블록체인 해커톤에서 Dapp 개발하기

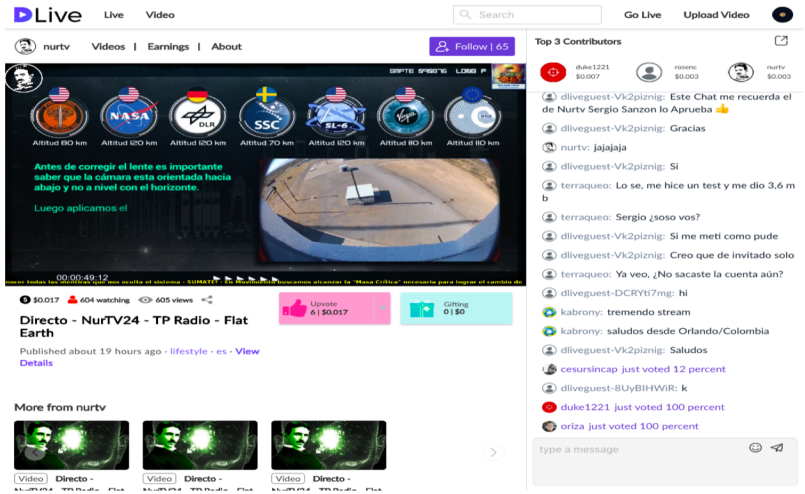
목차

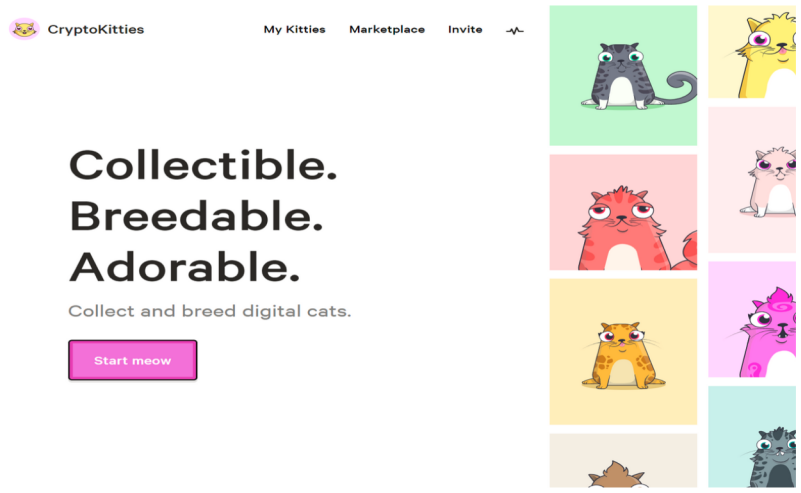
1. Dapp이란 무엇인가?

1.1 Dapp이란?

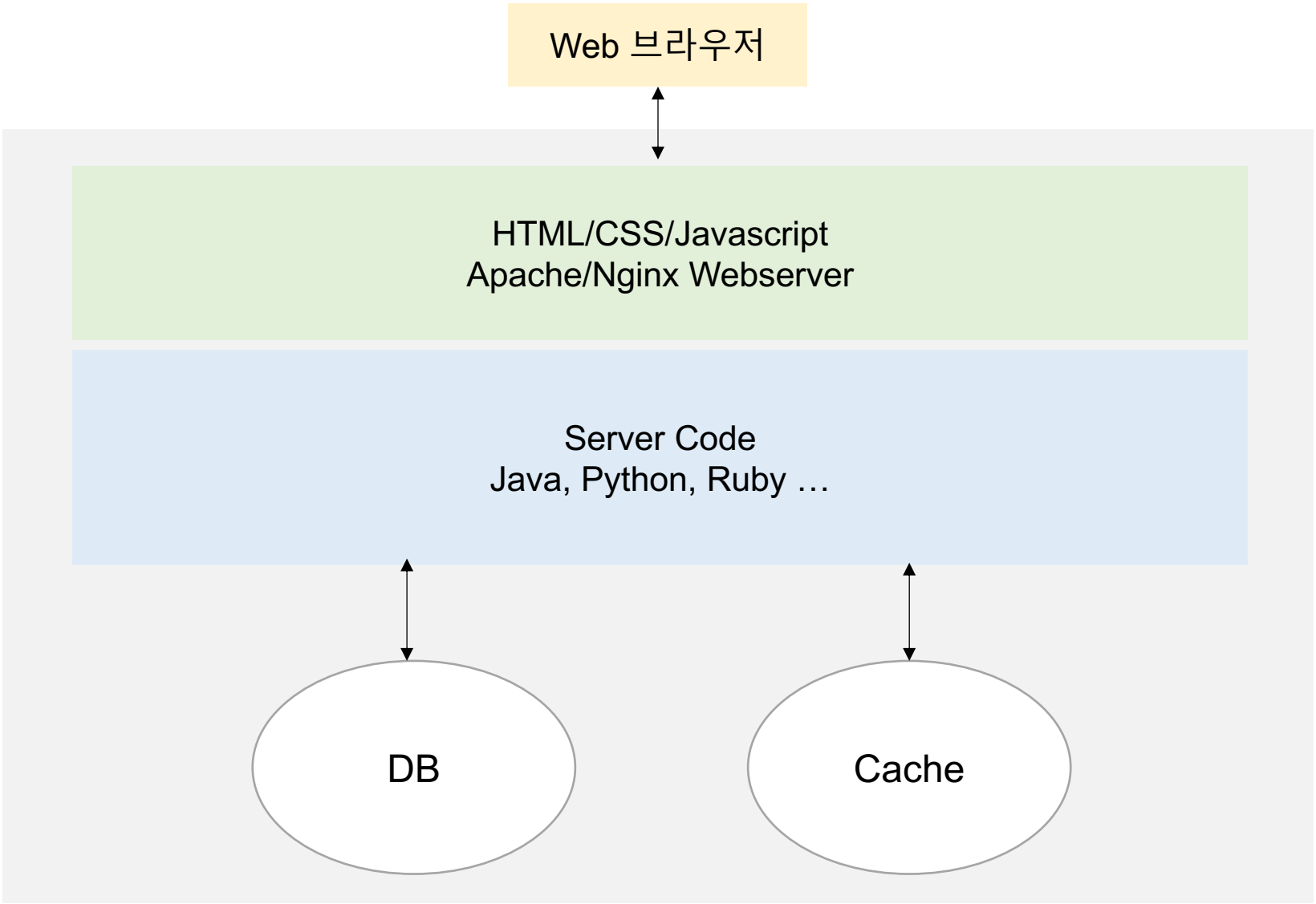
Decentralized Application

- ✓ Dapp은 Decentralized Application의 약자로 블록체인 기술을 활용한 탈중앙화된 어플리케이션.
- ✓ 코드가 탈중앙화된 peer-to-peer network 위에서 작동하고, 데이터 호출 및 등록을 블록체인 데이터베이스로 사용하는 애플리케이션.
- ✓ 이더리움 블록체인에 Dapp이 저장되며, EVM이 Dapp을 실행시키고, 처리 결과들을 블록체인에 기록.
- ✓ 누구나 실행 가능하며, 셧다운이 없고, 접속제한도 없음.

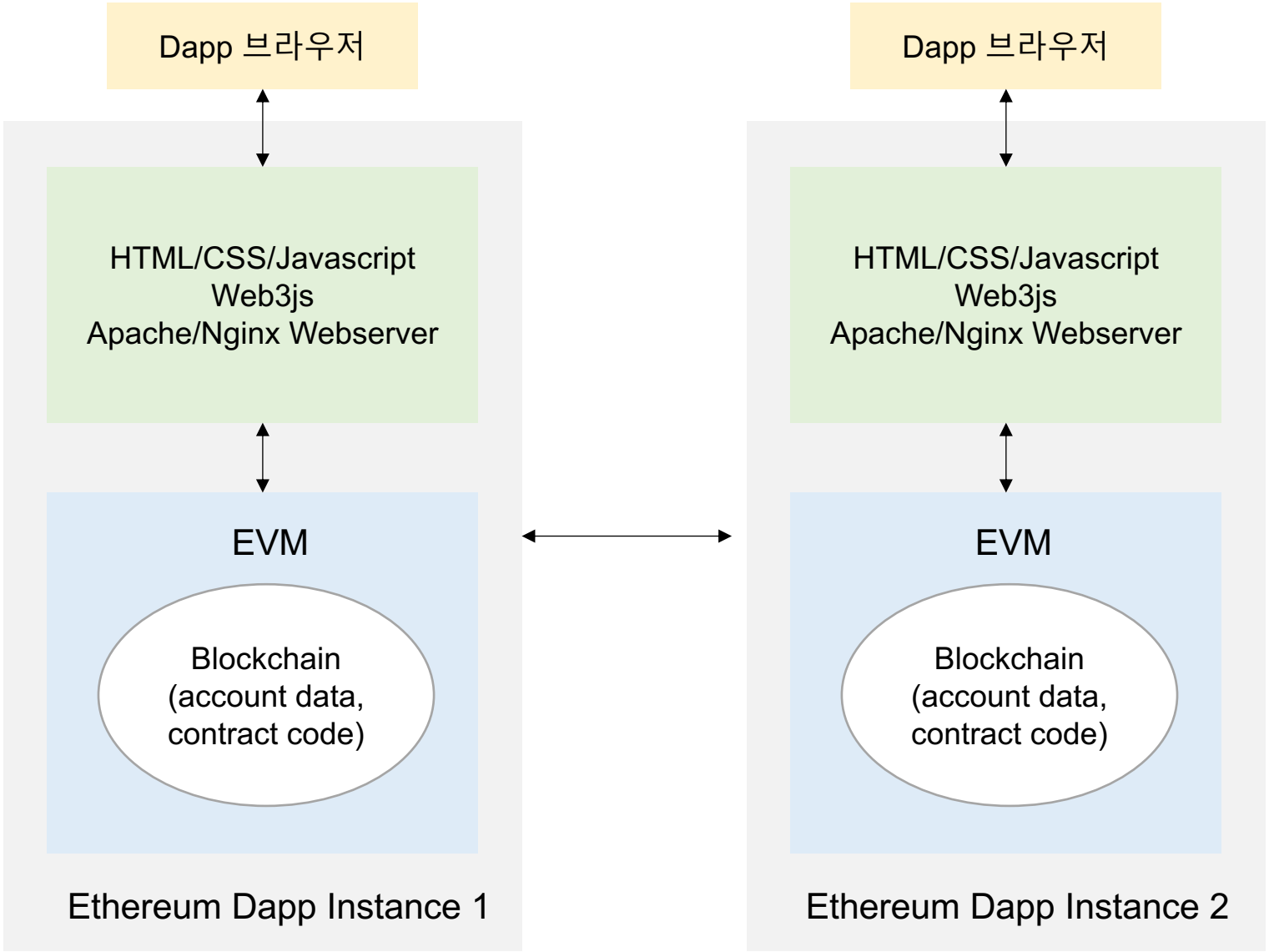




1.2 Web App Architecture



1.3 Ethereum Dapp Architecture



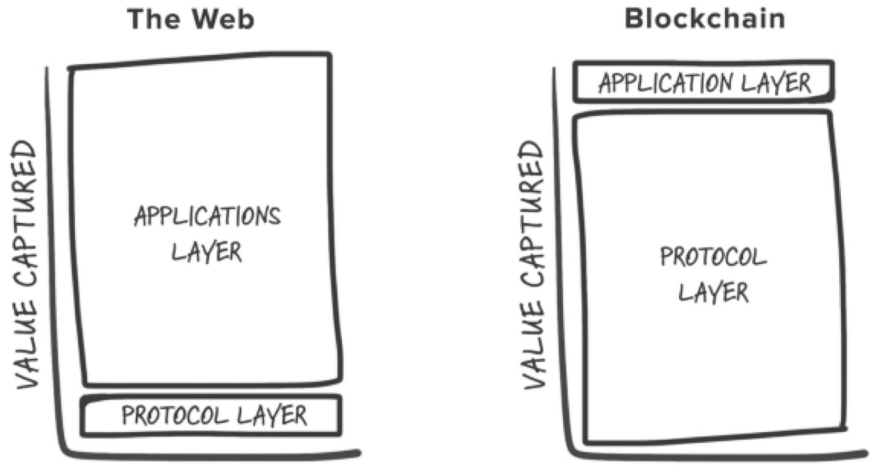
1.4 Dapp과 프로토콜

프로토콜(Protocol)이란?

- ✓ 프로토콜(Protocol) 또는 통신 규약은 우리가 통신하는 방식을 의미.
- ✓ 인터넷에서 대표적인 프로토콜은 TCP(Transmission Control Protocol)과 IP(Internet Protocol).

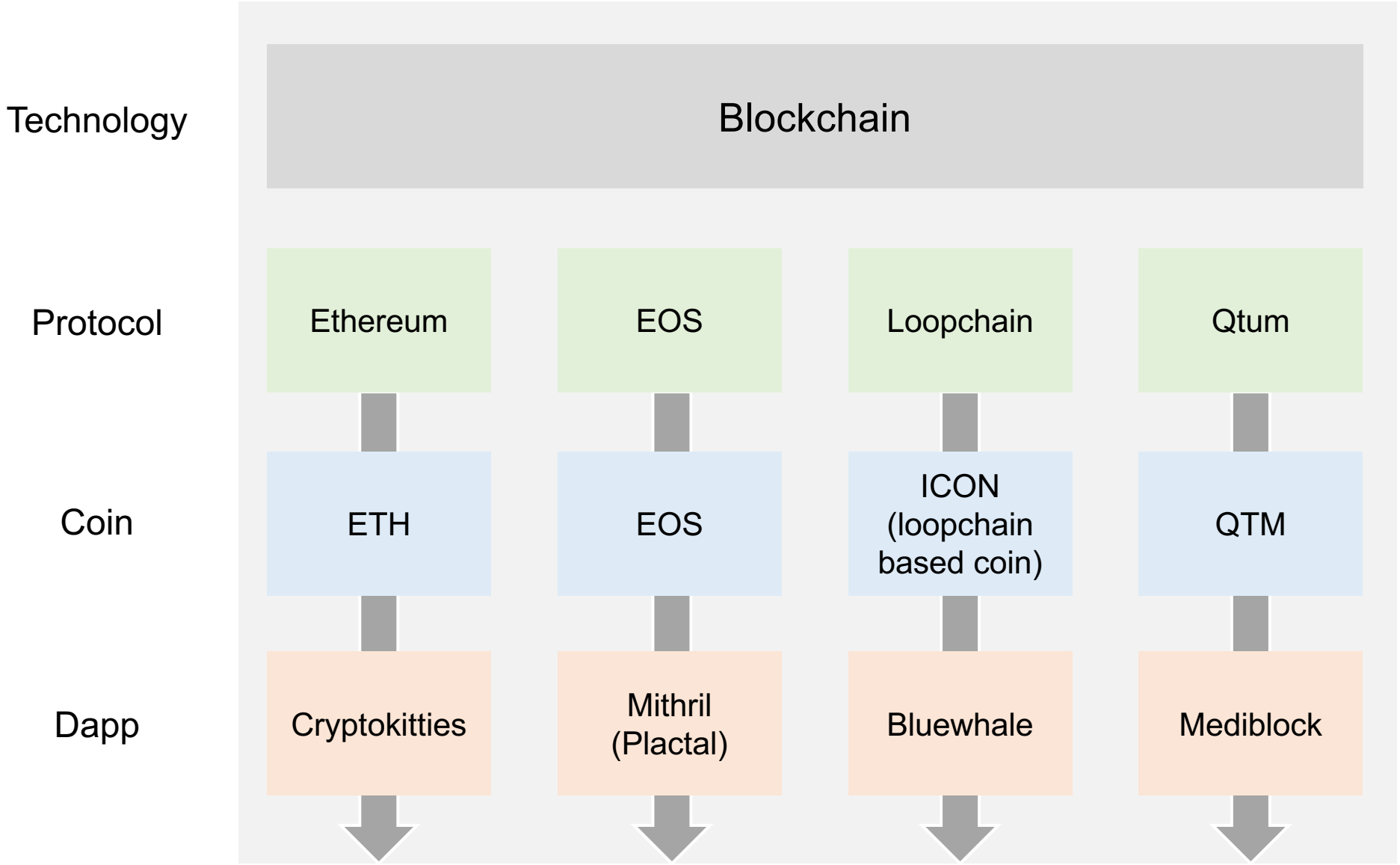
블록체인에서 프로토콜의 의미

- ✓ 블록체인 기술을 바탕으로 비트코인과 이더리움이 만들어졌음.
- ✓ 비트코인, 이더리움 등이 각 노드(Node)들을 연결하는 방식을 프로토콜로 지칭.



Joel Monegro의 Fat Protocol

1.5 Dapp과 프로토콜



2. 해커톤에서 Dapp 개발 노하우

2.1 아이디어 빌드하기

2.1.1 아이디어 체크 리스트

- [ ] 신뢰받는 제 3자가 필요한 아이디어인가?
- [ ] 데이터를 나눠서 저장해야 하는가?
- [ ] 빠른 처리 속도가 필요한 아이디어인가?
- [ ] 의사 결정이 타인에 의해 이루어져도 되는 아이디어인가?
- [ ] 거래의 투명성이 필요한 아이디어인가?

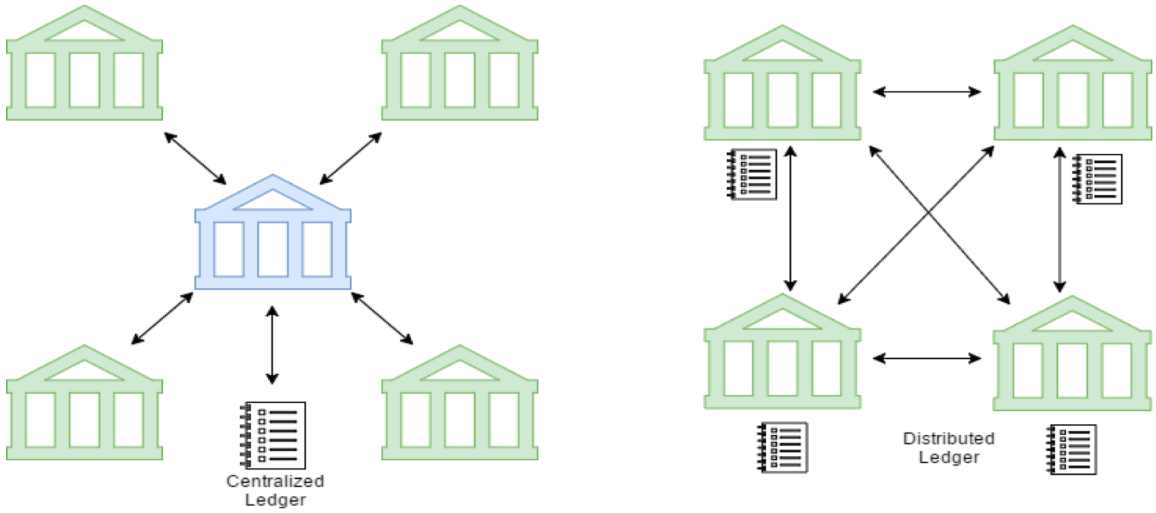
2.1.2 신뢰받는 제 3자가 필요한 아이디어인가?

▌블록체인은 신뢰받는 제 3의 중개기관을 배제시킬 수 있음.

- ✓ 다양한 경제 주체들이 개입되어 있는 비지니스에서 서로 간의 신뢰가 필요할 때 신뢰받는 제 3의 중개기관이 개입되어 신뢰 문제를 해결하고 있다.
- ✓ 에스크로 서비스, 은행 간 해외 송금 시 중개기관, 공인 인증서 발급 등이 신뢰를 제공하는 제 3자의 역할을 하는 것이다.
- ✓ 블록체인이 신뢰가 필요한 산업의 해결책이 되기 위해서는 신뢰받는 제 3의 중개 기관이 있음으로 산업의 비효율성을 초래하고 있는지 산업 내부 프로세스를 면밀히 파악 해야 한다.

2.1.3 데이터를 나눠서 저장해야 하는가?

▌블록체인과 기존 시스템과의 데이터 저장 차이점을 인지할 것.



- ✓ 블록체인은 하나의 노드가 변화하면 그에 따라 수많은 노드가 상태 값을 변경하는 구조이므로 저장된 내용의 변경이 많이 일어나거나 소량의 정보만을 저장하는 경우 블록체인이 적합하다
- ✓ 만약 데이터를 많이 수정할 필요가 없거나 대용량의 데이터를 저장해야 하는 사업의 경우 자료를 분산하여 저장하기 보단 기존 시스템과 같이 중앙 집중화된 저장이 효율적인 경우 기존 시스템을 사용하는 것이 효과적이다

2.1.4 빠른 처리 속도가 필요한 아이디어인가?

▌ 블록체인의 낮은 확장성을 인지해 서비스 모델에 적용할 것.

VS

- ✓ "아마존 EC2(가상 서버 서비스)의 미디엄 사이즈 가격은 시간 당 0.04 달러다. 이더리움의 오버헤드(같은 일을 처리는 데 드는 비용)는 그 100만 배 수준이다”
- ✓ 이더리움이 20 TPS인데 반해 비자 네트워크는 1667 TPS인 점을 감안할 시 빠른 처리를 요하는 시스템의 적용에는 현실적으로 어려움이 있다

2.1.4 빠른 처리 속도가 필요한 아이디어인가?

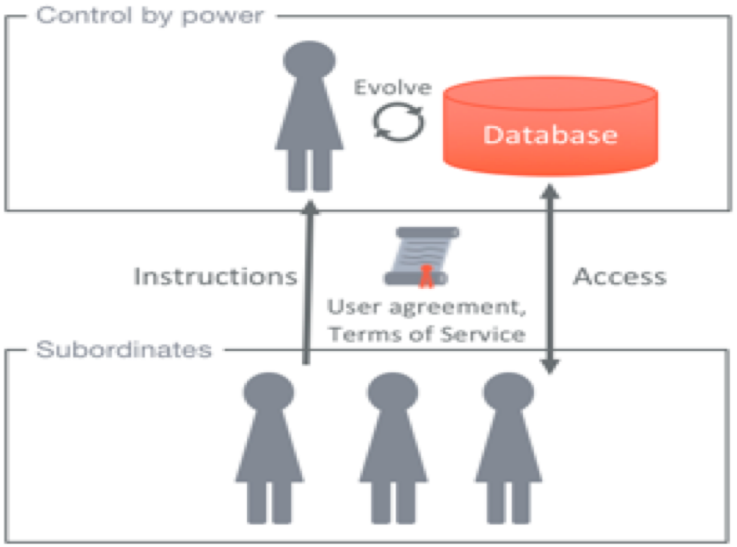
▌블록체인의 낮은 확장성을 인지해 서비스 모델에 적용할 것.

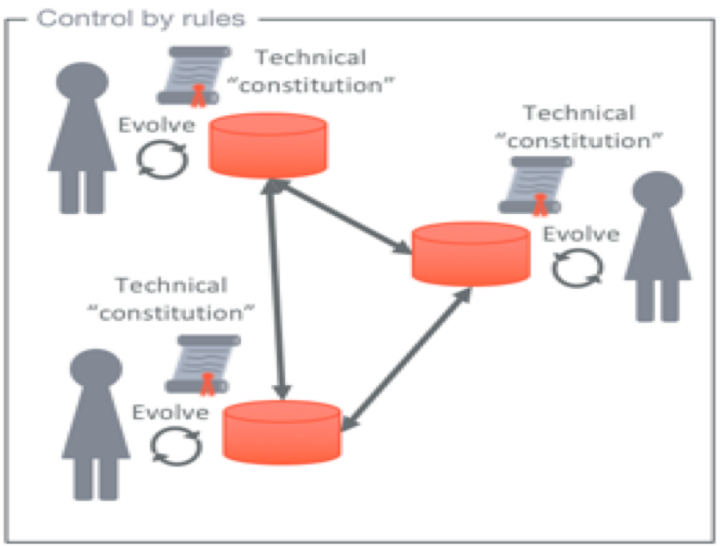
- 확장성은 아래 요소들을 포괄한다

1) 데이터 처리 속도(프로세싱, processing speed)
2) 데이터 검증 속도(검증, validation speed)
3) 데이터 전파 속도(네트워크, network relay speed)
4) 늘어나는 데이터 크기(저장공간, storage)
5) 합의구조에 참여하는 노드 갯수(검증인 수, the number of validators)
6) 활용 및 응용 프로그램의 다양성(어플리케이션, scalability on codes and applications)
7) 온체인/오프체인 거버넌스(on-chain/off-chain governance)

2.1.5 의사 결정이 타인에 의해 이루어져도 되는 아이디어인가?

▌서비스 모델의 내용 및 의사 결정 방식에 따라 블록체인 도입 여부를 결정.

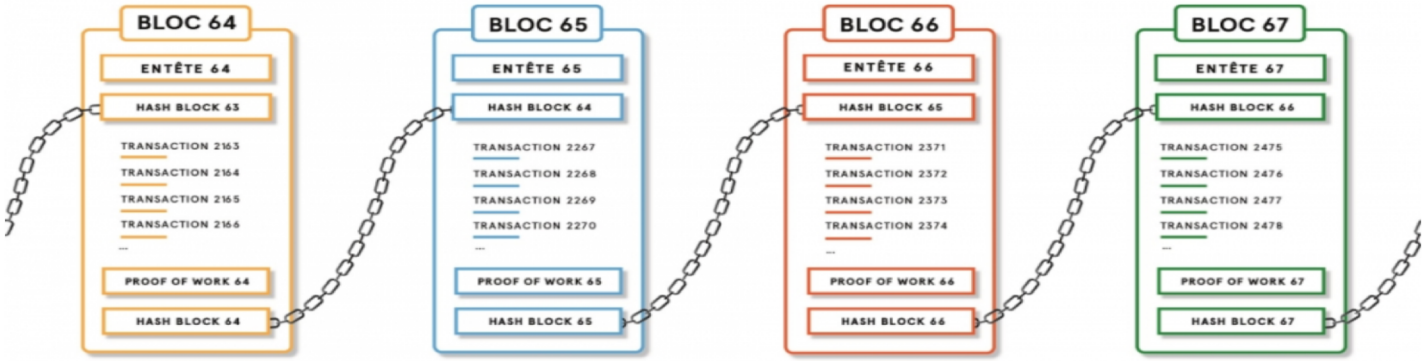




- ✓ 중앙 집중화 시스템의 경우 의사 결정을 독단적으로 가능했다면, 블록체인 위에서는 토큰이 발행되고, 이에 투자한 사람이나 컴퓨팅 파워를 제공한 사람에게 의사결정 권한이 나눠진다
- ✓ 따라서 의사결정 과정이 투명하게 공개되어야 하는 사업의 경우 오히려 블록체인 기반 의사결정이 효율적일 수 있으나, 장기적인 안목을 가지고 진행해야 하는 사업의 경우 블록체인 네트워크 상의 참여자들에 의해 의사결정 방향이 왜곡될 수 있다

2.1.6 데이터의 투명성이 필요한 아이디어인가?

▌데이터의 투명성을 입증해야 하는 서비스 모델의 경우 블록체인 적용이 타당.



- ✓ 블록체인은 모든 데이터가 암호화되어 시간 순으로 블록이 연결되어 있어 과거의 기록을 수정 및 위, 변조 하는 것이 불가함.
- ✓ 이러한 블록체인의 특성으로 감독 기관 및 규제 준수와 더불어 거래의 투명성을 보장해야 하는 사업의 경우 블록체인의 도입이 적합함. 또한 기존의 감독 및 관리 시스템 대비 낮은 비용이 발생하는 이점이 있음.

2.1.6 데이터의 투명성이 필요한 아이디어인가?

▌데이터 공개 여부를 결정.

- ✓ 블록체인이 제공하는 매우 극단적인 투명성은 블록체인의 최대 장점이자 단점으로 볼 수 있다.
- ✓ 개인 거래 내역이나 개인 민감 정보, 업체와의 거래 내역 등을 참여 노드 모두에게 공개할 시 문제점으로 발생할 수 있기 때문이다.
- ✓ 이로 인하여 블록체인 네트워크에 참여할 수 있는 요건 여부 등으로 구분되어 지는 Private Blockcahin과 Public Blockchain 중 어떠한 블록체인을 적용할 지 결정해야 한다

2.1.6 데이터의 투명성이 필요한 아이디어인가?

▌데이터 공개 이외에도 Private과 Public 블록체인의 차이를 인지할 것.

구분	Public Blockchain	Private Blockchain
네트워크 참여자	불특정 다수	신원이 판명된 참여자
거래 생성 및 검증	불특정 다수	신원이 판명된 참여자
프라이버시	개방	서비스 채널 암호화
처리속도	7~15 TPS	1,000 TPS 이상
블록확정 시간	15초 이상	1초 미만
예시	비트코인, 이더리움 등	loopchain, Hyperledger, R3 등

2.2 개발 시작!

2.2.1 온체인 개발 요소를 구분해서 설계

블록체인 특성을 고려

✓ 블록체인은 네트워크에 참여하는 모든 노드가 모든 거래를 기록하게 설계돼 있어, 네트워크 전체의 처리 효율이 떨어짐.

✓ 컨트랙트 비즈니스 로직의 복잡성을 최소화하기.

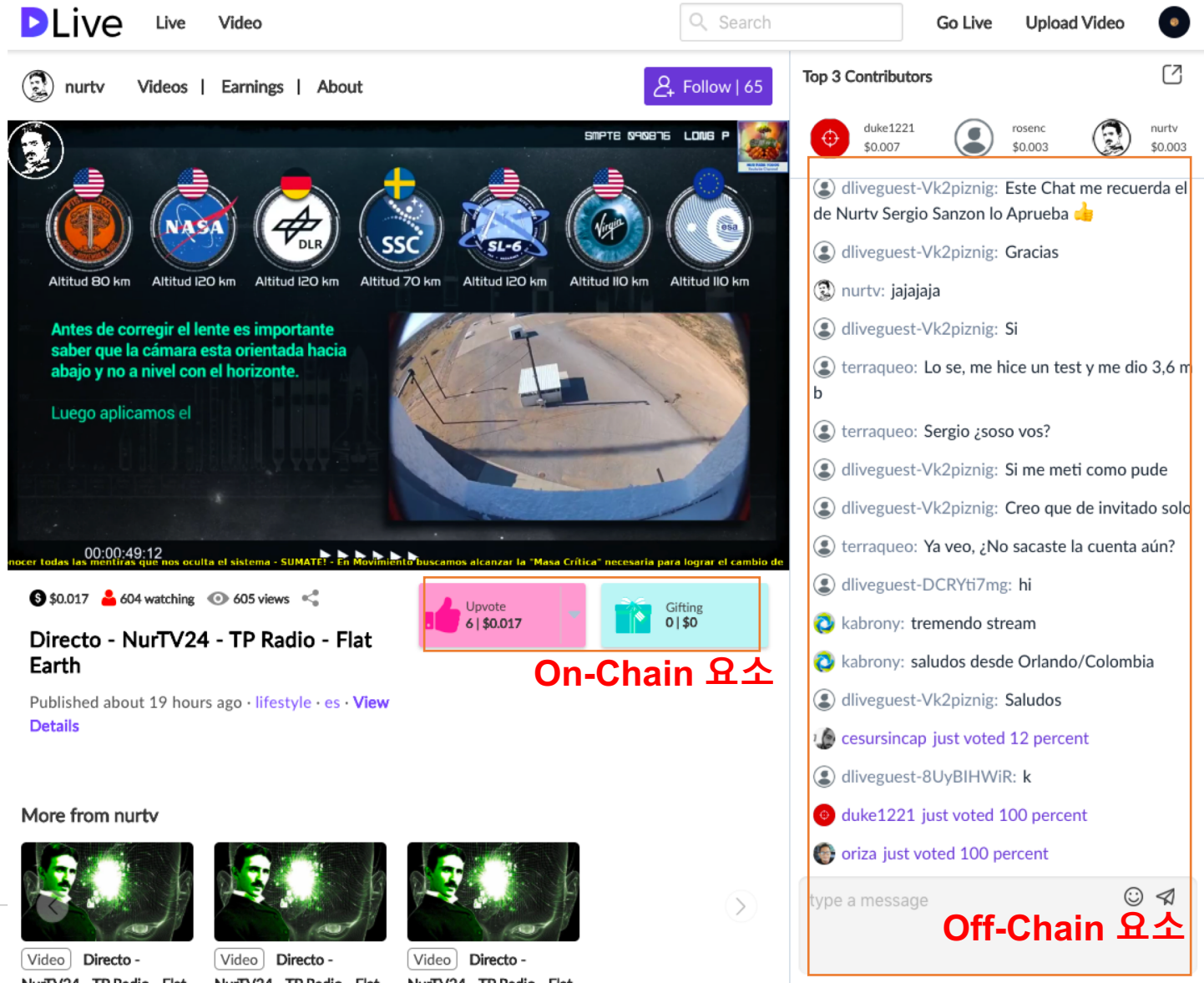
- 현재는 파일 공유, 머신 러닝 등 복잡한 프로그램을 돌리는 것은 거의 불가능하며, 간단한 조건문이나 인증 절차 등 단순 태스크에 적합.

이 경우에 온체인에 탑재하는 것은 적절하지 않다

- ✓ 크고 복잡한 데이터
  - 데이터를 많이 수정할 필요가 없거나 데이터베이스에 상당히 큰 데이터를 저장하는 경우 적합하지 않음. 자체 DB에 저장, 지불 절차만 컨트랙트를 활용하기.
- ✓ 개인 정보, 논란이 될만한 정보.
- ✓ 기밀 데이터가 포함되는 비즈니스 로직.
- ✓ 데이터 수집 및 저장을 위해 외부 서비스를 이용해야 하는 경우에는 적절하지 않음.

2.2.2 온체인 개발 요소를 구분해서 설계

컨트랙트에는 꼭 필요한 요소만



2.2.3 수수료를 줄이기

코드 실행 횟수를 최소화하기

	A	B	C	D	E	F	G	H	I	J
1		Approximations						Gas price		
2	Param	Compute (µs)	History (bytes)	State (bytes)	Bandwidth	Bloom topic	Mem quad	Computed	**Actual**	
3	DUP	3						3	**3**	FASTESTSTEP
4	SWAP	3						3	**3**	FASTESTSTEP
5	PUSH	3						3	**3**	FASTESTSTEP
6										
7	ADD	3						3	**3**	FASTESTSTEP
8	MUL	5						5	**5**	FASTSTEP
9	SUB	3						3	**3**	FASTESTSTEP
10	DIV	5						5	**5**	FASTSTEP
11	SDIV	5						5	**5**	FASTSTEP
12	MOD	5						5	**5**	FASTSTEP
13	SMOD	5						5	**5**	FASTSTEP
14	ADDMOD	8						8	**8**	MIDSTEP
15	MULMOD	8						8	**8**	MIDSTEP
16	EXPBASE	10						10	**10**	SLOWSTEP
17	EXPBYTE	10						10	**10**	
18	SIGNEXTEND	5						5	**5**	FASTSTEP
19	LT	3						3	**3**	FASTESTSTEP
20	GT	3						3	**3**	FASTESTSTEP
21	SLT	3						3	**3**	FASTESTSTEP

- ✓ 이더리움은 '튜링 완전'한 스마트 컨트랙트가 특정 시점이 되면 종료되도록 하기 위해 각 '실행 코드' 당 가격을 매겼음.
- ✓ 반복 횟수가 고정되어 있지 않은 루프는 신중하게 사용해야 함. 명시적으로 루프 반복 회수를 고정하기.
- ✓ 불필요한 반복문, 조건문 등을 최소화해서 실행 횟수를 줄이기.
- ✓ 사용되는 가스 총량이 가스 한도를 초과할 경우, 트랜잭션의 실행 도중 변화된 상태는 전부 원상태로. 하지만, 이미 사용된 가스 수수료는 반환되지 않고 채굴자가에게 지불.

2.2.4 한번 Deploy된 코드는 바꿀 수 없다

비가역성

- ✓ 한 번 deploy된 코드는 바꿀 수 없음. 코드를 업데이트 해야 하는 요구사항이 있을 경우 업데이트된 Contract의 인스턴스를 deploy한 뒤 기존 인스턴스의 변수를 새 인스턴스로 이전하거나 Proxy 역할을 하는 Contract를 써야 하는 등의 문제가 생김.

- ✓ 한번 배포된 계약은 사업 주체의 통제에서 벗어나기 때문에, 비즈니스 로직의 오작동을 방지하기 위한 대책 필요.

디버깅에 신경쓰기

- ✓ 아직까지 블록체인 개발 생태계가 불충분하기 때문에, 온라인에 충분한 자료가 존재하지 않음.
- ✓ 버전에 따라 코드의 내용이 상이하기에 주의.
- ✓ 솔리디티 자체의 결함에 유의하기.

2.2.4 한번 Deploy된 코드는 바꿀 수 없다

Fail-Safe Mode 매커니즘 개발

✓ 디플로이 하기 전에 일종의 fail-safe(오류 방지) 메커니즘을 포함시켜 향후 발생가능한 심각한 해킹, 버그에 대비하기.

✓ “임의의 이더가 유출 되었습니까?”, “토큰의 합계가 계약의 잔액과 같은가요?” 또는 이와 유사한 것들을 자기 점검으로 수행하는 스마트 계약서에 기능을 추가.

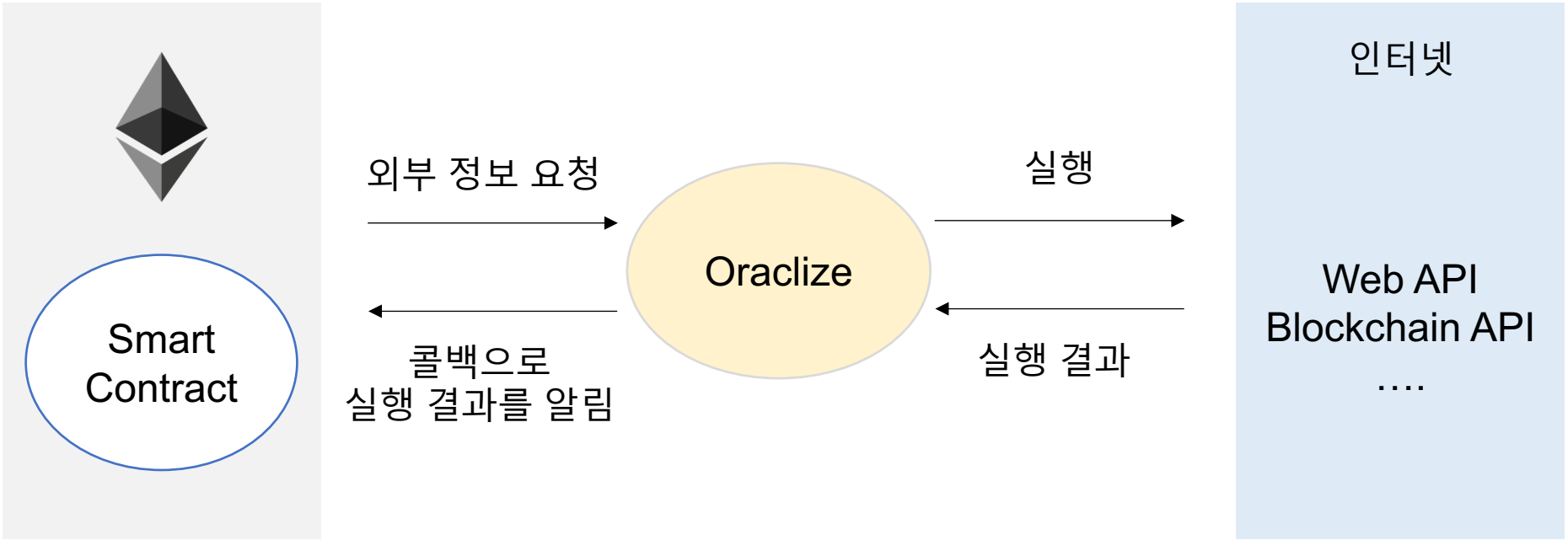
Smart Contract에 저장하는 Token의 양 제한하기

✓ Smart Contract에 저장할 수 있는 토큰의 양을 적절히 제한하기. 소스 코드, 컴파일러 또는 플랫폼에 버그가 있는 경우 이러한 자금이 손실될 수 있기 때문.

2.2.5 외부 데이터 호출에 유의하기

Oraclize 사용

- ✓ 일반적인 경우 smart contract에서 내부 데이터 혹은 다른 smart contract로 부터 받는 데이터 외에 외부 데이터를 사용하거나 HTTP call이 불가능.
- ✓ 이더리움 네트워크 외부와 정보를 연결해주는 서비스를 '오라클'이라고 지칭하며,(loopchain에서는 portal) 대표적으로 Oraclize가 있음.(사실 네트워크에서는 ethereum-bridge)



2.2.5 외부 데이터 호출에 유의하기

Oraclize 사용

- ✓ 파워렛져(Power Ledger)는 전기를 P2P로 판매할 수 있는 전력 거래 플랫폼 사업에 사용하는 암호화폐. 오라클 없다면, 내가 전기를 얼마나 생산했고 판매했는지 알 수 없이 해당 플랫폼이 돌아갈 수 없다.

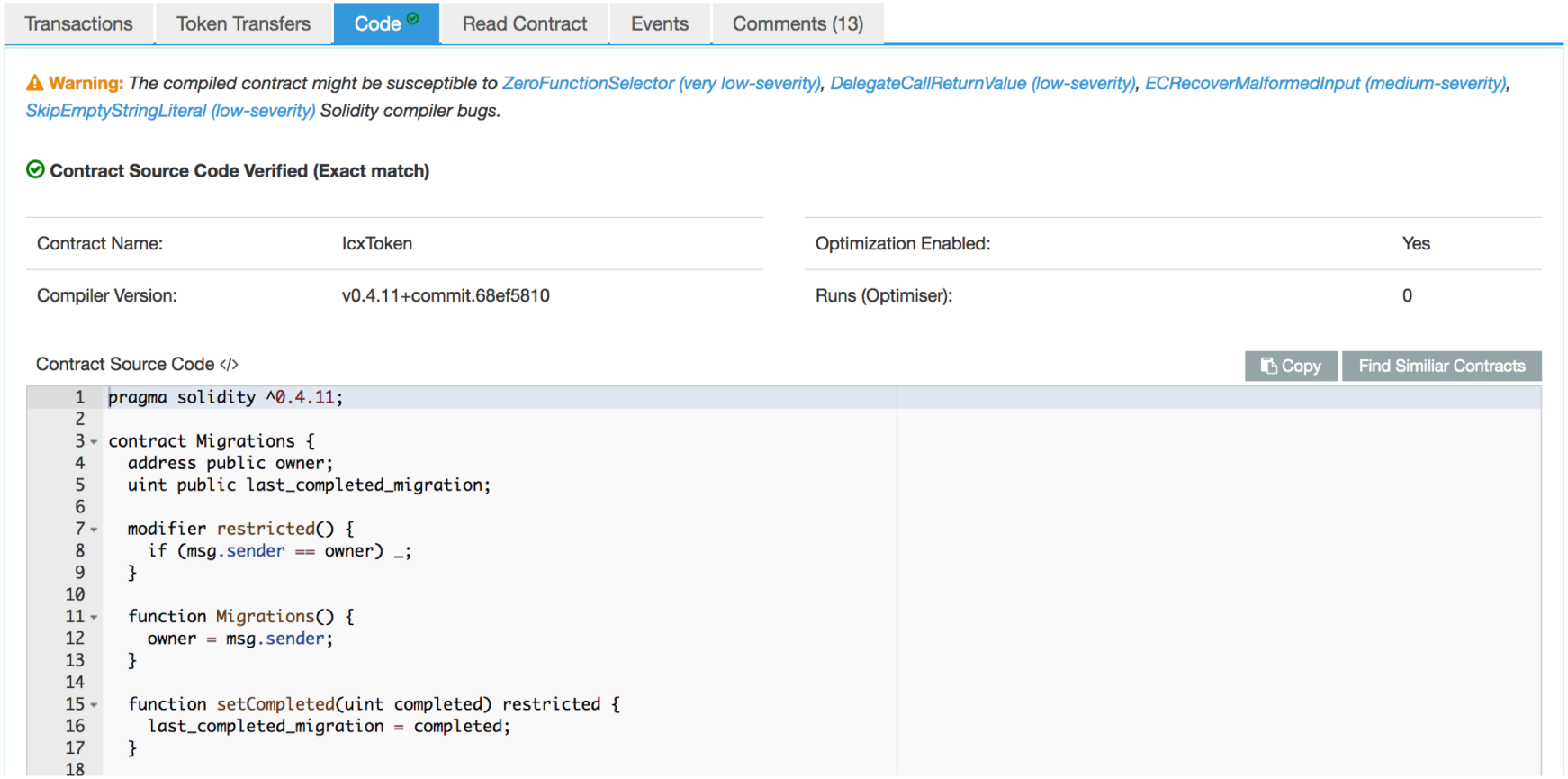
- ✓ 오라클이 필요한 이유는 스마트 컨트랙트를 미들맨(Middle Man) 없이 구현하는데 있어 외부 데이터가 반드시 필요하기 때문.

- ✓ 결국 오라클 없이 만들어질 수 있는 Dapp은 외부 데이터 없이 구현이 가능한 1차원적인 Dapp들 뿐이기에 오라클에 대한 이해는 필수적.

2.2.6 스마트컨트랙트 취약점 주의

코드 투명성으로 인한 취약점 노출

✓ 블록체인은 모두가 데이터를 조회할 수 있는 강점을 갖고 있지만, 누구나 컨트랙트 코드를 조회할 수 있기에 기술 취약점이 노출되기 쉬움.



2.2.6 스마트컨트랙트 취약점 주의

재진입 공격

- ✓ 외부 계약 호출의 실행이 완료되기도 전에 새로운 호출이 허용될 때 발생.
- ✓ 2016년 6월에 'DAO' 플랫폼에 존재하는 취약점을 이용해 360만 개 ETH 해킹 사고 발생.
- ✓ 다오 토큰은 소유자의 결정에 따라 이더로 환전할 수 있으며, 이때 사용된 토큰은 이더로 전환되는 순간 사라지게(burn) 되는데, 이를 '스플릿(split)'이라 함.
- ✓ 해커는 사용된 토큰이 사라지지(burn) 않는 취약점을 이용하여 같은 토큰을 여러 번 사용해 이더를 인출. 여기에는 재귀호출(Recursive call) 사용.

2.2.6 스마트컨트랙트 취약점 주의


```
function splitDAO(
        uint _proposalID,
        address _newCurator
    ) noEther onlyTokenholders returns (bool _success) {

    ...
        // Burn DAO Tokens
        Transfer(msg.sender, 0, balances[msg.sender]);
        withdrawRewardFor(msg.sender); // be nice, and get his rewards
        totalSupply -= balances[msg.sender];
        balances[msg.sender] = 0;
        paidOut[msg.sender] = 0;
        return true;
    }
```

1. splitDAO() 호출.
2. _recipient.call.value(_amount))를 통해 이더 받음.
3. balances[msg.sender] = 0이 호출되어 자신의 잔고가 업데이트되기 전에 다시 splitDAO() 호출.

```
function withdrawRewardFor(address _account) noEther internal returns (bool _success) {
        if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
            throw;

        uint reward =
            (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
        if (!rewardAccount.payOut(_account, reward))
            throw;
        paidOut[_account] += reward;
        return true;
    }
```

```
function payOut(address _recipient, uint _amount) returns (bool) {
         if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
             throw;
         if (_recipient.call.value(_amount)()) {
             PayOut(_recipient, _amount);
             return true;
         } else {
             return false;
         }
     }
```

2.2.6 스마트컨트랙트 취약점 주의

접근제어자 & delegatecall 유의

- ✓ 2017년 7월 이더리움 네트워크 역사상 두번째로 큰 ETH 절도 사건 발생
- ✓ 토큰 세일 때의 펀드를 저장하는 high-profile 멀티시그 컨트랙트에서 153,037 ETH (~330억)도난.
- ✓ 해커는 취약점이 발견된 컨트래트에 두 개의 트랜잭션 전송
    - 멀티시그 지갑의 배타적 소유권 획득을 위한 트랜잭션
    - 컨트랙트의 모든 펀드를 이체하는 트랜잭션

2.2.6 스마트컨트랙트 취약점 주의

접근제어자 & delegatecall 유의

```
1  contract Wallet is WalletEvents {
2    address _walletLibrary;
3    function Wallet(address[] _owners, uint _required, uint _daylimit) {
4      bytes4 sig = bytes4(sha3("initWallet(address[],uint256,uint256)"));
5      address target = _walletLibrary;
6      assembly {
7        delegatecall(sub(gas, 10000), target, 0x0, add(argsize, 0x4), 0x0, 0x0)
8      }
9    }
10   function() payable {
11     if (msg.value > 0)
12       Deposit(msg.sender, msg.value);
13     else if (msg.data.length > 0)
14       _walletLibrary.delegatecall(msg.data);
15   }
16 }
```

* fallback 함수
- 다른 함수에 매칭 되지 않을 때 기본으로 호출되는 함수

* walletLibrary에게 권한을 위임 하여 호출한다.
이때 호출되는 라이브러리의 함수는 msg.data에 의하여 결정된다.

```
1  contract WalletLibrary is WalletEvents {
2    function initDaylimit(uint _limit) {
3      m_dailyLimit = _limit;
4      m_lastDay = today();
5    }
6    function initMultiowned(address[] _owners, uint _required) {
7      ...
8    }
9    function initWallet(address[] _owners, uint _required, uint _daylimit) {
10     initDaylimit(_daylimit);
11     initMultiowned(_owners, _required);
12   }
13 }
```

* 함수 설명
- initWallet : 지갑을 초기화 한다
- initDaylimit : 하루 출금제한 설정
- initMultiowned : 지갑의 권한을 가지는 주인을 설정한다.

2.2.6 스마트컨트랙트 취약점 주의

접근제어자 & delegatecall 유의

✓ 함수를 internal로 바꾸고 컨트렉트의 initialize는 한번만 가능하도록 패치

9 js/src/contracts/snippets/enhanced-wallet.sol — Show comments — View

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104 
105    // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106    // as well as the selection of addresses capable of confirming them.
107 -  function initMultiowned(address[] _owners, uint _required) {
108       m_numOwners = _owners.length + 1;
109       m_owners[1] = uint(msg.sender);
110       m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198    }
199 
200    // constructor - stores initial daily limit and records the present day's index.
201 -  function initDaylimit(uint _limit) {
202       m_dailyLimit = _limit;
203       m_lastDay = today();
204    }
@@ -211,9 +211,12 @@ contract WalletLibrary is WalletEvents {
211       m_spentToday = 0;
212    }
213 
214 +  // throw unless the contract is not yet initialized.
215 +  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104 
105    // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106    // as well as the selection of addresses capable of confirming them.
107 +  function initMultiowned(address[] _owners, uint _required) internal {
108       m_numOwners = _owners.length + 1;
109       m_owners[1] = uint(msg.sender);
110       m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198    }
199 
200    // constructor - stores initial daily limit and records the present day's index.
201 +  function initDaylimit(uint _limit) internal {
202       m_dailyLimit = _limit;
203       m_lastDay = today();
204    }
@@ -211,9 +211,12 @@ contract WalletLibrary is WalletEvents {
211       m_spentToday = 0;
212    }
213 
214 +  // throw unless the contract is not yet initialized.
215 +  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

2.2.7 Checks-Effects-Interactions Pattern 사용

검사 우선의 패턴

✓ 검사(함수를 호출한 사람, 범위 내의 인수, 충분한 Ether를 보냈는지, 토큰의 유무 등)를 먼저 수행하고 다른 계약과 상호작용하는 패턴으로 코드를 짜기.

✓ 모든 검사들이 통과되면 현재 계약의 상태 변수에 영향을 주어야 하며, 다른 계약과의 상호 작용은 항상 모든 기능의 마지막 단계여야 함.

```
contract Fund {
    /// 콘트랙트의 이더 지분 정보 매핑
    mapping(address => uint) shares;

    /// 지분 인출
    function withdraw() {
        if (msg.sender.send(shares[msg.sender])
            shares[msg.sender] = 0;
    }
```

```
contract Fund {
    /// 콘트랙트의 이더 지분을 매핑
    mapping(address => uint) shares;

    /// 지분을 인출
    function withdraw() {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        if (!msg.sender.send(share))
            throw;
    }
```

2.2.8 기타 유의해야 할 특성들

랜덤 값 생성 불가

✓ 컨트랙트 안에서 랜덤 값을 생성하거나, 실행하는 모델은 불가하나, 블록의 해시 혹은 트랜잭션을 이용한 랜덤 값 이용은 가능.

부동소수점 처리 불가

✓ CPU에 따라 부동소수점표현방식이 달라질 수 있으므로 모든 연산은 정수 단위에서 처리해야 함.

시간에 따라 행동하는 혹은 실행 시간에 따라 내용이 바뀌는 모델 구현 어려움

✓ 현재 시간(실행시간)은 사용 불가능하며, 블록의 시간 혹은 트랜잭션 시간으로 대체 가능.

2.3 피치를 할 때

2.3 더 좋은 피치가 되기 위해서…

✓ 중앙화에 대한 이유 제시

- 온체인 요소와 오프체인 요소가 적절히 구분되어 있어야 좋은 Dapp이 될 수 있음.

✓ 실제 트랜잭션 보여주기

- 컨트랙트에 의해 실제로 트랜잭션이 발생됨을 증명하기.