log("Failed to send transaction:", hash, "(", err, ")")
        return;
      }
      console.log("Sent transaction: ", hash);
    });
  } catch (e) {
    console.log(e);
  }
});
```

앞서 스마트 컨트랙트 정의와 유사하게 생성자를 통해 LuaContractBuilder 인스턴스를 생성하고 난 뒤에 setContractId(String contractID)라는 함수로 대상 스마트 컨트랙트 주소를 지정합니다. 여기에서는 앞서 스마트 컨트랙트를 정의하는데 사용한 주소를 동일하게 입력하여 앞서 정의한 컨트랙트를 수행하도록 합니다.

컨트랙트 수행을 위한 트랜잭션을 생성할 때는 setExecution(String contractCode) 함수에 수행할 스마트 컨트랙트 코드를 문자열로 할당합니다. (해당 코드는 트랜잭션의 op_return nullData 영역에 스마트 컨트랙트 프로토콜에 맞추어 적재됩니다.) 그리고 스마트 컨트랙트 주소에 대응하는 개인키가 아닌 스마트 컨트랙트를 수행할 각 사용자 개개인의 개인키로 트랜잭션을 빌드합니다. 최종적으로 트랜잭션을 전송하면 블록체인을 통해 모든 노드에 수행 코드가 전파됩니다. 그리고 사전에 정의된 스마트 컨트랙트에 따라 코드가 수행되어 블록체인 상태를 변화시키게 됩니다. 참고로 스마트 컨트랙트 별로 키-밸류 상태값은 구분되어 집니다. 즉 A 스마트 컨트랙트의 Key1을 변경해도 B 스마트 컨트랙트의 Key1에는 영향이 없습니다.

위의 예제에서는 1번째 인자에 'Key1'을 넣고 2번째 인자에 '365'를 넣어 기 정의한 set 함수를 호출하는 코드를 수행하였습니다. 그러면 set함수 내에서 system.setItem이 불리우고 contractAddress에 해당하는 상태 저장소의 Key1에 365라는 값이 저장되게 됩니다.

6. Smart Contract Query

스마트 컨트랙트의 수행은 트랜잭션 형태로 블록체인에 전달되어 블록에 포함된 후 수행 결과가 임의의 시간 후에 반영되므로, 클라이언트에 결과를 바로 반환할 수 없습니다. 따라서 상태값을 확인하기 위해서는 먼저 수행 트랜잭션이 블록에 포함되어 있는지의 여부를 확인하고, 이후 조회 기능을 사용하여 상태값을 읽어와야 합니다. 같은 이유로 컨트랙트 수행중에 에러가 나도 바로 확인이 불가능합니다. 따라서 코인스택 코드등을 통해 먼저 충분한 테스트를 거쳐 코드를 배포하기를 권하며, 필요시 print 기능을 사용해 서버에 디버그 메시지 남겨 버그를 확인하시기를 권합니다.

다음은 스마트 컨트랙트의 조회 관련 함수를 호출하여 상태값을 조회하는 Lua 코드의 예입니다.

```
res, ok = call("get", "Key1");
assert(ok);

return res;
```

내부적으로 getItem 함수를 감싸고 있는 get 함수를 조회할 키값(Key1)을 인자로 호출해 해당 키에 저장된 값을 읽어옵니다.
위와 같은 코드를 작성한 뒤에는 코인스택 코드를 통해서 조회 해 보거나, SDK를 사용해 조회할 수 있습니다.

다음은 SDK를 활용한 상태값 조회 예제입니다.

JAVA

```
String lookupCode = "SMARTCONTRACT_LOOKUP_CODE";
ContractResult res = client.queryContract(ContractAddress, ContractBody.TYPE_LSC, lookupCode);
if (res == null) {
  System.out.println("Result is null"); }
else {
  System.out.println(res.asJson());
}
```

Node.js

```
var client = new CoinStack("", "", "127.0.0.1:8080", "http");

client.queryContract(contract, "LSC", query, function (err, res) {
  console.log(err);
  console.log(res);
});
```

이전의 컨트랙트 정의, 수행 시와 다른점은 조회는 트랜잭션을 생성해서 쓰기 요청을 보내지 않고 client를 통해 읽기 요청을 보내는 것이라는 점 입니다. 조회 시에는 내부적으로 함수에 setItem 함수를 호출해도 블록체인 저장소에 반영되지 않습니다.

> Json으로 변환한 결과값은 다음과 같이 출력됩니다.
>
> ```
> {"result":"365", "success":true}
> ```

결과는 json 포맷으로 반환되며, result를 dApp의 용도에 맞게 파싱하여 사용하면 됩니다.

7. 확장 기능

기본 system 패키지 외에 개발의 편의성을 위해 추가적으로 다음과 같은 기능들을 제공합니다.

7.1. Json

입출력 시의 사용자 편의성을 위해 Json 패키지를 제공합니다. 이 패키지로 Json 형식의 문자열과 Lua Table 구조체 간의 자동 변환이 가능합니다.

함수명	인자	반환값	기능
encode	(any)	(string) JSON 형식 문자열	인자로 주어진 루아값을 JSON 형식의 문자열로 변환한다.
decode	(string) JSON 형식 문자열	(any)	JSON 형식의 문자열을 대응되는 Lua 구조로 변환한다.

다음은 실제로 문자열을 `decode` 한 예제입니다. (Lua 테이블로 표현되는 일반 Array 는 1 부터 시작되니 사용시에 주의가 필요합니다.)

```
local json = require("json")

arrayVar = json.decode("[V1, V2, V3]")
system.print(arrayVar[1])  -- 1번째 배열 값 출력
system.print(arrayVar[2])  -- 2번째 배열 값 출력
system.print(arrayVar[3])  -- 3번째 배열 값 출력
```

> 수행결과는 다음과 같습니다.
>
> ```
> V1
> V2
> V3
> ```

다음은 Lua 테이블을 문자열로 `encode` 한 예제입니다.

```
local json = require("json")

jsonVar = {V1, V2, V3; name="john"}
jsonStr = json.encode(jsonVar)
system.print(jsonStr)
jsonVar = {{V1, V2, V3}, name="john"}
jsonStr = json.encode(jsonVar)
system.print(jsonStr)
```

> 수행결과는 다음과 같습니다.

```
{ "1":V1, "2":V2, "3":V3, "name":"john"}
{ "1":[V1, V2, V3], "name":"john"}
```

위와 같이 encode 시에는 필드 명이 명시 된 경우 각 값에 필드 명을, 필드 명이 없는 경우는 자동으로 필드 명을 추가하여 문자열로 변환합니다. 만약에 값을 배열로 저장하고 싶다면, 두번째 결과와 같이 필드 명을 붙이지 않고 "{}"로 배열로 저장할 값을 묶어서 변환하면 됩니다.

7.2. Execution Permission

스마트 컨트랙트의 정의를 위해서는 배포할 주소의 개인키가 필요하지만, 수행은 기본적으로 모든 주소에서 해당 스마트 컨트랙트를 대상으로 가능합니다. 코인스택 스마트 컨트랙트에서는 이를 제한하여 권한이 있는 사용자만 함수를 수행할 수 있도록 하는 기능을 제공합니다. 이를 위해서는 `grant` 함수로 수행 권한을 특정 주소에 부여하고, 각 함수 내에서 명시적으로 `system.hasPermission` 함수로 권한이 있는지를 확인하여 권한이 있는 사용자만 해당 함수를 수행하도록 할 수 있습니다.

다음은 수행 권한 설정을 위해 제공되는 `system` 패키지 함수 설명입니다.

함수명	인자	반환값	기능
grant	(string)address, (string)tokenName		입력한 주소에 tokenName 대한 권한을 부여
revoke	(string)address, (string)tokenName		입력한 주소에 tokenName 대한 권한을 제거
hasPermission	(string)tokenName	(bool)	스마트 컨트랙트를 호출한 주소에 tokenName 대한 권한이 여부 반환

수행 권한을 설정하려면 다음과 같은 방법을 따릅니다.

1. 수행 권한을 설정 하고자 하는 스마트 컨트랙트 함수에 `hasPermission` 를 이용한 권한 여부 체크 로직을 추가합니다.
2. 스마트 컨트랙트 배포자는 호출 권한을 특정 사용자에게 `grant` 하는 스마트 컨트랙트를 수행 합니다.

다음은 `example()` 함수에 권한 체크 로직을 추가한 예입니다.

```
local system = require("system")
function example()
  -- example이라는 토큰을 가진 sender만 이 블록안의 코드를 수행 가능
  if system.hasPermission("example") {
  -- 실제 수행 로직 작성
  }
end
```

권한을 부여하기 위해서는 다음과 같은 내장 `grant` 함수를 호출하는 Lua 스마트 컨트랙트 트랜잭션을 수행하면 됩니다.

```
  -- CallerAddress에 example 이라는 토큰을 할당 함
  grant("CallerAddress", "example")
```

반대로 기 부여한 권한을 삭제하려면 `revoke` 함수를 호출하는 Lua 스마트 컨트랙트 트랜잭션을 수행하면 됩니다.

```
  -- CallerAddress에 기 부여되어 있던 example 토큰 권한을 제거
  revoke("CallerAddress", "example")
```

SDK에서는 스마트 컨트랙트 권한 부여, 제거 트랜잭션의 손쉬운 생성을 위해 유틸리티 함수를 제공합니다. 아래는 SDK를 이용한 권한 부여의 예입니다.

```
CoinStackClient client = new CoinStackClient("CONNECTION_INFO");
String contractAddress = "ADDRESS_FOR_SMARTCONTRACT"; // 수행할 대상 스마트 컨트랙트 주소
String privateKey = "SMARTCONTRACT_PRIVATEKEY"; //대상 스마트 컨트랙트 개인키
String granteeAddress = "ADDRESS_TO_GIVE_PERMISSION"; // 권한을 부여할 주소

LuaContractBuilder builder = new LuaContractBuilder();
 // 이전에 정의한 Smart Contract Address 할당
builder.setContractId(contractAddress);
// 빌더에 권한을 부여할 주소, 권한 또는 함수명, 부여 여부를 설정
// 부여 여부는 true 부여, false면 기존의 권한 제거
```

```
builder.setPermission(granteeAddress, "example", true);

String rawTx = builder.buildTransaction(client, privateKey);
client.sendTransaction(rawTx);
```

7.3. Event Notification

코인스택은 스마트 컨트랙트 수행중에 외부로 데이터를 전송하는 기능(Event Notification)을 제공합니다. 데이터는 전송 상황에 대한 식별자(이벤트 식별자)와 부가적인 정보로 구성됩니다. 데이터는 JSON 이며 형식은 아래와 같습니다.

```
{"UDEvent":eventname,"data":사용자지정값}
```

- eventname 은 전송 상황에 대한 식별자 입니다. `pushEvent` 의 `eventname` 인자값과 동일합니다.
- 사용자지정값 은 임의의 Lua 값입니다. `pushEvent` 의 `data` 인자값의 JSON 표현입니다.

데이터를 전송 받고자하는 외부 객체를 endpoint 로 부릅니다. 코인스택은 endpoint 로 HTTP PUSH request 를 보냅니다. 따라서, Endpoint 는 HTTP PUSH Method 를 구현하고 JSON 데이터를 처리하는 HTTP 서버의 URL 이어야 합니다.

다음은 Event Notification 을 위해 제공되는 `system` 패키지 함수 설명입니다.

함수명	인자	반환값	기능
registEvent	(string) eventname (string) endpoint (string) nodeid		eventname 으로 이벤트를 등록합니다. endpoint 는 HTTP PUSH Method 를 처리하는 서버의 url 입니다. 동일 eventname 으로 여러 개의 endpoint 를 등록할 수 있습니다. 등록시 동일한 endpoint 가 존재하면 업데이트 됩니다. nodeid 를 사용해서 특정 노드에만 등록할 수 있습니다. 명시하지 않으면 모든 노드에 등록됩니다.
unregistEvent	(string) eventname (string) endpoint		eventname, endpoint 로 등록된 이벤트를 제거합니다.
pushEvent	(string) eventname (any) data		eventname 에 해당하는 endpoint로 이벤트를 발생시킵니다. data는 임의의 사용자 지정 값입니다.
delEvent	(string) eventname		eventname 으로 등록된 전체 이벤트를 제거합니다.

아래는 Event Notification API 를 이용하여 입출금 상황을 통보하는 예제입니다.

모든 노드에서 이벤트 통보를 보내면 혼란스럽기 때문에 발생 노드를 지정합니다. 설정 파일의 nodeid 를 아래와 같이 수정합니다.

```
nodeid=alpha
```

스마트 컨트랙트 구현을 등록합니다.

event_def.lua

```
local system = require("system")

function addEventListener(eventName, endPoint)
    system.registEvent(eventName, endPoint, "alpha")
end

function deposit(name, amount)
    changeBalance(name, amount)
    system.pushEvent("balance", {op = "deposit", name = name, amount = amount})
end

function withdraw(name, amount)
    changeBalance(name, -amount)
    system.pushEvent("balance", {op = "withdraw", name = name, amount = amount})
end
```

```
function balance(name)
    accounts = system.getItem("accounts")
    if accounts == nil then
        accounts = {}
        accounts[name] = 0
    end
    return accounts[name]
end

function changeBalance(name, amount)
    accounts = system.getItem("accounts")
    if accounts == nil then
        accounts = {}
        accounts[name] = 0
    end
    balance = accounts[name] + amount
    accounts[name] = balance
    system.setItem("accounts", accounts)
end
```

이벤트 등록을 위한 스마트 컨트랙트를 호출합니다. (예제의 HTTP Server 는 JSON 데이터를 단순 출력하도록 구현했으며 상세 내용은 생략합니다.)

```
call("addEventListener", "balance", "http://postserver:3001")
```

`addEventListener` 함수는 `system.registEvent` 함수를 이용해서 설정 파일에 명시한 "alpha" 노드에 "balance" 이벤트를 등록합니다.

입출금 스마트 컨트랙트를 수행합니다.

```
call("deposit", "ktlee", 100)
call("deposit", "ktlee", 200)
call("withdraw", "ktlee", 50)
```

`deposit/withdraw` 함수는 입출금 처리 후에 `pushEvent` 함수를 통해 입출금 내역을 통보하게 됩니다.

http://postserver:3001 는 아래와 같이 출력합니다.

```
{"UDEvent":"balance","data":{"amount":100,"name":"ktlee","op":"deposit"}}
{"UDEvent":"balance","data":{"amount":200,"name":"ktlee","op":"deposit"}}
{"UDEvent":"balance","data":{"amount":50,"name":"ktlee","op":"withdraw"}}
```

10. Coinstack Permission

비트코인은 공개된 환경을 지향하여 기본적으로 누구나 트랜잭션을 전송 할 수 있고 풀노드를 운영하거나 마이닝을 시도할 수 있습니다. 하지만, 기업의 프라이빗 네트워크 등 비공개 환경에서 사용할 때에는 제한된 권한을 특정 사용자에게 부여해야 할 필요가 있습니다. 이러한 요구에 따라 코인스택에서는 특정 비트코인 주소에 역할을 부여하여 권한을 제한하는 방법을 제공합니다.

1. 사전 요구 사항

아래와 같은 설정을 사용하는 프라이빗 코인스택의 경우에만 권한 설정 기능을 사용할 수 있습니다. 코인스택 서버의 상세 설정 정보는 코인스택 설치 문서를 참조하시기 바랍니다.

- privnet=1: 프라이빗 네트워크를 구성합니다. 프라이빗 네트워크인 경우에만 역할 관리 기능을 사용 가능합니다.
- privnetgenesis={serialized genesis block}: 참여하려는 네트워크의 직렬화된 제네시스 블록(블록 높이가 0인 제일 처음 블록)을 기재합니다. 별도로 제공하는 gengenesis라는 프로그램에 최초의 admin으로 사용할 주소의 개인키를 사용하여 제네시스 블록을 생성할 수 있습니다.
- privnetnodekey={privatekeyWIF}: 노드에 부여할 개인키로 노드에 고유한 ID를 부여합니다. 각 노드마다 다르게 설정해 주어야 합니다.

2. 개요

프라이빗 코인스택에서는 하나의 비트코인 주소에 총 4가지의 역할을 부여 할 수 있습니다. 주소를 사용하는 주체가 사용자나 어플리케이션 일 경우 `Admin`과 `Writer` 역할을 부여하여 각각 계정관리와 트랜잭션의 쓰기 권한을 줄 수 있습니다. 역할별 상세 설명은 다음과 같습니다.

- Admin (Authority Manager): 역할을 설정/관리합니다. Admin을 제외한 다른 모든 역할(Writer, Miner, Node)을 활성화/비활성화 할 수 있습니다. 다른 모든 역할의 권한도 가집니다. 즉, 트랜잭션을 쓰고 블록을 생성하고 프라이빗 네트워크에 참여할 수 있습니다.
- Writer (Transaction Writer): 트랜잭션 생성 및 전송할 수 있습니다. 이 역할이 활성화 되어 있는 경우에 트랜잭션 쓰기 역할이 없는 주소의 개인키로 서명하여 전송한 트랜잭션은 코인스택 노드에서 수용하지 않습니다.

주소를 사용하는 주체가 노드인 경우 `Miner` 와 `Node` 역할을 부여하여 노드의 권한을 제한할 수 있습니다. (여기서의 주소는 노드의 설정파일의 privnetnodekey에 기재되어 있는 개인키와 대응하는 주소를 지칭합니다.)

- Miner (Block Miner): 블록을 생성할 수 있습니다. 이 역할이 활성화 되어 있는 경우에 Miner 역할이 없는 노드가 마이닝한 블록은 다른 코인스택 노드들이 수용하지 않습니다.
- Node (Network Participant): 이 역할이 있는 노드만 프라이빗 네트워크에 참여할 수 있습니다. 이 역할이 활성화 되어 있는 경우에 Node 역할이 없는 노드는 코인스택 노드에 접속 및 동기화를 할 수 없습니다.

역할을 부여하는 보안상 가장 안전한 방법은 역할별로 주소를 여러개 생성하여 하나의 주소에 하나의 역할만 부여하는 것이지만, 경우에 따라 주소 하나에 여러개의 역할을 부여해야 하는 경우도 있습니다. 예를 들면 Node와 Miner 권한이 활성되어 있는 경우, 주소 하나에 Node 와 Miner 역할을 동시에 부여해서 사용해야 해당 노드는 프라이빗 네트워크에 접속해여 자신이 생성한 블록을 전파할 수 있습니다. 또한 Admin 역할을 가지는 주소는 환경 설정의 편의성을 위해 모든 권한을 가집니다.

참고로 앞서 설명한 스마트 컨트랙트에서의 `grant` , `revoke`를 사용한 함수 수행 권한관리는 Admin이 아닌 스마트 컨트랙트를 정의한 주소의 소유자가 설정 가능함으로 유의하시기 바랍니다.

3. 역할 활성화 / 비활성화

가장 처음 코인스택을 구동하여 제네시스 블록만 존재할 때는 위의 역할들 중 Admin 역할만 활성화 되어 있고 하나의 Admin 주소만 존재합니다. 즉, 나머지 역할들(Writer, Miner, Node)은 비활성화되어 있습니다. 이는 초기 환경 구성 시의 편의성과 필요한 기능만 선택적으로 사용 가능하도록 하기 위함입니다.

3.1. 역할 활성화 트랜잭션 생성

역할을 활성화/비활성화 하기위해서는 Admin 권한을 가진 개인키로 SDK에서 제공하는 역할 활성화용 트랜잭션 빌더(EnableRoleBuilder)를 사용하여 트랜잭션을 생성, 네트워크에 전파하면 됩니다. 다음의 예제는 java SDK를 사용하여 프라이빗 네트워크의 Miner, Node 역할을 활성화, Writer는 비활성화 시키는 예제입니다. 참고로 Admin 역할은 항상 활성화 되어 있습니다.

JAVA

```
// 역할 활성화 트랜잭션 빌더 생성
EnableRoleBuilder enRoleBuilder = new EnableRoleBuilder();
// OR operator를 이용하여 Miner와 Node 두 역할을 지정하는 Role 객체 생성
Role role = new Role(Role.MINER|Role.NODE);
// 역할 객체 할당
enRoleBuilder.setRole(role);
// 트랜잭션 생성, 이 때 개인키는 Admin 역할을 가진 주소의 키를 사용해야 함
String rawTx = enRoleBuilder.buildTransaction(client, ADMIN_PRIVATEKEY);
client.sendTransaction(rawTx);
```

일반적인 트랜잭션과는 달린 역할 관리에 연관된 트랜잭션들은 해당 트랜잭션이 블록에 포함 된 후 적용됩니다. 또한 Admin 권한이 없는 개인키를 사용한 경우 등 역할 설정 트랜잭션에 문제가 있을 경우 트랜잭션이 블록에 포함됬더라도 설정이 변경되지 않을 수 있습니다. 이 경우 에러 메시지가 서버의 로그에 기록됩니다.

3.2. 활성된 역할 조회

현재 활성화 되어 있는 역할들을 확인해 보려면 다음과 같이 조회합니다.

JAVA

```
// CoinStackClient에서 현재 활성화 되어 있는 역할 조회
Role currentEnabledRole = client.getEnabledRole();
System.out.println(currentEnabledRole);
```

> 앞서 역할 활성화 트랜잭션이 정상적으로 수행된 경우, 수행결과는 다음과 같습니다.
>
> ```
> MINER|NODE
> ```

REST API

```
curl http://privnet.cloudwallet.io/roles/enabled
```

> 결과 값은 JSON 형식으로 반환됩니다.
>
> ```
> {
>    "permission": 12
> }
> ```

REST API로 조회한 경우 결과는 4개의 bit로 표현됩니다. 1(0001)은 Admin, 2(0010)는 Writer, 4(0100)는 Miner, 8(1000)은 Node 권한을 의미합니다. bit의 조합으로 여러권한을 의미할 수도 있는데 예를들면 12(1100)은 Miner와 Node의 권한이 활성화 되어 있다는 의미입니다. 단, Admin은 항상 활성화 되어 있기 때문에 0으로 표기됩니다.

4. 역할 설정

SDK를 이용하여 특정 주소에 역할을 부여하는 방법을 설명합니다.

4.1. 역할 설정 트랜잭션 생성

특정 주소에 역할을 설정할 때는 역할을 설정하는 마커를 가진 데이터 트랜잭션을 이용합니다. 코인스택 SDK에서는 이를 쉽게 지원하기 위한 역할 설정 트랜잭션 빌더(SetRoleBuilder)를 제공합니다. 아래는 SDK를 이용해 특정 주소에 Miner, Node 역할을 부여하고 Admin, Writer 역할을 제거하는 예입니다.

JAVA

```
// 역할 설정 트랜잭션 빌더 생성
SetRoleBuilder setRoleBuilder = new SetRoleBuilder();
// OR operator를 이용하여 Miner와 Node 두 역할을 지정하는 Role 객체 생성
Role role = new Role(Role.MINER|Role.NODE);
// 역할 객체 할당
setRoleBuilder.setRole(role);
// 역할을 부여할 대상 주소 입력
roleBuilder.setAddress(TARGET_ADDRESS);
// 트랜잭션 생성, 이 때 개인키는 Admin 역할을 가진 주소의 키를 사용해야 함
String rawTx = setRoleBuilder.buildTransaction(client, ADMIN_PRIVATEKEY);
client.sendTransaction(rawTx);
```

먼저 Role 클래스 생성자에 부여할 역할을 할당합니다. 여러개의 역할을 부여하려면 OR 연산자로 여러 역할을 조합합니다. 그리고 SetRoleBuilder에 Role객체를 할당하고 해당 Role을 부여할 대상 주소를 입력합니다. 그 후 Admin 역할을 가진 개인키를 사용하여 트랜잭션을 생성하여 전송합니다. 해당 역할은 트랜잭션이 블록에 포함된 뒤 모든 노드에 반영됩니다.

Admin은 다른 주소에 Admin 권한을 부여/박탈 가능합니다. 심지어 자기 자신의 권한도 없엘 수 있습니다. 권한 부여는 Set 형식으로 기존의 권한에 추가가 아닌 덮어쓰는 형태이기 때문에 사용에 주의가 필요합니다. 예를 들어 기존의 Admin에 new Role(Role.WRITER) 같이 역할을 할당하면 Admin 권한이 사라지고 Writer 권한만 남게 됩니다.

특정 역할이 비활성화 되어 있어도 특정 주소에 해당 역할을 미리 부여할 수 있습니다. 예를 들면 Node 역할이 비활성화 되어 있어도 노드들의 주소에 미리 Node 역할을 부여할 수 있습니다. Node 역할을 미리 노드들에 부여하지 않고 Node 역할을 활성화 시킬 경우 서로간에 Node 권한이 없다고 생각하고 노드들이 서로 연결을 끊어 블록이 전파가 안되므로 주의해야 합니다. Miner의 경우도 마찬가지로 미리 역할을 부여하고 해당 기능을 활성화 시켜야 마이너가 권한 생성 트랜잭션을 블록에 포함, 전파 시킬 수 있습니다.

4.2. 특정 주소의 역할 조회

특정 주소에 부여된 역할을 조회할 때는 CoinStackClient의 getRole 함수를 사용합니다. 아래는 권한 조회 예입니다.

JAVA

```
// CoinStackClient에서 특정 주소에 부여되어 있는 역할 조회
Role assignedRole = client.getRole(TARGET_ADDRESS);
System.out.println(assignedRole);
```

> 앞서 역할 설정 트랜잭션이 정상적으로 수행된 경우, 수행결과는 다음과 같습니다.
>
> ```
> MINER|NODE
> ```

REST API

```
curl http://privnet.cloudwallet.io/roles/{TARGET_ADDRESS}
```

> 결과 값은 JSON 형식으로 반환됩니다.
>
> ```
> {
>    "permission": 12
> }
> ```

permission 이라는 키에 조회를 요청한 주소의 역할값이 할당되어 반환됩니다.

4.3. 역할이 설정된 전체 주소 조회

현재 역할이 정의된 모든 주소와 각 주소의 역할을 조회해 보기 위해서는 다음과 같이 조회해 볼 수 있습니다.

JAVA

```
// CoinStackClient에서 역할이 부여되어 있는 전체 주소 조회
```

```
Map<String, Role> addressRoleMap = client.listAllRole();
for (Entry<String, Role> roleEntry : addressRoleMap.entrySet()) {
    System.out.printf("%s: %s\n", roleEntry.getKey(), roleEntry.getValue());
}
```

> 수행결과는 다음과 같습니다.
>
> ```
> 133N13JpiWcncKLiNJLDD5yvTxT8tazmc7: ADMIN
> 18G3MD6RdhpZbAaQiuVpvxedzQK4UmTqWs: MINER|NODE
> 1NX8A7pxaAr7A3QQtBYVLDaK17jRpzigTP: NODE
> ...
> ```

REST API

```
curl http://privnet.cloudwallet.io/roles
```

> 결과 값은 JSON 형식으로 반환됩니다.
>
> ```
> {
>    "133N13JpiWcncKLiNJLDD5yvTxT8tazmc7": 1,
>    "18G3MD6RdhpZbAaQiuVpvxedzQK4UmTqWs": 12,
>    "1NX8A7pxaAr7A3QQtBYVLDaK17jRpzigTP": 8
> }
> ```

현재 네트워크에 등록되어 있는 전체 역할의 목록이 주소를 키로, 역할을 값으로 하여 반환됩니다.

11. Additinal Features

코인스택은 범용적인 블록체인의 개발 API를 제공하지만 더 단순화 된, 혹은 반대로 성능 향상 등을 위해 더 심화된 기능을 사용하고자 하는 요청들이 있습니다. 이러한 것들 중 자주 회자되는 기능들을 모아 SDK에서 이를 지원하는 유틸리티 클래스들을 제공합니다.

1. Easy Transaction Sender Utility

코인스택의 트랜잭션은 비트코인의 표준(Standard)을 따릅니다. 단순한 트랜잭션의 경우 트랜잭션 빌더의 기본 설정으로 트랜잭션을 빌드해도 문제가 없습니다. 하지만 너무 적은 양의 비트코인을 보내려 하거나 너무 큰 사이즈의 트랜잭션을 보내고자 하는 등 예외적인 경우에는 정책상 트랜잭션이 거절 될 수 있습니다. 이에 따라 트랜잭션 빌더에서는 정책적인 부분을 세세하게 설정해 줄 수 있는 API가 존재합니다. 하지만 대다수의 사용자의 경우 표준 형식의 비트코인 트랜잭션의 정책을 숙지하고 있지 않습니다.

또한 앞에서와 같이 트랜잭션을 설정해야 하기 때문에 일반적으로 블록체인에서는 트랜잭션을 전송하기 위해서는 트랜잭션 빌더를 통한 설정, 생성, 코인스택 클라이언트를 통한 전송의 3단계를 거치게 됩니다. 그러나 이에 따른 중복된 코드 사용이 발생하고 코드의 이해도가 낮아지는 경우가 있습니다.

이러한 측면에서 사용자 편의의성을 위해 코인스택에서는 자동적으로 표준 형식에 맞추기 위한 TransactionBuildTip이라는 클래스와, 한 단계로 트랜잭션을 전송 할 수 있는 TransactionSender라는 유틸리티 클래스를 제공합니다.

1.1. Transaction Build Tip

트랜잭션 생성 팁(TransactionBuildTip)은 이후 설명할 TransactionSender에서 표준 트랜잭션을 생성하기 위해 필요한 가이드를 제공합니다. SDK에서는 일반적으로 많이 쓰이는 설정을 모아 다음과 같이 두가지의 팁을 제공합니다.

1.1.1. Standard Tip

일반적인 비트코인 표준 수수료 정책을 따릅니다. 다음과 같은 사전 정의된 정책을 사용합니다.

- 기본 수수료: 0.00001 btc
- KB 당 (크기에 따른) 수수료: 0.00001 btc
- UTXO 최대치 가이드: 100개

기본 수수료는 무조건 하나의 트랜잭션에 할당되는 최소 금액의 트랜잭션입니다. KB당 수수료는 트랜잭션 사이즈가 커짐에 따라 KB당 추가로 부가할 수수료 입니다. UTXO 최대치 가이드는 한 트랜잭션에 Input으로 너무 많은 UTXO를 넣을 경우 트랜잭션 생성에 시간이 오래 걸리므로 최대 해당 개수만큼만 사용하도록 가이드 합니다. 위의 100개를 예로 들면 트랜잭션을 보낼 주소의 UTXO가 100개를 넘을 경우, 트랜잭션의 'output + 수수료'만큼의 비트코인 합계를 가지도록 UTXO를 일부만 선택하여 트랜잭션의 입력으로 넣게 됩니다. 하지만 UTXO 각각의 금액이 작을 경우 100개의 UTXO를 더해도 보낼 총 량보다 적을 경우에는 100개를 넘겨 필요한만큼 입력합니다.

JAVA

```
TransactionBuildTip standardTip = TransactionBuildTip.STANDARD;
```

1.1.2. Zero Fee Tip

트랜잭션에 수수료를 항상 0으로 설정합니다. 수수료를 받지 않아도 트랜잭션을 수용하도록 설정된 코인스택 프라이빗 네트워크일 경우에만 사용하기를 권합니다. (코인스택 서버가 다음과 같이 설정되어야 합니다. minrelaytxfee=0, blockprioritysize={블록사이즈와 동일}) 기본 비트코인 정책을 사용하도록 설정된 네트워크에서는 이 팁을 사용해 성성한 트랜잭션은 수수료가 부족하여 블록에 포함되지 않을 수 있습니다.

- 기본 수수료: 0 btc
- KB 당 (크기에 따른) 수수료: 0 btc
- UTXO 최대치 가이드: 10개

JAVA

```
TransactionBuildTip zerofeeTip = TransactionBuildTip.ZEROFEE;
```

네트워크가 수수료 0인 트랜잭션을 허용하는 경우 트랜잭션 팁과 샌더를 사용하지 않고 기존 처럼 트랜잭션 빌더를 통해 트랜잭션을 만드는 경우에도 setFee를 통해 수수료를 0으로 설정하여 보내는게 가능합니다.

1.2. Transaction Sender

트랜잭션 샌더는 트랜잭션 설정, 빌드, 전송의 3 단계를 한 단계로 단순화 하여 트랜잭션 전송을 쉽게 할수 있도록 돕는 유틸리티 클래스 입니다. 트랜잭션 샌더는 앞서 설명한 트랜잭션 생성 팁을 인자로 입력 받습니다. 다음은 트랜잭션 샌더의 생성 예 입니다.

JAVA

```
// standard tip을 사용할 경우
TransactionBuildTip txBuildTip = TransactionBuildTip.STANDARD;
// transaction sender 인스턴스 생성
TransactionSender txSender = new TransactionSender(coinstackClient, MY_PRIVATEKEY, txBuildTip);
```

이렇게 생성한 트랜잭션 샌더를 이용하면 아래와 같은 기능들을 생성자에 입력한 개인키와 트랜잭션 생성 팁을 사용해 쉽게 수행할 수 있습니다.

1.2.1. Send Coin

내 계좌로부터 대상 주소에 입력한 수량만큼의 비트코인을 보냅니다. 아래는 SDK를 이용해 대상 주소에 3 btc를 보내는 예입니다. 한번에 하나의 주소에만 가능하며 잔고가 부족할 경우 실패합니다.

```
txSender.sendCoin(TARGET_ADDRESS, 3);
```

1.2.2. Write Data

대상을 지정하지 않은 데이터 트랜잭션을 생성합니다. 스탬핑 등에 사용합니다. 아래는 SDK를 이용해 사용자 데이터(SERIALIZED_DATA_BYTE)를 기록하는 예입니다.

```
txSender.writeData(SERIALIZED_DATA_BYTE);
```

1.2.3. Send Data

대상을 지정하는 데이터 트랜잭션을 생성합니다. Openkeychain 등에 사용합니다. 아래는 SDK를 이용해 대상 주소에 사용자 데이터를 기록하는 예입니다.

```
txSender.sendData(TARGET_ADDRESS, SERIALIZED_DATA_BYTE);
```

1.2.4. Define Contract

스마트 컨트랙트를 정의합니다. 아래는 SDK를 이용해 대상 컨트랙트 주소에 스마트 컨트랙트를 정의하는 예입니다. 일반적으로 대상 컨트랙트 주소는 트랜잭션 샌더를 생성했을 때 입력한 개인키에서 유추된 주소여야 합니다. 즉, 해당 주소의 소유자이여야 해당 주소에 컨트랙트를 정의 가능합니다.

```
txSender.defineContract(TARGET_CONTRACT_ID, luaContractDefinitionCode);
```

1.2.5. Execute Contract

스마트 컨트랙트를 수행합니다. 아래는 SDK를 이용해 대상 컨트랙트 주소에 스마트 컨트랙트를 수행하는 예입니다. 컨트랙트 수행은 정의와 달리 어떠한 주소도 가능합니다. 이를 제한하기 위해서는 앞서 설명한 스마트 컨트랙트 권한 grant 기능을 참조하시기 바랍니다.

```
txSender.executeContract(TARGET_CONTRACT_ID, luaContractExecutionCode);
```

2. Special NullData: Address-Data

코인스택에서는 Openkeychain과 스마트 컨트랙트 등 OP_RETURN으로 데이터를 보낼 시 특정 주소에 600 사토시를 보내는 트랜잭션 아웃풋(dusty 아웃풋)을 추가해 대상 주소(e.g. 수행 대상 스마트 컨트랙트, 인증서 발급 주소)에 이력을 남기고 있습니다. 이렇게 하는 이유는 한 트랜잭션에는 하나의 Null Data 트랜잭션 아웃풋만 허용되기 때문입니다. 하지만 이로 인해 트랜잭션을 쓰기위해 600 사토시가 계속 소모되고, 상대방의 주소에는 사용 불가능한 UTXO가 지속적으로 쌓이게 됩니다.

이러한 이슈를 해소하기 위해 코인스택에서는 Address-Data라는 현재 트랜잭션을 받는 대상을 표현하는 Null Data를 제공합니다. 트랜잭션에 Address-Data로 대상 주소를 설정하면 대상의 트랜잭션 이력(Transaction History)에 해당 트랜잭션이 보이나 UTXO가 새로이 생기지는 않습니다.

앞서 설명한 수수료 무료 네트워크에 이 기능을 조합해서 사용하면 데이터 트랜잭션을 무제한 만들 수도 있습니다. 최소 하나의 UTXO는 필요하므로 초기에 필요한 만큼 충전을 해줘야 합니다. 단, 이 기능은 쓰면 한 트랜잭션에 Null Data가 2개 이상 존재하는 비 표준 트랜잭션을 생성합니다. 따라서 이 기능은 비 표준 트랜잭션을 허용하는 프라이빗 네트워크에만 사용 가능합니다.

TransactionBuilder의 addDataAddr(String dataAddress) 함수를 이용하면 직접 사용자 지정 Address-Data를 추가할 수 있습니다. 함수 명에서 알 수 있듯이 addOutput처럼 여러개의 주소를 할당할 수 있습니다.

```
TransactionBuilder txBuilder = new TransactionBuilder();
// tx output의 순서를 맞추기 위해 shuffleOutput을 끄는게 좋습니다.
txBuilder.shuffleOutputs(false);
// 첫번째 대상 주소를 추가합니다.
txBuilder.addDataAddr(TARGET_ADDR_1);
// 두번째 대상 주소를 추가합니다.
txBuilder.addDataAddr(TARGET_ADDR_2);
txBuilder.
```

그 외에 코인스택에서 제공하는 Openkeychain, 스마트 컨트랙트, TransactionSender 등에서 Address-Data를 사용하도록 하려면 다음과 같이 CoinStackClient을 설정해 야 합니다.

```
coinstackClient.initUseAddrData(true);
```

위와 같이 CoinStackClient를 설정하면 이후 이를 사용해 생성하는 코인스택이 제공하는 프로토콜(Openkeychain 등)의 트랜잭션들은 데이터 대상을 지정할때 dusty 아웃풋 아닌 address-data를 기본으로 사용하게 됩니다.

3. Utxo Cache

비트코인 트랜잭션은 UTXO 기반으로, 트랜잭션을 생성하기 위해서는 사용하지 않은 다른 트랜잭션의 아웃풋을 입력으로 받아야 합니다. 코인스택에서는 사용자가 클라이언트에서 직접 UTXO를 관리하는 부담을 줄이기 위해 트랜잭션 빌더를 통해 트랜잭션을 생성하는 시점에 코인스택 서버로 부터 매번 사용 가능한 UTXO를 조회해 사용하도록 되어 있습니다. 그러나 이로 인해 트랜잭션 생성 시 추가적인 시간(Latency)이 필요하게 됩니다. 따라서 매번 서버로부터 UTXO를 가져오지 않고 UTXO를 SDK 레벨에서 캐싱해 두고 필요한 만큼 사용하게 하면 UTXO 조회 시간만큼 트랜잭션 생성 시간을 줄일 수 있습니다.

단, 캐시의 효과를 보려면 개인키를 여러 서비스나 쓰래드가 공유하지 않고 하나의 서비스가 하나의 개인키로 여러번의 트랜잭션을 지속적으로 전송하는 경우여야만 합니다. 만약에 여러 서비스나 쓰래드에서 동시에 하나의 개인키를 공유해서 쓸 경우 double spent가 발생하므로 절대 하나의 개인키를 여럿이 공유하지 않아야 합니다. 여러개의 서비스가 블록체인에 접근할 경우 서비스 마다 다른 개인키를 할당하여 사용하십시오.

캐시 기능을 사용하려면 SDK에서 CoinStackClient 객체를 다음과 같이 설정합니다.

```
coinstackClient.initUtxoCache(cacheSize);
```

여기에 입력하는 cacheSize 사용할 개인키의 개수만큼 (e.g. 서비스가 하나의 개인키만 쓴다면 1을, 2개의 개인키를 필요에 따라 번갈아 가며 사용한다면 2를) 입력하는 것을 추천합니다. 여기에 입력한 cacheSize 수 만큼의 개인키에 대한 UTXO(저장하는 UTXO는 개수 무제한) 최대 1분동안 유지합니다. 캐시는 처음 트랜잭션 빌드 시 채워지고 트랜잭션 전송시 업데이트 됩니다. 트랜잭션 전송 에러가 발생하거나 잔고가 부족하면 캐시를 삭제하고 서버로 부터 다시 UTXO를 가져옵니다.

API Reference

코인스택 API는 모든 데이터를 객체로 취급하여 주고 받습니다. 코인스택 API를 호출할 때 파라미터로 코인스택 객체를 제공해야 하고, 또한 그 결과로 코인스택 객체를 반환받게 되므로 어떤 종류가 있는지, 어떤 속성을 갖는지 본 문서를 통해 확인하시기 바랍니다.

코인스택 API는 다음과 같이 다양한 종류의 객체를 제공하지만, 대부분 블록체인의 데이터 모델에 대응하여 설계되어 있으므로 쉽게 파악할 수 있습니다.

BlockchainStatus

BlockchainStatus는 블록체인의 상태 정보를 나타내는 객체입니다.

Attribute	Type	Description
best_block_hash	string	가장 최신의 블록체인 블록 해시
best_height	number	블록체인의 현재 높이, 즉 가장 최신의 블록 번호

Block

Block은 블록체인을 구성하는 특정 블록의 상태 정보를 나타내는 객체입니다.

Attribute	Type	Description
block_hash	string	블록의 해시
height	number	블록의 높이
confirmation_time	date	블록이 승인된 시간
parent	string	블록의 부모 블록 해시
children	array[string]	블록의 자식 블록 해시 목록
transaction_list	array[string]	요청한 블록에 포함된 트랜잭션 해시 목록

Address

Address는 비트코인 주소를 의미하며, Address 관련 객체들은 아래와 같습니다.

1. Address Balance

Address Balance는 현재 특정 비트코인 주소에서 다른 주소로 송금 가능한 잔액을 사토시 단위로 나타내는 객체입니다.

Attribute	Type	Description
balance	number	주소의 사용 가능한 잔고 (사토시 단위)

2. Address History

Address History는 특정 비트코인 주소와 관련된 트랜잭션들의 해시값 목록을 가지는 객체입니다.

Attribute	Type	Description
(없음)	array[string]	주소에 발생한 트랜잭션 해시 목록

3. Address Unspent Outputs

Address Unspent Output은 특정 비트코인 주소의 잔액을 구성하는, 소비되지 않은 출력값에 관한 객체입니다.

Attribute	Type	Description
transaction_hash	string	output이 속한 트랜잭션 해시
index	number	output index
value	string	output의 값 (사토시 단위)
script	string	output의 script
confirmations	number	output이 속한 트랜잭션이 승인된 회수

Transaction

Transaction은 블록에 저장된 주소 간의 거래 정보인 트랜잭션을 나타내는 객체입니다.

Attribute	Type	Description
transaction_hash	string	트랜잭션의 해시
block_hash	list[string]	트랜잭션이 포함된 블록의 해시
block_hash.block_hash	string	트랜잭션이 포함된 블록의 해시
block_hash.block_height	number	트랜잭션이 포함된 블록의 높이
coinbase	boolean	Coinbase 트랜잭션 여부
inputs	array[object]	Transaction Input 목록
outputs	array[object]	Transaction Output 목록
timestamp	string	트랜잭션이 포함된 블록이 최초 승인된 시간 - date 참조
initialtimestamp	string	트랜잭션이 broadcast된 시간 - date 참조
addresses	array[string]	트랜잭션과 관련된 주소 목록

1. Transaction Input

Transaction Input은 특정 트랜잭션의 입력값을 나타내는 객체입니다. 하나의 트랜잭션에는 하나 이상의 입력값이 존재할 수 있으며, Transaction 객체에는 이들의 목록이 배열로 저장됩니다. Transaction Input은 이 목록을 구성하는 개별 입력값 단위에 해당합니다.

Attribute	Type	Description
transaction_hash	string	input이 포함된 트랜잭션
output_index	number	해당 트랜잭션에서의 output index
address	array[string]	해당 input과 관련된 주소 목록
value	number	input 값 (사토시 단위)

2. Transaction Output

Transaction Output은 특정 트랜잭션의 출력값을 나타내는 객체입니다. 하나의 트랜잭션에는 하나 이상의 출력값이 존재할 수 있으며, Transaction 객체에는 이들의 목록이 배열로 저장됩니다. Transaction Output은 이 목록을 구성하는 개별 출력값 단위에 해당합니다.

Attribute	Type	Description

index	number	
address	array[string]	output과 관련된 주소 목록
script	string	output의 script
value	number	output 값 (사토시 단위)
used	boolean	(Optional) output 의 사용 여부
data	string	(Optional) OP_RETURN 데이터

Stamp

코인스택 고유 기능인 문서 진위 확인 서비스(Document Stamping)에서 사용하는 객체입니다.

Attribute	Type	Description
tx	string	스탬프 정보가 기록되어 등록된 트랜잭션 해시
vout	number	상기 트랜잭션의 출력값 중 스탬프 정보가 기록된 위치
confirmations	number	상기 트랜잭션이 속한 블럭의 승인된 횟수
timestamp	string	상기 트랜잭션이 속한 블럭이 최초 승인된 시간 - date 참조