

자바스크립트 개발자를 위한

이더리움 NFT

자바스크립트를 알고 있지만 블록체인(이더리움)을 모르는 개발자들을 위한 Dapp 개발 입문서

목차

CHAPTER 1 이더리움(Ethereum)

- 1.1. 퍼블릭 블록체인
- 1.2. 이더리움의 동작원리
 - 1.2.1. 작업증명
 - 1.2.2. 블록
 - 1.2.3. 계정과 상태트리
 - 1.2.4. 트랜잭션과 가스
 - 1.2.5. 블록 가스 제한(Block Gas Limit)
 - 1.2.6. EVM(Ethereum Virtual Machine)
 - 1.2.7. 네트워크 분기와 하드포크
 - 1.2.8. 이더리움 접속 방법
- 1.3. 지분증명
 - 1.3.1. 단계별 전환
 - 1.3.2. 작업증명 종료와 머지(The Merge)
- 1.4. 정리

CHAPTER 2 솔리디티(Solidity)

- 2.1. 스마트 컨트랙트
- 2.2. 리믹스(Remix) 사용법
- 2.3. 컨트랙트 구조

- 2.4. 데이터 탑입
- 2.5. constant와 immutable
- 2.6. 제어문
- 2.7. 함수
- 2.8. receive와 fallback
- 2.9. modifier
- 2.10. assert, require, revert
- 2.11. 예외처리(try-catch)
- 2.12. 이벤트(event)
- 2.13. 상속과 인터페이스
- 2.14. 기타
- 2.15. 보안 코딩
 - 2.15.1. 컨트랙트 중단 함수
 - 2.15.2. 권한과 역할
 - 2.15.3. 인출(Withdrawal) 패턴
 - 2.15.4. 재진입(Reentrancy) 문제
 - 2.15.5. 대리자(proxy) 컨트랙트
 - 2.15.6. tx.origin과 msg.sender
- 2.16. 정리

CHAPTER 3 NFT(Non-Fungible Token) 표준

- 3.1. 이더리움의 표준
- 3.2. 토큰 표준
- 3.3. ERC-721
 - 3.3.1. 기본 인터페이스
 - 3.3.2. 확장 인터페이스

3.4. ERC-1155

3.4.1. 기본 인터페이스

3.4.2. ERC1155TokenReceiver

3.4.3. ERC1155Metadata_URI

3.5. NFT 유형

3.5.1. 디지털 수집품

3.5.2. 디지털 아트

3.5.3. 제너레이티브 NFT

3.5.4. 유틸리티 NFT

3.5.5. 게임과 메타버스

3.5.6. 1차 시장과 2차 시장

3.6. 정리

CHAPTER 4 오픈제펠린(OpenZeppelin)

4.1. 컨트랙트

4.2. 접근 제어(Access Control)

4.2.1. Ownable

4.2.2. 역할(Role) 기반의 접근제어

4.3. ERC721

4.4. ERC1155

4.5. 보안

4.5.1. Pull Payments

4.5.2. Reentrancy Guard

4.6. 정리

CHAPTER 5 Dapp 개발환경

5.1. 탈중앙화 애플리케이션(Dapp)

5.2. Dapp 개발요소

5.2.1. 솔리디티

5.2.2. 자바스크립트 기반 개발도구

5.3. 트러플(Truffle)

5.3.1. 트러플 설치

5.3.2. 기본 구조

5.3.3. 기본 설정

5.3.4. 컴파일

5.3.5. 배포(deploy)

5.3.6. 단위 테스트

5.3.7. 디버깅

5.4. 하드햇(Hardhat)

5.4.1. 하드햇 설치

5.4.2. 기본 설정

5.4.3. 컴파일과 배포

5.4.4. 단위 테스트

5.4.5. 플러그인

5.5. 정리

CHAPTER 6 Simple NFT 프로젝트

6.1. 스마트 컨트랙트와 애플리케이션

6.2. 시나리오

6.3. 이미지 준비

6.3.1. 이미지 조합

6.3.2. NFT의 속성

6.3.3. 이미지 생성

6.3.4. 메타정보 업로드

6.4. 스마트 컨트랙트

6.4.1. ERC-721 컨트랙트

6.4.2. 단위 테스트

6.5. 리액트 애플리케이션

6.5.1. 상단 메뉴

6.5.2. 지갑 연결

6.5.3. NFT 메타정보

6.5.4. NFT 목록과 페이지 분할

6.5.5. 스마트 컨트랙트 호출

6.6. OpenSea 게시하기

6.6.1. opensea.js

6.7. NFT 발행 방식

6.8. 정리

실습환경

이 문서가 작성되는 시점에 사용된 환경입니다. 기본적으로 Window 10/11을 기준으로 예제들이 실행되었습니다. 당연히 시점 차이 때문에 어느정도 트러블슈팅이 필요할 수도 있겠지만 크게 영향을 받는 부분은 없을 것 같습니다.

Node.js 16.13.0	https://nodejs.org/ko/
React.js 17.0.2	https://reactjs.org/
Solidity 0.8.4	https://docs.soliditylang.org
Truffle 5.4.19	https://www.trufflesuite.com/
Hardhat 2.6.8	https://hardhat.org/
Ethers.js 5.5.1	https://docs.ethers.io/v5/
Openzeppelin	https://docs.openzeppelin.com/openzeppelin/
Alchemy	https://www.alchemy.com/
NFT Storage	https://nft.storage/

예제

예제에 사용된 코드는 깃허브를 참조하기 바랍니다.

<https://github.com/boyd-dev/alice-NFT>

질문과 제안

내용에 관한 문의나 개선 제안은 아래 채널을 이용하기 바랍니다.

<https://gitter.im/LearningHelper/community>

boyd-dev@nate.com

swkim109@gmail.com

버전

2022-0.3.1

라이선스

✓ MIT License

Copyright (c) 2022 boyd-dev@nate.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

✓ CC BY-SA 3.0

이더리움(Ethereum)



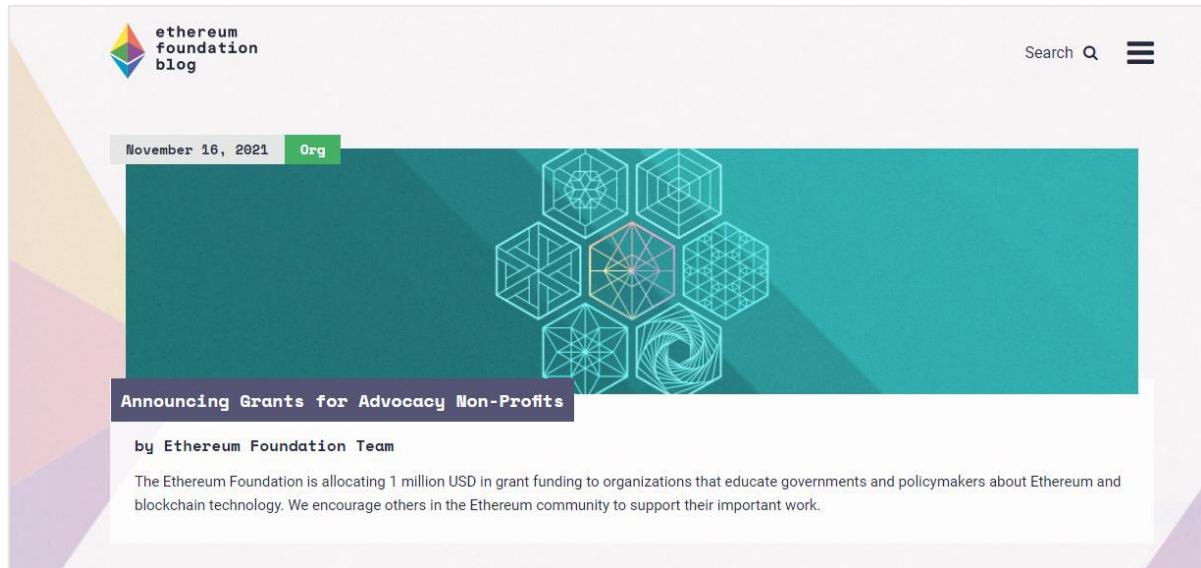
1장에서는 블록체인과 이더리움에 관한 일반적인 사항들을 알아봅니다. 이더리움의 개념과 동작 원리를 가능하면 단순하게, 포괄적으로 설명합니다.

1.1. 퍼블릭 블록체인

블록체인은 국가 또는 몇몇 글로벌 기술 기업들이 주도하는 중앙집권적인 체제와는 다르게 불특정 다수의 노드들이 프로토콜과 알고리즘, 암호학을 기반으로 서로 연결되어 있는 P2P(Peer-to-Peer) 네트워크입니다. 비트코인에서 시작된 블록체인은 기존의 중앙화된 시스템들의 지배구조, 의사결정구조, 경제구조 등을 포함하는 사회 전반의 패러다임을 변화시키고 있습니다.

물론 추구하는 이상과 현실의 차이는 존재합니다. 탈중앙화가 내세우는 가치에 맞지 않게 소수의 기업형 채굴자들, 무리한 채굴로 인한 에너지 자원의 낭비, 익명성을 악용하는 해킹, 불투명한 거버넌스, 중앙정부의 규제 등의 문제는 여전히 해결해야 할 과제입니다. 하지만 블록체인은 수많은 난제들을 하나씩 풀어 나가면서 계속 진화하고 있습니다.

가장 활발하게 움직이고 있는 퍼블릭 블록체인들 중 하나는 이더리움(Ethereum)입니다. 이더리움은 당시 19세였던 비탈릭 부테린(Vitalik Buterin)이 블록체인을 “애플리케이션 플랫폼”으로 활용하자는 제안으로 시작되었고 드디어 2015년 7월, “Frontier”를 공개했습니다. 5년이 지난 지금, 믿거나 말거나 4400억 달러를 넘는 가치를 형성하면서 이더리움의 생태계는 점점 확장하고 있습니다.



이더리움은 아직 끝나지 않은 글로벌 프로젝트입니다. 이더리움 재단은 비영리 재단으로 프로젝트를 관리하면서, 동시에 사회의 각 분야에서 블록체인의 가치를 실현하기 위해 노력하고 있습니다. 이더리움 블로그를 방문하면 다양한 소식들을 접할 수 있을 것입니다.

<https://blog.ethereum.org>

1.2. 이더리움의 동작 원리

이더리움은 불특정 다수가 자발적으로 참여하는 P2P 네트워크입니다. 현재 이더리움 네트워크를 이루는 노드의 개수는 정확히 파악할 수는 없지만 대략 수 천대 규모로 추산하고 있습니다. 통계로 잡히는 노드 분포는 아래 사이트에서 확인할 수 있습니다.

<https://ethernodes.org/>

이더리움이 비트코인과 가장 크게 다른 점은 바로 “튜링 완전한” 컴퓨터 프로그래밍이 가능하다는 것입니다. 블록체인에서 실행되는 프로그램을 “스마트 컨트랙트(Smart Contract)”라고 부르는데 바로 이더리움은 스마트 컨트랙트를 실행할 수 있는 최초의 퍼블릭 블록체인으로 시작되었습니다.

비트코인이 단순히 화폐의 기능만을 제공한다면 이더리움은 스마트 컨트랙트를 이용하여 다양한 애플리케이션 플랫폼의 역할을 할 수 있습니다. “Programmable Money”가 이더리움을 가장 잘 표현하는 말이 아닐까 싶습니다.

스마트 컨트랙트 덕분에 다양한 유무형의 가치들이 토큰화 되면서 중앙 기관의 개입이 없어도 탈중

앙화 금융(DeFi)과 같은 서비스들이 가능하게 된 것입니다. 그렇다면 한 번도 만난 적이 없고 신원을 알 수 없는 불특정 다수가 참여함에도 불구하고 엄청난 금액의 자금이 모이고 거래가 이루어질 수 있는 이유는 무엇일까요?

이제부터 이더리움의 기본 원리를 살펴보도록 하겠습니다.

1.2.1. 작업증명

이더리움은 다양한 프로그래밍 언어로 구현된 이더리움 프로토콜 구현체, 즉 클라이언트 소프트웨어가 서로 P2P 방식으로 연결되면서 시작됩니다. 이들 클라이언트를 실행하는 주체는 불특정 다수입니다. 누구나 소프트웨어를 다운로드하여 실행하고 네트워크에 접속할 수 있습니다.

불특정 다수가 참여하게 되면 누군가 나쁜 의도로 거래를 조작하거나 상대방을 속이고 자산을 훔칠 가능성은 없을까요? 블록체인에서는 알고리즘과 암호학 기술로 그것을 방지합니다(물론 프로그램의 버그나 암호 유출로 인한 보안 사고는 언제든지 발생할 수 있습니다).

이더리움에서 이루어지는 모든 거래들은 특별한 임무를 수행하는 자발적인 참여자에 의해 처리됩니다. 스스로 참여하는 이유는 보상을 받기 때문입니다. 이를 "채굴자(miner)"라고 부릅니다. 급여 생활자들은 어떤 "채굴"을 할까요? 그렇습니다. 한국은행이 발행한 원화(KRW)를 채굴하는 것입니다. 이더리움 채굴자들의 참여 유인은 사람들이 일을 하려는 동기와 본질적으로 같습니다. 이더리움에서는 그 보상으로 "이더(ETH)"라는 가상자산이 지급됩니다.

보상을 받기 위해서는 규칙을 따라야 합니다. 그 규칙이 이더리움 프로토콜입니다. 알고리즘과 암호학 기술로 구현된 프로토콜은 규칙이고 약속이며 그것을 지키지 않으면 네트워크에서 소외되고 블록체인 데이터를 동기화할 수 없게 됩니다. 채굴자들은 자신의 컴퓨팅 자원(해시 파워)을 사용하여 다른 사람들이 전송한 트랜잭션들을 "블록"이라는 자료구조에 담아서 네트워크에 전파합니다.

다른 참여자들이 블록을 검증하고 이상이 없으면 자신의 거래 "원장(元帳)", 즉 블록체인에 추가합니다. 결국 모든 사람들이 정해진 프로토콜과 알고리즘으로 계산을 한다면 같은 결과를 얻게 되고 내가 가진 원장은 다른 모든 사람들이 가진 원장과 정확히 일치하게 되는 것입니다. 이러한 이유로 블록체인을 "분산 원장"이라고 부르기도 합니다.

채굴자들이 자발적으로 참여하기 때문에 누구나 블록을 만들 수 있습니다. 그렇다면 누가 만든 블록을 선택해야 할까요? 이더리움은 게임의 규칙을 미리 정했습니다. 가장 먼저 만든 사람에게 보상을 주는 것입니다. 블록체인에서는 이러한 규칙을 합의 방식이라고 합니다.

이더리움은 현재 "작업증명(Proof of Work, PoW)"이라는 합의 방식을 채택하고 있습니다. "작업"이라

는 것은 컴퓨팅 자원을 소모하면서 유효한 블록을 만들기 위해 들인 수고를 의미하는 것입니다. 실제로는 복잡하지만 단순하게 표현하면 대략 이런 방식입니다.

$$Ethash(X) \leq \text{Difficulty target}$$

"이더해시, Ethash"는 이더리움의 작업증명 알고리즘의 이름입니다. 물론 "Ethash(X)"라는 해시 함수는 없지만 이 알고리즘을 사용하여 Difficulty target이라는 숫자보다 작거나 같은 X를 찾는 일이 바로 채굴자들이 하는 일입니다.

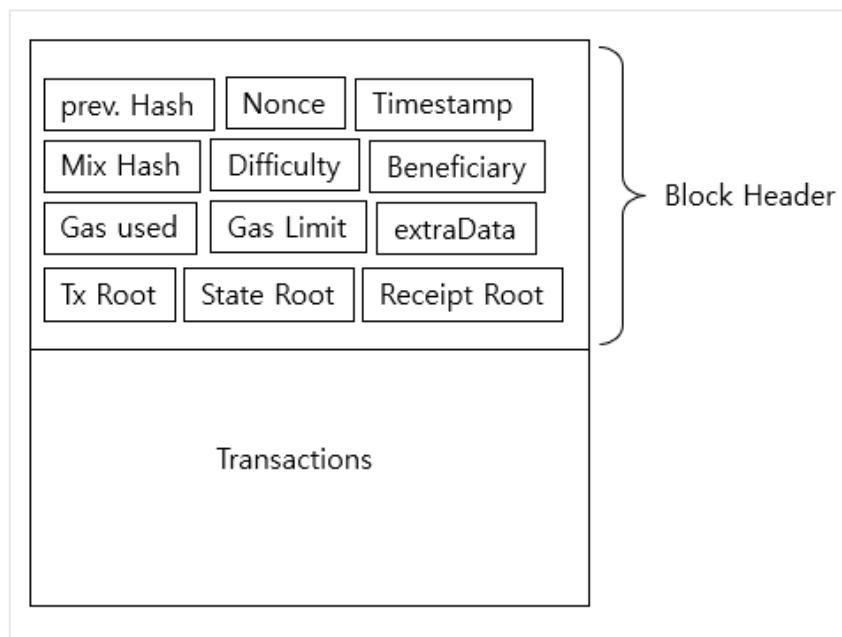
역상(preimage) 저항성이라는 특성으로 인하여 X를 구하는 해법이 없기 때문에 계속 여러 값을 차례로 넣어보고 조건에 맞는지 검사해보는 수 밖에는 없습니다(Brute force 알고리즘). X를 구해야 모든 클라이언트에게 인정받는 블록을 만들 수 있는 것입니다. 일단 찾기만 하면 다른 노드들이 그것을 검증하는 것은 어렵지 않습니다.

Difficulty target은 굉장히 큰 범위에 속한 어떤 숫자입니다. "Difficulty"라고 표현한 것은 이 값이 작을수록 X를 찾기 어렵기 때문입니다. 예를 들어 안을 들여다볼 수 없는 상자에 1부터 100까지 숫자가 적힌 공을 넣고 90보다 작은 수를 찾는 것과 5보다 작은 수를 찾는 것 중 어느 쪽이 어려운 일인지 생각해보면 되겠습니다.

1.2.2. 블록

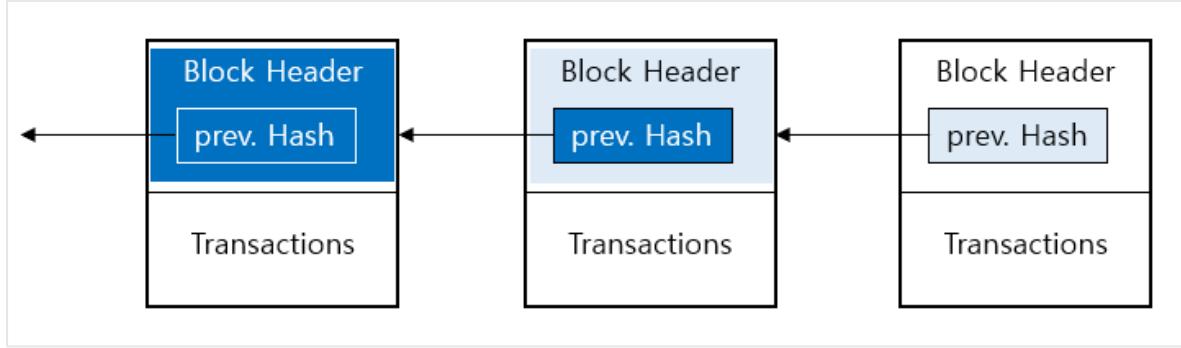
채굴자들은 네트워크에 전파되는 거래 메시지들을 모아서 블록이라는 자료구조를 만듭니다. 앞에서 Ethash라는 알고리즘에 의해 특정 조건을 만족하는 값을 찾는다고 했는데 그렇게 계산된 값을 바로 블록의 식별자로 사용합니다. 이 값을 “블록해시”라고 합니다. 예를 들어 블록번호 #13681814번의 블록해시는 0x8565fbbdfc79208f5c7523b58f3b391f3a08d8bccbb02f5b59dbe9d5124d42c으로 표시됩니다.

블록의 구조를 간단하게 나타내면 블록 헤더와 트랜잭션으로 구분할 수 있습니다. 블록 헤더에는 블록 내의 트랜잭션을 검증할 수 있는 정보들, 블록의 용량, 보상계정 등의 값이 저장됩니다. 블록해시는 블록 헤더에 있는 값들을 해시하여 일정한 길이 256비트의 값이 됩니다.



그런데 블록해시를 계산할 때 필요한 값들 중 하나는 이전 블록의 블록해시입니다. 이렇게 모든 블록들은 바로 직전 블록의 블록해시와 연결되어 있습니다. 채굴자들은 블록헤더의 각 항목들과 “Nonce”的 값을 변경해가면서 Ethash 알고리즘에 의해 블록해시 값을 계산하는 것입니다.

해시 함수의 특성 때문에 어떤 블록의 블록해시를 변경하면 그것을 참조하는 모든 블록들의 블록해시를 순차적으로 바꾸어야 하고, 또 그렇게 되면 작업증명을 다시 수행해야 하므로 유효한 블록으로 인정받기 힘들어지는 것입니다. 이와 같이 블록체인의 데이터를 변경하는 것이 매우 어렵기 때문에 블록체인을 위변조하는 것은 거의 불가능한 일입니다.

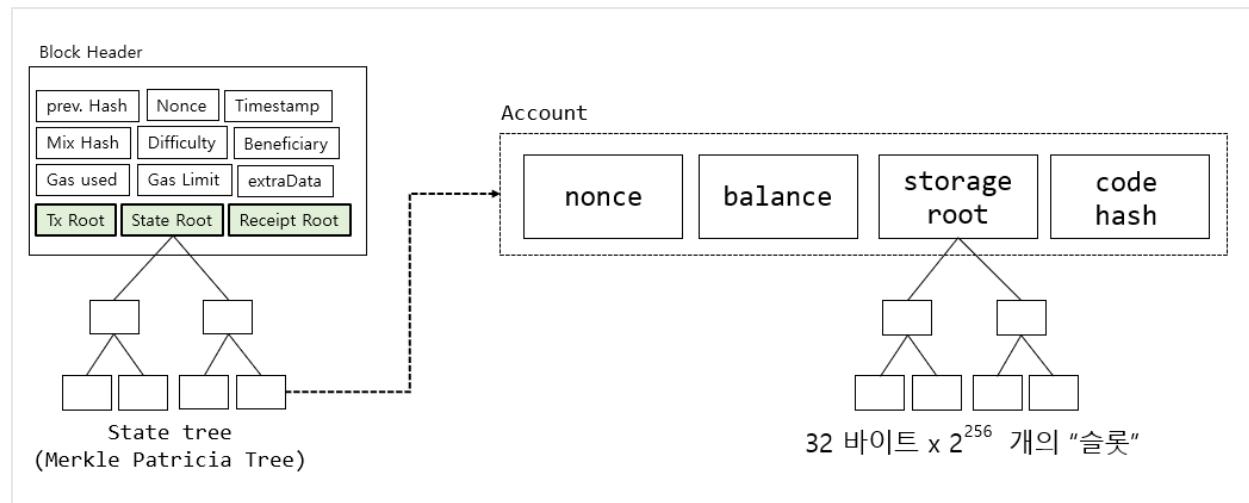


1.2.3. 계정과 상태트리

이더리움을 사용하려면 계정(Account)을 만들어야 합니다. 계정은 은행의 계좌와 비슷하다고 생각할 수 있습니다. 계좌는 은행에서 발급하지만 탈중앙화 블록체인에서는 개인 스스로 자신의 계좌를 만듭니다. 이것이 가능한 이유는 타원곡선 암호학 덕분입니다.

타원곡선 암호를 이용하여 비대칭 키 페어(key pair)를 만들면 개인키는 노출시키지 않으면서도 공개 키를 사용하여 고유한 계정을 생성할 수 있습니다. 계정은 잔액정보, 컨트랙트의 데이터(Storage)와 코드 해시(Code hash)를 저장할 수 있는 구조입니다.

계정은 사람이 소유하는 계정 EOA(Externally Owned Account)가 될 수도 있고 컨트랙트 계정(Contract Account)이 될 수도 있습니다. 계정들 사이에서는, 그것이 EOA이든 CA든 상관없이 서로 자산이나 데이터를 주고받는 것이 가능합니다.



컨트랙트는 32바이트 크기의 슬롯 단위로 데이터를 저장합니다. 슬롯은 최대 2^{256} 개까지 가능하므로 저장공간을 거의 무한대까지 늘릴 수 있습니다. 계정은 사람만 가질 수 있는 것이 아니라 스마트 컨

트랙트에게도 계정이 부여됩니다.

이더리움은 이러한 수많은 계정의 상태가 저장된 분산 데이터베이스라고 볼 수 있습니다. 계정의 상태는 “머클 파트리샤 트리(MPT)”라는 트리 구조로 저장되는데 이것을 “상태 트리”라고 합니다. 상태 트리를 만들기 위한 데이터들은 블록체인에 저장되지만 상태 트리 자체는 각 노드의 로컬 데이터베이스에 저장됩니다. 상태 트리의 정합성은 블록헤더 내에 상태 루트를 해시를 만들어서 보장합니다. 상태 트리의 값을 변경하면 상태 루트의 값도 변하게 되고 그렇게 되면 결국 블록해시를 다시 계산해야 합니다.

1.2.4. 트랜잭션과 가스

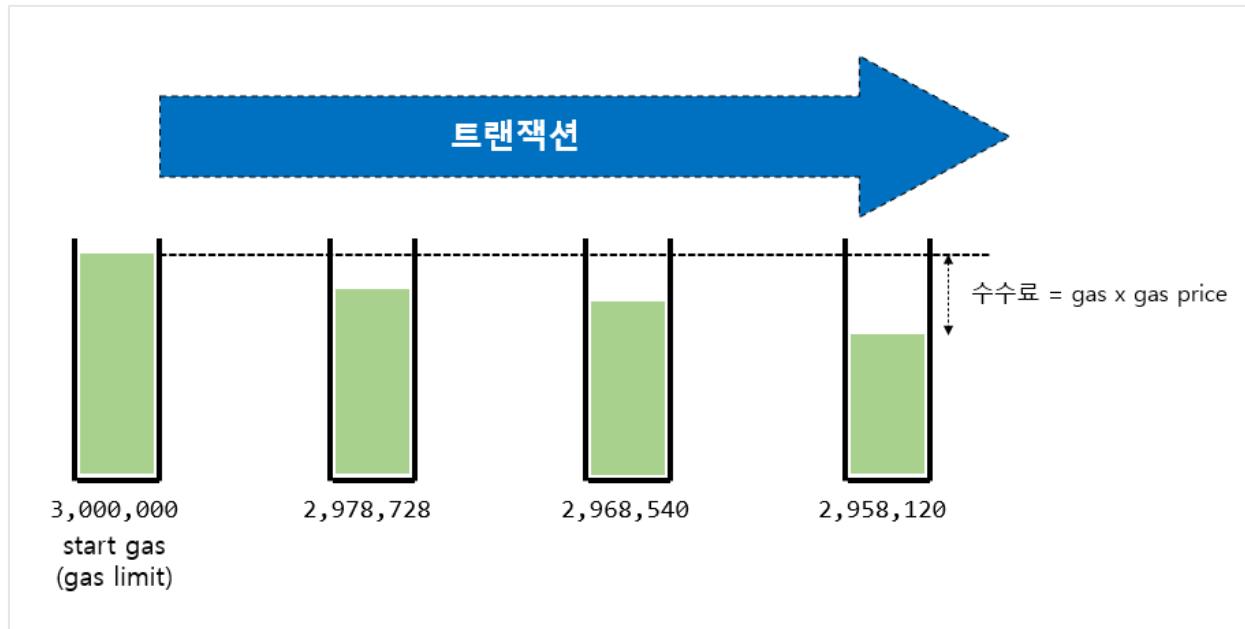
흔히 트랜잭션이라고 하면 데이터를 변경시키는 요청을 말합니다. 관계형데이터베이스의 INSERT나 UPDATE 문에 해당합니다. 트랜잭션은 이더리움 계정의 상태를 변경하는 메시지 호출입니다. 송금을하거나 스마트 컨트랙트의 함수를 호출하여 컨트랙트에 저장된 값을 바꾸는 것들이 모두 트랜잭션이라고 볼 수 있습니다.

채굴자들이 하는 일이 트랜잭션을 처리하는 일입니다. 네트워크에서 발생하는 트랜잭션들을 수집하여 블록을 만들고 전파하고 그에 따라 보상을 받게 됩니다.

이더리움은 누구나 접근 가능한 네트워크이기 때문에 스팸 트랜잭션에 취약합니다. 그래서 트랜잭션 비용을 높이기 위해 “가스(gas)”라는 개념을 도입했습니다. 가스는 실제 천연가스 같은 것을 말하는 것이 아니라 그냥 숫자로 표시되는 어떤 값입니다.

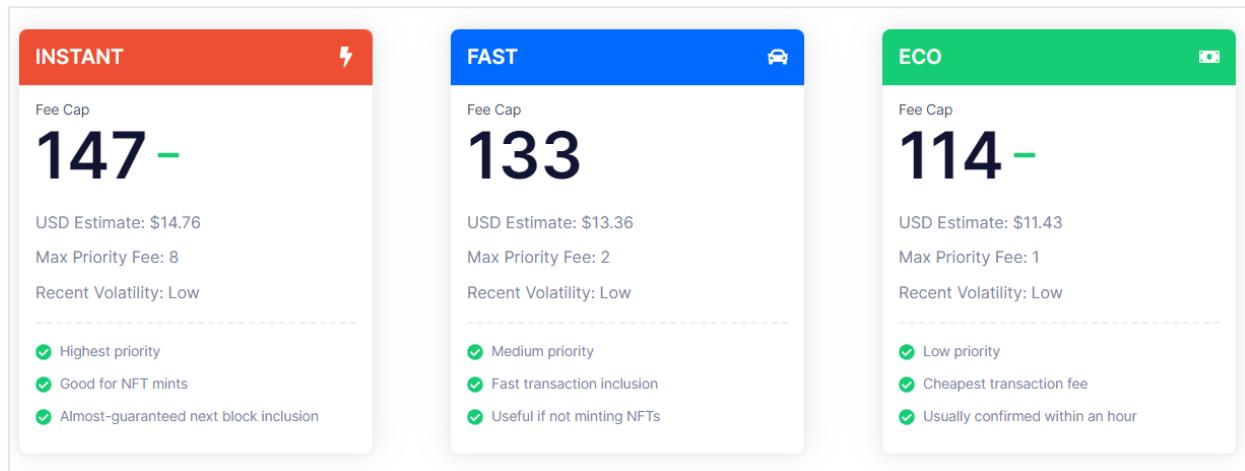
이더리움에서는 트랜잭션을 실행하는 모든 명령어(opcode)에 가스 값을 책정합니다. 예를 들어 송금은 21,000 가스가 필요합니다. 명령어가 실행될 때마다 처음 설정한 가스(gas limit)를 소모하면서 점점 줄어들게 됩니다. 그래서 트랜잭션을 보낼 때는 충분한 가스가 필요하고 또 이것을 보내는 사람은 1 가스에 얼마의 비용(ETH)을 지불할지 결정해야 합니다. 이것이 흔히 말하는 “가스비”라고 하는 수수료입니다. 수수료는 거래를 처리해주는 채굴자에게 돌아갑니다.

트랜잭션 수수료는 트랜잭션에 전자서명을 하는 계정이 부담해야 합니다. 이렇게 비용이 들기 때문에 공격자는 함부로 무리한 트랜잭션을 보낼 수 없습니다. 또 컨트랙트의 잘못된 코드가 무한 루프에 빠지는 일도 없습니다. 가스가 소모되어 0이 되면 실행은 중지될 것입니다. 이더리움은 가스를 통해 “공유지의 비극”을 방지합니다.



가스의 도입으로 인해 단점도 생깁니다. 어떤 트랜잭션을 처리하는데 들어가는 가스는 정해져 있고 가스당 지불하는 수수료는 사용자가 결정합니다. 그런데 채굴자들은 수수료가 높은 거래를 먼저 블록에 저장하므로 사용자들끼리 수수료 경쟁이 일어날 수 있습니다. 다음 블록에 즉시 저장되어야 이익인 트랜잭션은 높은 수수료를 책정하여 전송하기 때문에 수수료 경쟁이 일어나게 되고 수요가 많을 때 수수료는 그야말로 천정부지로 폭등할 수 있습니다. 높은 가스비에 대한 부담은 결국 사용자들의 불만으로 이어지게 됩니다. 아래 사이트에서 가스비 수준을 확인할 수 있습니다.

<https://www.gasprice.io/>



현재 이더리움은 트랜잭션의 처리 수요에 따라 기본 수수료(Base fee)를 자동으로 결정하고 거기에 사용자가 얼마의 추가 비용, 즉 팁(Priority fee)을 더해서 트랜잭션을 전송하게 됩니다. 기본 수수료는 채굴자에게 지급되지 않고 내부적으로 없어지게 됩니다(수수료 소각이라고 표현합니다). 이것은 이더의 무제한 발행으로 인한 인플레이션을 조절하는 작용을 합니다. 트랜잭션이 증가할수록 기본 수수료가 오르게 되고 반대로 트랜잭션의 수요가 줄면 기본 수수료는 내려갑니다.

1.2.5. 블록 가스제한(Block Gas Limit)

하나의 블록에 저장할 수 있는 트랜잭션은 제한이 있습니다. 많은 트랜잭션을 담을수록 좋을 것 같지만 블록의 크기가 커지면 네트워크를 구성하는 수 천대의 노드들에게 시간 내에 전파되기 어려울 수도 있습니다. 그래서 이더리움에는 “블록 가스제한(Block Gas Limit)”이라는 것이 존재합니다.

블록에 담을 수 있는 트랜잭션의 수를 제한하는 대신 최대 가스량을 정하여 제한을 두는 것입니다. 각 트랜잭션들은 실행에 필요한 가스량을 가지고 있고 그것을 합산했을 때 블록 가스제한 이내가 되도록 블록을 만드는 것입니다. 현재 블록 용량은 가변적으로 변하는데 최대 3000만 가스입니다. 평상시에는 1500만을 목표로 유지하지만 트랜잭션이 증가하면 3000만까지 늘릴 수 있습니다.

예를 들어 이더스캔에서 13700002번 블록(<https://etherscan.io/block/13700002>)을 조회하면 블록의 최대 가스량은 3000만이고 실제 블록에 담긴 트랜잭션들의 가스는 모두 21,844,012로 약 72.81%를 점유하고 있습니다. 그리고 목표 가스량 1500만 가스를 +46% 초과했음을 나타내고 있습니다.

② Size:	185,486 bytes
② Gas Used:	21,844,012 (72.81%)  +46% Gas Target
② Gas Limit:	30,000,000

1.2.6. EVM(Ethereum Virtual Machine)

이더리움은 스마트 컨트랙트가 실행되는 블록체인입니다. 스마트 컨트랙트는 EVM(Ethereum Virutal Machine)이라고 하는 가상머신에서 실행합니다. 자바의 JVM과 유사하다고 생각하면 되겠습니다.

다른 블록체인에서도 EVM을 구현하면 이더리움 컨트랙트가 실행될 수 있습니다. 이더리움에 배포된 컨트랙트를 활용하기 위해 다른 블록체인에서도 EVM을 포팅하려는 움직임들이 있고, 또 요즘 주목 받는 레이어2 기술, 예를 들면 “Optimistic Rollup” 솔루션들은 각자 EVM과 호환되는 실행 엔진을 구

현하고 있습니다.

이더리움의 스마트 컨트랙트 프로그래밍 언어는 솔리디티(Solidity)입니다. 파이썬 기반의 바이퍼(Vyper)라는 언어도 있지만 대부분 솔리디티로 작성됩니다.

솔리디티로 작성된 소스코드를 컴파일하면 바이트 코드로 변환되고 그 바이트 코드를 이더리움에 전송합니다. 컨트랙트 생성 역시 트랜잭션이기 때문에 가스가 필요하고 수수료를 내야 합니다. 컨트랙트 주소는 이더리움 내에서 유일하게 식별됩니다.

사용자 애플리케이션이 이더리움에 배포된 컨트랙트의 함수를 호출하면 노드는 EVM으로 해당 함수를 실행합니다. 컨트랙트의 상태를 변경하는 트랜잭션은 채굴자에 의해 블록에 저장이 되고 네트워크에 전파됩니다. 유효한 블록을 받은 노드들은 각자 상태 트리를 업데이트하게 됩니다.



EVM은 내부적으로 바이트코드로 변환된 명령어들, 즉 opcode들을 실행하는 것입니다. 앞서 설명한 것처럼 이들 opcode들은 각각 자신들이 실행될 때 소비하는 가스 값이 책정되어 있습니다. EVM에 정의된 opcode에 대한 정보는 아래 링크를 참고하기 바랍니다.

<https://www.evm.codes/>

1.2.7. 네트워크 분기와 하드포크

어떤 경우는 다수의 채굴자들이 여러 개의 유효한 블록들을 만들 가능성이 있습니다. 왜냐하면 네트워크 사정이 나빠져서 블록의 전파가 지연되는 경우 다른 채굴자가 만든 블록의 존재를 인지하지 못하고 자신의 블록을 전파할 수 있기 때문입니다.

이렇게 일시적으로 블록체인이 하나 이상의 버전으로 존재하는 것을 “분기(Split)”라고 합니다. 다시 말해서 분산 원장이 두 가지 버전이 존재하게 되는 상황입니다. 분기 현상은 수 천대 규모의 컴퓨터들이 연결된 블록체인에서는 당연히 발생할 수 있는 일입니다.

이더리움에서는 이러한 분기 현상이 발생하면 누적된 작업증명이 많은 체인을 “정본(canonical)” 체인으로 결정하는 “GHOST(Greedy Heaviest Object subTree)”라는 규칙을 가지고 있습니다. 이를 위해서 다른 채굴자들이 만든 블록(언클 블록, Uncle block)의 정보도 포함시켜서 더 “무거운” 체인을 메인 체인으로 판단합니다. 또 언클 블록을 만든 채굴자들에게도 일정 비율의 보상을 지급함으로써 더 많은 참여자들을 네트워크에 유인할 수 있게 됩니다.

분기 현상은 그렇게 자주 일어나는 것이 아니고, 발생하더라도 GHOST에 의해 각 노드들이 단일 체인을 결정할 수 있으므로 블록체인이 계속 유지될 수 있게 됩니다.

분기와 비슷하지만 전혀 다른 개념으로 “하드포크(Hard Fork)” 있습니다. 일반적으로 하드포크는 클라이언트 소프트웨어의 업그레이드를 의미합니다. 클라이언트는 정해진 로드맵에 따라 새로운 기능이 추가되기도 하고 보안 패치가 적용되기도 합니다. 이 때마다 호환성의 문제가 발생할 수 있는데, 이렇게 되면 업그레이드한 노드와 그렇지 않은 노드들이 서로의 블록을 유효하지 않은 것으로 판단하여 네트워크가 분리될 수 있습니다.

정상적인 경우 대다수의 노드들이 새로운 버전의 클라이언트로 점차 업그레이드하면 하드포크는 성공이지만 업그레이드에 찬성하지 않는 쪽이 이전 버전을 그대로 고수한다면 네트워크의 분기는 사실상 영원히 지속될 수밖에 없습니다. 이 때는 새로운 블록체인, 즉 새로운 코인이 생기는 결과로 이어지게 됩니다. 이더리움과 이더리움 클래식(ETC)이 그런 예가 되겠습니다.

1.2.8. 이더리움 접속 방법

이더리움에 연결하기 위해서는 크게 두 가지 방법이 있습니다.

- ✓ 직접 노드를 운영(클라우드 서비스 포함)
- ✓ 이더리움 노드를 제공하는 서비스 이용

이더리움 클라이언트는 여러 가지 프로그래밍 언어로 개발된 구현체들이 존재합니다. 주요 클라이언트는 다음과 같습니다.

클라이언트	개발 언어
Geth(Go-Ethereum)	Go
OpenEthereum(Parity)	Rust
Hyperledger Besu	Java
Nethermind	C#
Erigon(Turbo-geth)	Go

클라이언트를 처음 실행하면 그동안 쌓인 블록체인 데이터들을 받고 검증하고 데이터베이스를 생성하는데 비교적 오랜 시간이 걸립니다. 현재(2021년 12월) 이더리움 데이터의 크기는 약 1 TB 정도입니다. 아래 통계에서 확인할 수 있습니다.

<https://etherscan.io/chartsync/chaindefault>

두 번째는 직접 노드를 운영하지 않고 외부 이더리움 노드 서비스를 이용하는 것입니다. 가장 많이 알려진 서비스는 인퓨라(Infura)입니다. 대부분의 서비스들은 유료 또는 무료 옵션을 제공하므로 자신에게 맞는 선택을 하면 되겠습니다.

서비스	URL
인퓨라(Infura)	https://infura.io/
알케미(Alchemy)	https://www.alchemy.com/
퀵노드(QuickNode)	https://www.quicknode.com/
아카이브 노드	https://archivenode.io/

1.3. 지분증명

1.3.1 단계별 전환

앞서 작업증명에 관한 이야기를 했습니다. 그런데 작업증명에 대한 비판이 많습니다. 왜냐하면 엄청 난 규모의 장비(수백대의 컴퓨터가 도서관의 책장처럼 즐비하게 늘어서 있는)를 마련하고 조건에 맞는 값을 찾으려고 24/7 돌아가는 것은 누가 봐도 에너지 낭비이기 때문입니다. 또 채굴 경쟁이 심해 질수록 더 많은 컴퓨팅 자원이 필요합니다. 그리고 전력을 공급하기 위해 화력이나 원자력 발전 같은 환경에 나쁜 영향을 주는 시설도 함께 유지해야 합니다.

그래서 비트코인을 제외한 많은 블록체인들이 “지분증명(Proof of Stake, PoS)”을 선택하는 추세에 있습니다. 이더리움은 프로젝트 초기부터 장기적으로 지분증명으로 전환하겠다는 것을 로드맵에 명시했습니다.

지분증명이란 해시 파워를 확보하기 위한 채굴 시설에 많은 자본을 투입하지 말고 대신 네트워크에서 발생되는 지분, 즉 암호화폐를 일정 수량 보유하면 해시 파워 없이도 블록을 만들거나 검증할 수 있는 자격을 주겠다는 것입니다. 지분증명에서는 채굴자 대신 “검증자(Validator)”라는 용어를 사용합니다.

2020년 12월 1일 드디어 이더리움 지분증명의 첫 단계인 비콘(Beacon) 체인이 시작되었습니다. 지분증명을 위한 예치 컨트랙트는 다소 극적으로, 11월 24일 21시(한국 시간) 마감 몇 시간 전에 목표 수량 524,288를 넘었고, 목표 검증인 수인 16,384를 초과한 총 20,163명의 “제네시스(Genesis)” 검증자들을 모았습니다. 현재(2021년 11월) 지분증명으로 락업된 이더의 수량은 800만개 이상이며 발급된 검증키는 26만개에 달합니다. 이더리움의 지분증명은 대단히 성공적인 출발을 한 셈입니다.

그러나 비콘 체인은 지분증명의 첫 번째 단계이고 아직 트랜잭션을 처리할 수 없는 상태입니다. 단지 검증자들이 블록을 검증하고 전파하는 서명 트랜잭션만 저장하고 있습니다. 비콘 체인 탐색기는 아래 링크에서 확인할 수 있습니다.

<https://beaconcha.in/>

이더리움 지분증명의 두 번째 단계는 “샤딩(sharding)”입니다. 현재 단일 블록체인에서 처리하던 트랜잭션들을 샤드(shard)라고 하는 작은 단위의 네트워크로 나누어 처리하는 것을 말합니다. 마치 관계형데이터베이스에서 하나의 테이블을 기간별로 나누어 여러 테이블에 분할 저장하는 것과 유사하다고 볼 수 있습니다. 트랜잭션을 나누어서 처리하므로 처리 속도를 높일 수 있습니다. 샤드의 트랜잭션을 검증하는 것은 비콘 체인입니다.

세 번째 단계는 “실행환경(Execution Environment)”입니다. 지금과 같이 스마트 컨트랙트가 실행되는 환경이 되겠습니다. 주요 목표는 현재 EVM을 “eWasm”이라고 명명된 “웹어셈블리” 기반의 실행엔진

으로 전환하는 것입니다. “웹”이라는 단어가 들어가서 웹과 관련되어 있는 것으로 오해할 수 있지만 “어셈블리”에 초점을 맞춰야 있습니다. 솔리디티 뿐만 아니라 다른 프로그래밍 언어로 작성된 스마트 컨트랙트를 eWasm으로 컴파일하여 성능을 향상시키고 또 확장성을 높이겠다는 것입니다.

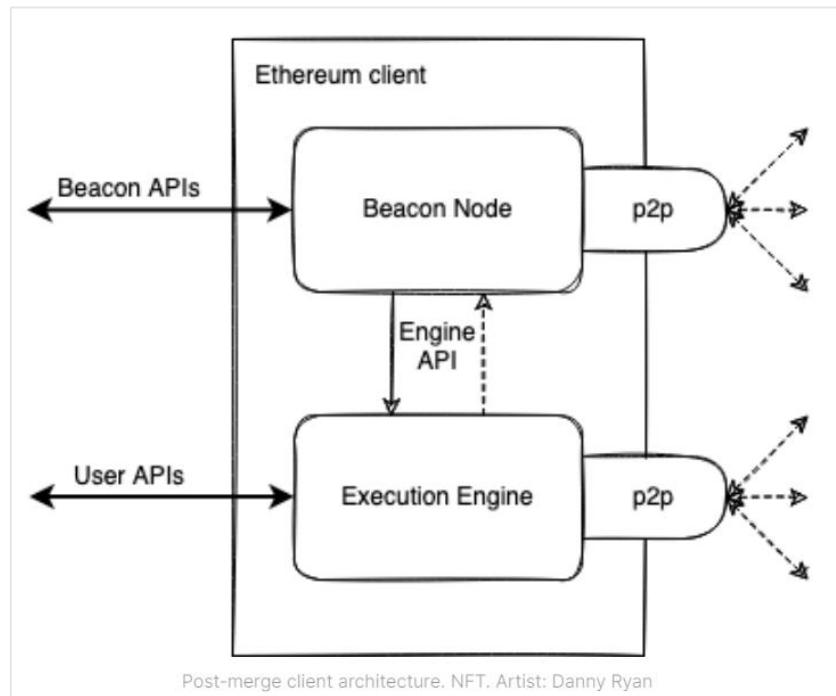
1.3.2. 작업증명 종료와 머지(The Merge)

앞에서 이야기한 것처럼 이더리움은 작업증명에서 지분증명으로 전환하는 과정 중에 있습니다. 물론 원래 로드맵에 계획된 샤딩과 실행환경을 모두 완료하려면 아직도 몇 년은 더 걸릴 것으로 보이지만 일단 현재 이더리움의 작업증명 방식을 조기에 “종료(switch off)” 하는 것이 가능하다고 보고 있습니다.

작업증명의 조기 종료를 “머지(The Merge)”라고 이름을 붙였습니다. 현재 이더리움 메인넷을 비콘체인에 합친다는 말인데, 사실은 하나의 체인으로 합치는 것이 아니라 두 개의 체인이 공존하되, 현재 채굴자들이 하고 있는 블록 생성을 비콘 체인의 검증자들이 대신하겠다는 말입니다.

2021년 10월에 그리스에 모인 이더리움 클라이언트 개발자들이 각자 자신들의 개발한 클라이언트를 서로 연결하여 이것이 가능한지 워크샵을 진행했습니다.

<https://consensys.net/blog/ethereum-2-0/an-update-on-the-merge-after-the-amphora-interop-event-in-greece/>



궁극적으로는 이더리움은 각 계층에서 정해진 기능을 수행하는 다수의 블록체인들로 구성되는 것이고 비콘 체인은 모든 체인들을 관리하는 탈중앙화된 지분증명 블록체인이 되는 것입니다. 머지에 관한 전반적인 내용은 아래 글에 잘 정리되어 있습니다.

https://tim.mirror.xyz/sR23jU02we6zXRgsF_oTUkttL83S3vyn05vJWnnp-Lc

이더리움 클라이언트 개발자들은 공식적으로 “이더리움 1.0”이나 “이더리움 2.0”이라는 용어를 쓰지 않기로 합의했습니다. 이더리움은 하나이고 단지 비콘 체인을 “합의 계층”, 현재 이더리움을 “실행 계층”으로 구분합니다. 즉 각각의 체인은 변함없이 예전과 다름없이 동작하고 노드들끼리 서로 필요한 데이터를 주고받을 뿐입니다.

현재 머지는 개발 단계에 있으며 순조롭게 진행되고 있습니다. 빠르면 2022년 안에 머지를 완료할 계획을 세우고 있는데, 완료 후에는 현재 작업증명에서 지급되던 블록 보상은 사라지고(당연히 채굴자들도 함께) 비콘 체인의 블록 보상과 수수료가 검증자에게 돌아갑니다. 블록 보상은 아직 인출이 안되지만 수수료는 실행 레이어의 계정(Etherbase 계정)으로 지급되므로 인출이 가능할 것으로 예상하고 있습니다.

1.4. 정리

1장에서는 블록체인을 개관하고 이더리움의 기본 동작 원리를 통해 블록체인의 여러 가지 사항들을 살펴보았습니다. 작업증명, 블록, 계정, 트랜잭션, 하드포크 등의 용어는 블록체인에서 흔하게 사용되는 단어들이고 앞으로 자주 등장하게 되므로 확실하게 개념을 정리하는 것이 좋습니다.

솔리디티(Solidity)



이번 장에서는 이더리움의 스마트 컨트랙트 프로그래밍 언어인 솔리디티에 관해서 이야기 합니다. 주로 기본적인 문법에 관한 내용이 되겠습니다.

2.1. 스마트 컨트랙트

블록체인에서 실행되는 프로그램을 일반적으로 "스마트 컨트랙트(Smart Contract)"라고 합니다. 스마트 컨트랙트의 개념은 비트코인이나 이더리움이 등장하기 훨씬 전에 나왔지만 사람의 개입없이 코드에 의해 되돌릴 수 없는 계약의 실행이 가능해진 것은 이더리움에서 비로소 실현되었다고 할 수 있습니다. 스마트 컨트랙트는 한번 블록체인에 배포되면 수정이 불가능하고 작성된 코드대로 실행됩니다. 그래서 "Code is law"라고 말을 하기도 하고 "Self-executing"이라는 표현도 합니다.

이더리움이 비트코인과 가장 크게 차별되는 점이 바로 "튜링 완전한" 프로그래밍 언어로 스마트 컨트랙트를 작성할 수 있다는 것입니다. 비트코인에서도 제한적으로 스크립트를 작성할 수 있지만 "튜링 완전한" 프로그래밍은 아닙니다.

스마트 컨트랙트를 실행할 수 있는 블록체인들은 저마다 나름의 언어로 스마트 컨트랙트를 작성할 수 있도록 지원하고 있습니다.

블록체인	언어
이더리움(Ethereum)	솔리디티(Solidity)
카르다노(Cardano)	하스켈(Haskell)
솔라나(Solana)	러스트(Rust)

이더리움은 기존의 프로그래밍 언어를 차용하지 않고 전용 프로그래밍 언어인 솔리디티를 독자적으로 개발하고 있습니다. 현재 시점(2021년 12월)을 기준으로 아직 1.0이 나오지 않은 상태이고 최신 버전은 0.8.11입니다. 두 번째 자리의 버전이 올라갈 때마다 호환성이 유지되지 않는 변경사항(breaking change)들이 발생할 수 있습니다.

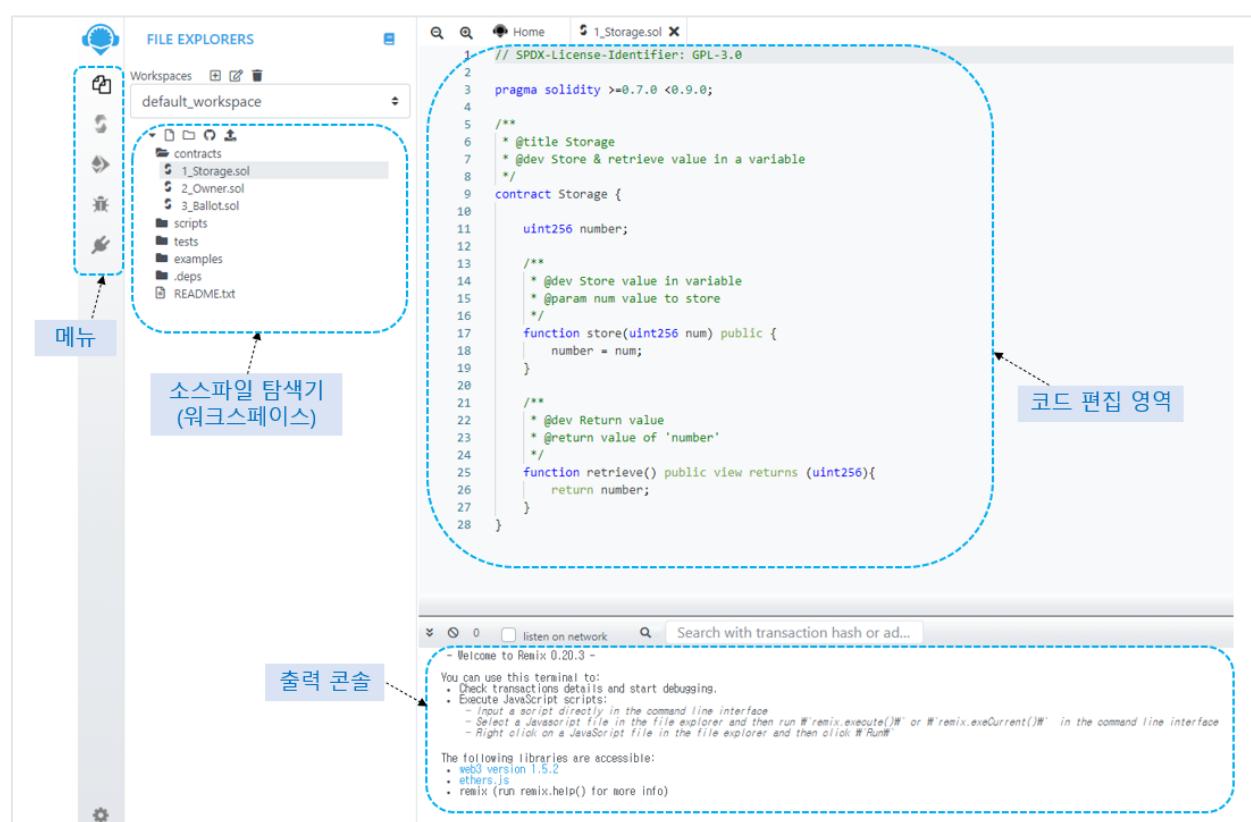
솔리디티로 스마트 컨트랙트를 작성하고 실행해볼 수 있는 환경으로는 웹 기반의 스마트 컨트랙트 개발도구인 “리믹스(Remix)”가 있습니다. 웹 사이트에 접속해서 코드 작성과 컴파일, 실행, 배포를 쉽게 할 수 있기 때문에 솔리디티를 배울 때 매우 유용하게 쓸 수 있는 도구 중 하나입니다. 또 여러 가지 플러그인들도 제공되고 있어서 다양한 작업이 가능합니다.

아래 링크에 접속하면 리믹스를 만날 수 있습니다.

<http://remix.ethereum.org/>

2.2. 리믹스(Remix) 사용법

리믹스는 웹기반이라는 것만 제외하면 일반적인 개발도구와 구성이 유사합니다. 현재 시점에서 리믹스 버전은 0.20.3입니다.



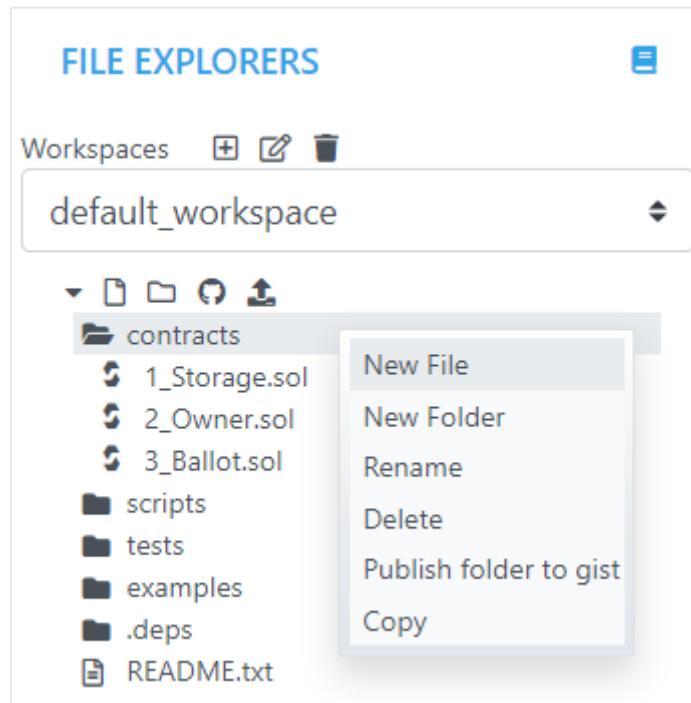
리믹스는 컨트랙트 작성부터 컴파일, 배포, 그리고 디버깅까지 할 수 있는 도구입니다. 메타마스크와 같은 지갑과 연동하는 것이 가능하고 직접 이더리움 클라이언트와 연결될 수도 있습니다. 여기서는 기본적인 사용법을 위주로 살펴보겠습니다. 상세 매뉴얼은 아래 사이트를 참고하기 바랍니다.

<https://remix-ide.readthedocs.io/en/latest/index.html>

간단한 컨트랙트를 작성해보면서 필수적인 기능을 알아보겠습니다. 먼저 새로운 소스파일 하나를 생성합니다. 소스파일 탐색기에서 contracts 폴더를 선택하고 마우스 오른쪽 버튼을 클릭하면 메뉴가 나옵니다. New File을 선택하고 파일명을 입력합니다. 확장자를 생략하면 .sol이 자동으로 붙습니다. contracts 폴더가 아니더라도 새로운 폴더를 만들어서 작성해도 상관없습니다.

폴더나 파일을 선택하고 오른쪽 클릭으로 열리는 메뉴에는 New Folder, Rename, Delete 등의 기능이 제공되므로 워크스페이스를 다양하게 구성할 수 있습니다.

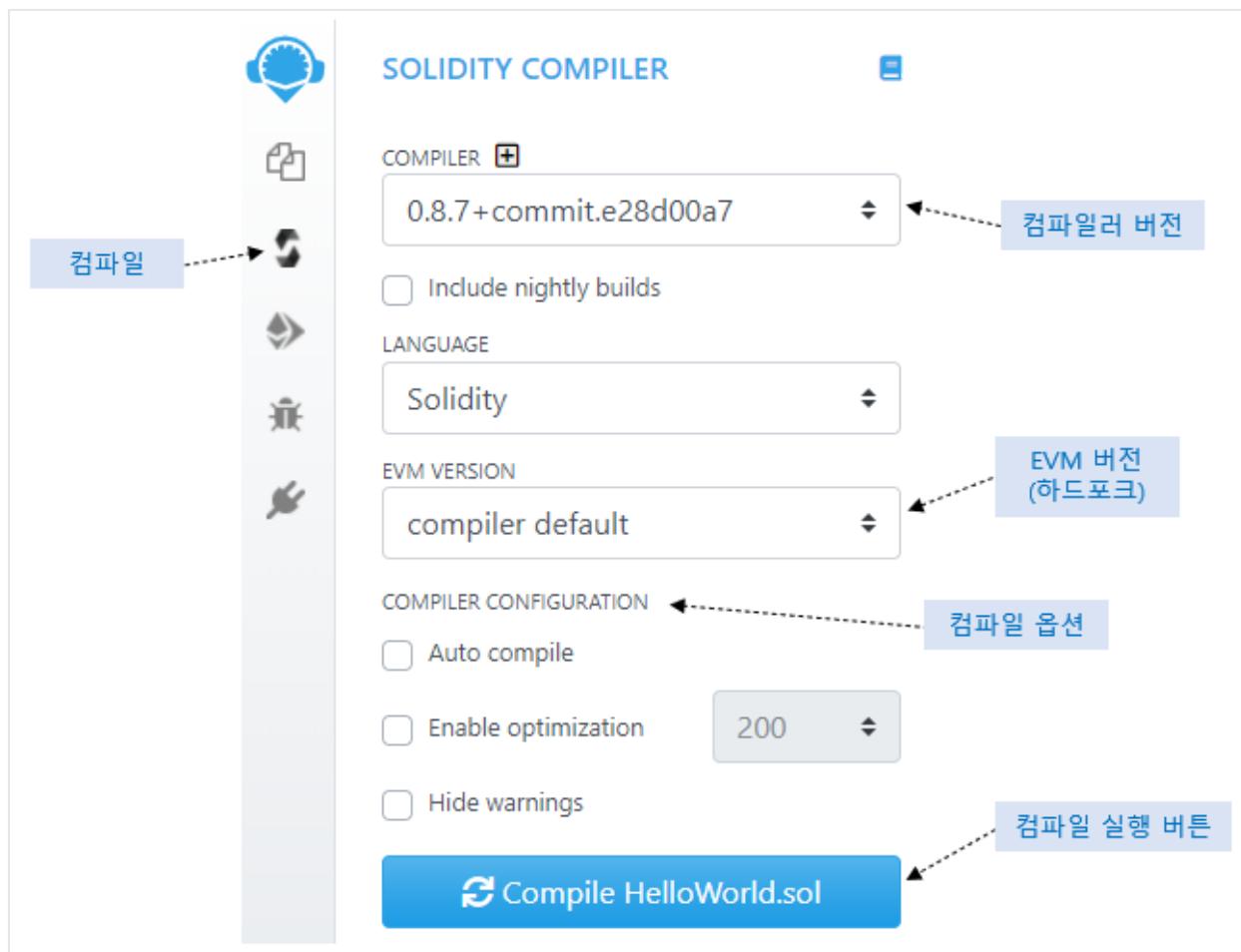
리믹스는 자바스크립트도 실행할 수 있습니다. scripts 폴더는 자바스크립트 파일을 모아 놓는 폴더입니다. tests는 단위 테스트를 작성하는 폴더입니다. 또 외부 컨트랙트를 import하는 경우 자동으로 .deps 폴더가 생기면서 해당 컨트랙트가 설치될 수 있습니다.



우측에는 코드 작성과 편집 영역이 있습니다. 여기에서 컨트랙트를 작성하고 편집합니다.

```
1 // SPDX-License-Identifier:MIT
2 pragma solidity ^0.8.0;
3
4 import "./MyContract.sol";
5
6 contract HelloWorld {
7
8     string public s;
9
10    constructor(string memory _s) {
11        s = _s;
12    }
13
14    function set(string memory _s) public {
15        s = _s;
16    }
17}
18
```

이 컨트랙트를 컴파일하려면 왼쪽 컴파일 메뉴를 선택합니다. 컴파일러 버전과 옵션 등을 설정할 수 있고 방금 작성한 컨트랙트 파일명이 표시된 버튼을 클릭하면 컴파일이 수행됩니다. 보통 디포트 설정을 그대로 사용하면 됩니다.



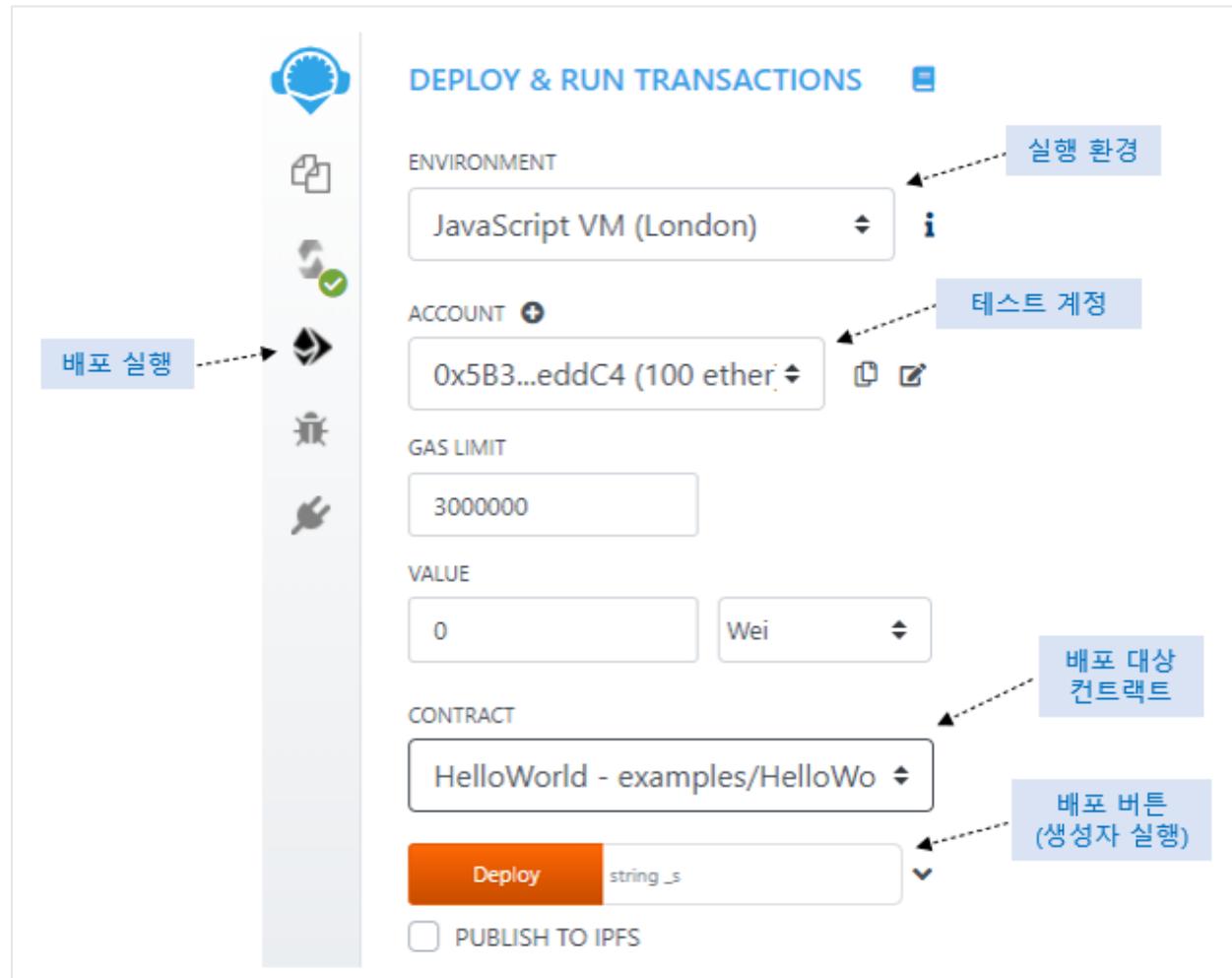
컴파일이 성공하면 다음과 같이 컴파일 메뉴 아이콘에 체크 표시가 생깁니다. 오류가 나면 컴파일은 실패하고 오류 개수가 표시됩니다. 오류가 발생한 곳을 찾아 수정한 후 다시 컴파일해야 합니다.



컴파일을 하면 컴파일 결과 파일들이 artifacts 폴더 안에 생깁니다. JSON 형식의 파일인데 주로 컨트랙트 검증이나 애플리케이션에서 사용됩니다.

단순히 컨트랙트를 리믹스 상에서만 실행하는 용도로 사용하려면 artifacts가 생기지 않도록 비활성화 할 수 있습니다. 좌측 맨 아래에 있는 설정 메뉴를 클릭하고 메타 데이터 생성 옵션을 꺼면 되겠습니다.

컴파일에 성공하면 배포 실행 메뉴로 이동합니다.



맨 위에 있는 "ENVIRONMENT"은 실행환경을 의미하는데, 이것은 컨트랙트가 배포되는 위치라고 생각하면 됩니다. 아래와 같은 옵션들을 선택할 수 있습니다.

- JavaScript VM

배포된 컨트랙트는 리믹스 내에서만 존재합니다. 이 환경은 코드를 즉시 실행해볼 수 있는 가상 더리움이라고 할 수 있습니다. 배포와 실행 결과들은 저장되지 않기 때문에 리믹스를 닫으면 사라지게 됩니다. 작성한 컨트랙트 코드를 빠르게 실행하고 확인할 때 유용합니다.

- Injected Web3

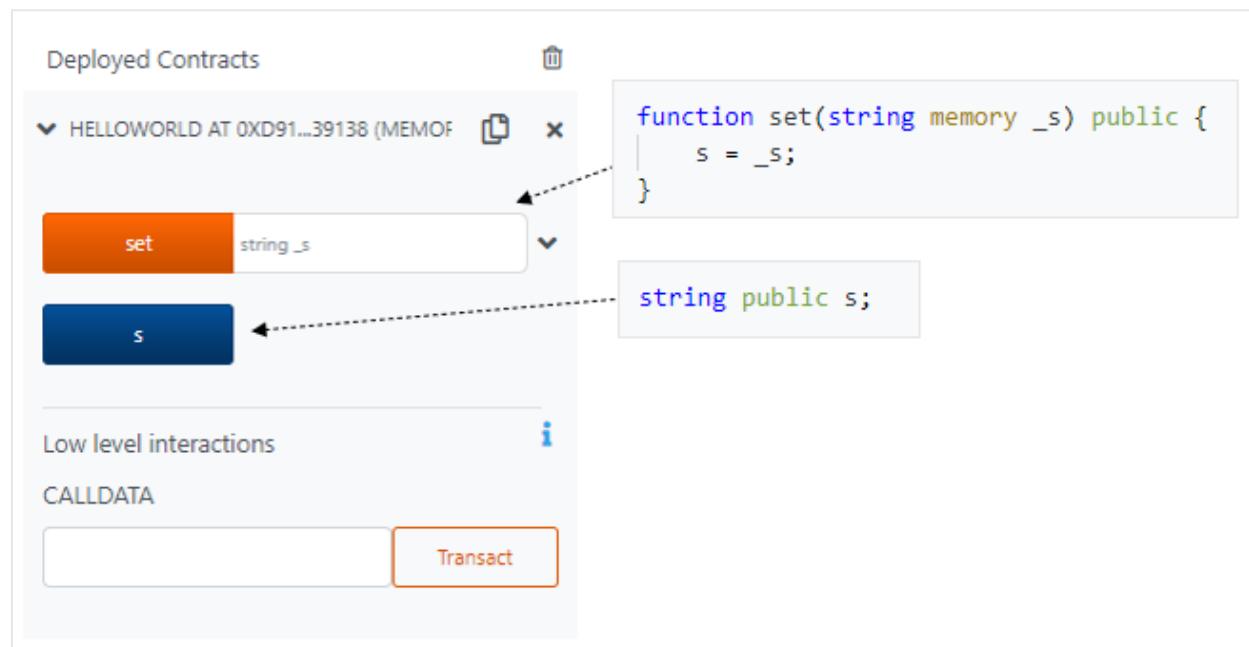
"Injected"라는 것은 메타마스크와 같이 웹브라우저에 플러그인으로 설치된 외부 지갑을 의미합니다. 외부 지갑이 애플리케이션에 결합되는 형태로 동작합니다. 이 옵션을 사용하여 이더리움 메인넷 또는 테스트넷으로 배포할 수 있습니다.

- Web3 Provider

리믹스는 직접 이더리움 클라이언트에 연결할 수 있습니다. 흔히 "RPC endpoint"라고 부르는 외부 노드에 연결할 때 이 옵션을 사용합니다. 예를 들어 로컬에 Geth와 같은 이더리움 클라이언트가 설치되어 있다면 `http://127.0.0.1:8545`로 설정하여 연결할 수 있습니다.

"ACCOUNT"는 컨트랙트의 함수를 호출하는 테스트 계정입니다. 함수를 실행하려면 이더가 필요한 경우가 있기 때문에 각 계정들은 100개의 테스트 이더를 가지게 됩니다. "CONTRACT"에는 방금 컴파일한 컨트랙트가 표시됩니다. "Deploy"를 클릭하면 지정된 실행 환경에 컨트랙트가 배포됩니다. 만약 컨트랙트가 생성자 파라미터를 가지고 있는 경우에는 값을 넣을 수 있는 필드가 활성화됩니다.

이렇게 배포가 성공적으로 끝나면 하단에 컨트랙트를 호출할 수 있는 실행 UI가 생깁니다. 외부에서 호출할 수 있는 함수가 표시되고 클릭해서 실행할 수 있습니다.



이더리움의 상태를 변경하는 것과 그렇지 않은 것을 구분하기 위해 함수 실행 버튼의 색이 다릅니다. 상태 변수의 값을 바꾸는 함수는 짙은 오렌지색 계열로 표시됩니다. 함수 전달인자가 있는 경우에는 값을 입력할 수 있는 필드가 활성화됩니다. 조회 함수 버튼은 푸른색 계열로 표시됩니다.

버튼을 클릭하여 함수를 실행하면 오른쪽 하단의 출력 콘솔 영역에 다음과 같은 결과가 출력됩니다. 리턴 값이 있는 조회 함수의 경우에는 실행 UI에 결과가 표시됩니다.

	[vm] from: 0x5B3...eddC4 to: HelloWorld.set(string) 0xf8e...9fBe8 value: 0 wei data: 0x4ed...00000 logs: 0 hash: 0xde1...b93d5
status	true Transaction mined and execution succeed
transaction hash	0xde1601cee99b6de7d19fd2884b876937575ecff9d1cc4664d26a537a7b3b93d5 🔗
from	0x5B380a6a701c568545dCfcB03FcB075f56beddC4 🔗
to	HelloWorld.set(string) 0xf8e81D47203A594245E36C48e151709F0C19fBe8 🔗
gas	80000000 gas 🔗
transaction cost	30238 gas 🔗

이더리움 스마트 컨트랙트는 일단 배포되면 코드를 변경할 수 없습니다. 따라서 컨트랙트의 소스파일을 수정하는 경우에는 컴파일과 배포를 다시 수행해야 하고, 그렇게 배포된 컨트랙트는 이전 컨트랙트와 이름이 동일해도 전혀 다른 컨트랙트로 인식합니다(컨트랙트 계정주소가 다릅니다).

2.3. 컨트랙트 구조

일반적인 컨트랙트 구조는 아래와 같습니다. 솔리디티는 상속이 가능하기 때문에 조금 더 복잡해질 수 있지만 대략적인 구조라고 이해하면 되겠습니다.

```
1 // SPDX-License-Identifier:MIT
2 pragma solidity ^0.8.0;
3
4 import "./MyContract.sol";
5
6 contract HelloWorld {
7
8     string public s;
9
10    constructor(string memory _s) {
11        s = _s;
12    }
13
14    function set(string memory _s) public {
15        s = _s;
16    }
17}
18
```

- SPDX-License-Identifier

소스파일의 라이선스 정보를 표시합니다. 생략해도 컴파일 오류는 나지 않지만 명시할 것을 권고하고 있습니다. 라이선스 종류는 <https://spdx.org/licenses/>에서 확인 가능합니다.

- pragma solidity

솔리디티 컴파일러 버전을 명시합니다. npm에서 사용하는 "semver" 표기법(<https://semver.org/>)을 그대로 사용합니다. 예를 들어 다음과 같이 표기하면 0.8.0 이상, 0.9.0 미만의 컴파일러를 기준으로 문법을 체크합니다.

```
pragma solidity ^0.8.0
```

다음과 같이 컴파일러 버전의 범위를 지정하면 사용된 컴파일러가 pragma 범위 내에 있어야 합니다.

```
pragma solidity >=0.7.0 <0.9.0;
```

솔리디티 버전 0.8.0 이전에는 함수 전달인자나 리턴 값으로 구조체를 받을 수 없었기 때문에 다음과 같은 추가적인 ABIEncoderV2 프라그마 지정을 해야 했습니다.

```
pragma solidity ^0.6.0;
pragma experimental ABIEncoderV2;

...
struct UserInfo {
    string userId;
    string name;
    uint256 date;
}

mapping(address => UserInfo) public users;

function getMemmber(address _addr) public view returns (UserInfo memory user) {
    return users[_addr];
}
```

0.8.0 부터는 기본적으로 ABI 인코더 V2가 사용되므로 생략해도 되겠습니다. 만약에 이전 V1을 사용할 경우에는 명시적으로 프라그마를 지정해야 합니다.

```
pragma solidity ^0.8.0
pragma abicoder v1;
```

- import

외부 소스파일을 참조할 때 사용합니다. 참조 경로는 현재 컨트랙트 소스파일을 기준으로 상대경로로 작성하는 것이 일반적입니다. 한 소스파일에 여러 개의 컨트랙트가 존재할 수 있으므로 다음과 같이 선택적으로 import할 수 있습니다.

```
pragma solidity ^0.8.0

import { Contract1, Contrac3 } from "./MyContract.sol"
```

이름 충돌이 발생하는 경우에는 다음과 같이 `as`를 사용하여 별칭을 지정합니다.

```
pragma solidity ^0.8.0

import { Contract1 as ContractAlias, Contrac3 } from "./MyContract.sol"
```

- `contract { ... }`

컨트랙트의 코드가 작성됩니다. 하나의 소스파일에 여러 개의 `contract { ... }`가 작성될 수 있습니다. 상태변수, 이벤트, modifier, 함수 등이 컨트랙트 블록 안에 들어갑니다.

솔리디티 기본 문법을 살펴보기 전에 솔리디티 스마트 컨트랙트를 작성할 때 지켜야 할 기본 규칙들을 요약해보겠습니다.

- ✓ 대소문자를 구분합니다.
- ✓ 문장 끝에는 세미콜론(:)을 붙입니다.
- ✓ 주석은 // 과 /* ... */ 이 가능합니다.
- ✓ 소스파일명과 컨트랙트 이름은 일치할 필요가 없습니다.
- ✓ 하나의 소스파일에 여러 개의 컨트랙트를 작성할 수 있습니다.
- ✓ `undefined`와 `null`이 존재하지 않습니다. 변수 선언을 하면 기본 초기값이 저장됩니다.

그리고 이더의 단위를 알고 있어야 합니다. 솔리디티에서는 `ether`, `wei`, `gwei` 등이 예약어로 사용됩니다. `wei`는 최소단위로, 1이더는 1×10^{18} `wei`이고 1 `gwei`는 10^9 `wei`를 말합니다.

```
1 ether == 1e18 wei
1 gwei == 1e9 wei
```

솔리디티는 아직 개발 중인 언어이기 때문에 공식 사이트에서 최신 정보를 얻는 것이 좋습니다. 아래 링크를 참고하기 바랍니다. 여기서는 0.8.10을 기준으로 설명합니다.

<https://blog.soliditylang.org/>

<https://docs.soliditylang.org/en/latest/>

앞으로 나오는 예제들을 리믹스에서 작성하여 테스트해보면 솔리디티를 이해하는데 도움이 될 것입니다.

2.4. 데이터 타입

솔리디티는 정적 타입을 가지고 있는 프로그래밍 언어입니다. 정적 타입이라는 것은 컴파일 단계부터 타입을 체크한다는 것을 의미합니다. 자료형은 크게 “Value” 타입과 “Reference” 타입 두 가지로 나눌 수 있습니다. 자바스크립트와 비교하면 Primitive 타입과 Object 타입에 대응한다고 볼 수 있습니다. Value 타입은 변수가 값을 저장하고 있고 Reference 타입은 값이 있는 곳을 가리키는 포인터를 저장하고 있다는 점에서 차이가 있습니다.

2.4.1. Value 타입

Value 타입에는 다음과 같은 타입들이 있습니다.

정수형	int8, int16, int24 ... int256(=int)
부호 없는 정수형	uint8, uint16, uint24 ... uint256(=uint)
불리언(true, false)	bool
계정주소	address
고정길이 바이트	bytes1, bytes2, bytes3 ... bytes32
컨트랙트	Contract
열거형	enum

정수형의 뒤에 붙은 숫자는 비트를 나타냅니다. uint256은 256비트 정수형을 의미합니다. 8의 배수로 증가하여 256까지 있습니다. 숫자를 생략하는 경우 256비트로 간주합니다. 정수형 타입의 최댓값과 최솟값은 다음과 같이 구할 수 있습니다.

```
type(uint256).min  
type(uint256).max
```

정수형에 사용할 수 있는 연산자는 다른 언어와 유사합니다. **은 지수를 나타냅니다.

비교	<=, <, ==, !=, >, >=
산술	+, -, *, /, %, **
비트	&, , ^, ~
시프트	>>, <<

솔리디티 0.8.0 이후부터는 산술 연산에서 오버(언더)플로우가 발생할 경우 revert가 발생하면서 실행이 중단됩니다. 0.8.0 이전에는 revert가 되지 않아서 혼란을 초래하는 경우가 있었습니다. 예를 들어 ^0.7.0 컴파일러에서는 다음과 같은 코드가 아무 문제없이 실행됩니다.

```
int8 x = -2**7; // -128
assert(-x == x); // true 128 == -128
```

bool 타입에 적용되는 연산자 역시 다른 언어와 동일합니다. 논리연산에서는 “short-circuit” 규칙이 적용됩니다. 즉 A || B에서 A가 true이면 B가 true인지 false인지 평가하지 않습니다.

같음	==
다름	!=
논리곱(AND)	&&
논리합(OR)	
부정(NOT)	!

고정길이 바이트 bytes1 ... bytes32는 1부터 시작하여 1씩 증가하여 32까지 가능합니다. 이름에서 알 수 있듯이 바이트형의 데이터를 저장합니다. 이 타입은 Value 타입이지만 배열처럼 인덱스와 length라는 읽기 전용 속성을 가지고 있습니다.

bytes라는 이름 때문에 뒤에 나오는 Reference 타입 중 하나인 bytes 타입과 혼동할 우려가 많으니 주의하기 바랍니다.

```
pragma solidity ^0.8.0

function goo() public pure returns (bytes1){
    bytes8 b8 = "ABCDEFGH";
    return b8[0];
}
```

- address 타입

이더리움 계정은 트랜잭션의 주체가 되기 때문에 매우 중요합니다. 1장에서 사람이 소유하는 계정을 EOA라고 하고 스마트 컨트랙트의 계정을 CA라고 했습니다. 솔리디티에서 이러한 계정들은 20바이트 길이의 16진수로 표현되는 address 타입으로 표현됩니다.

```
address addr = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;
```

address 타입은 Value 타입에 속하지만 다른 타입들과 다르게 멤버 속성과 메소드를 가지고 있습니다. 예를 들어 계정의 잔액을 리턴하는 balance 속성이 있습니다.

```
address addr = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;
uint256 v = addr.balance;
```

또 계정들은 서로 이더를 주고받을 수 있기 때문에 transfer()와 send() 메소드를 제공합니다. 예제를 통해 transfer()가 어떻게 사용될 수 있는지 알아보겠습니다. 다음과 같은 컨트랙트를 작성합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {
    constructor() payable {}

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }

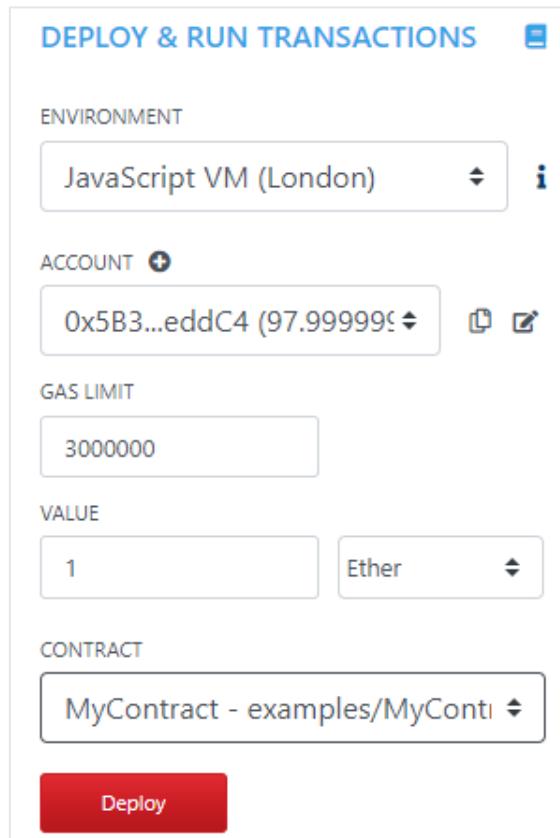
    function sendEther(address payable _addr) public {
        _addr.transfer(0.1 ether);
    }
}
```

sendEther() 함수를 보면 address 타입의 _addr을 인자로 받아서 _addr에게 0.1 이더를 transfer()로 전송합니다. 그런데 여기서 주의할 것은 transfer 앞에 있는 계정이 이더를 받는 계정입니다. 이더를 보내주는 것은 컨트랙트 계정인 MyContract입니다.

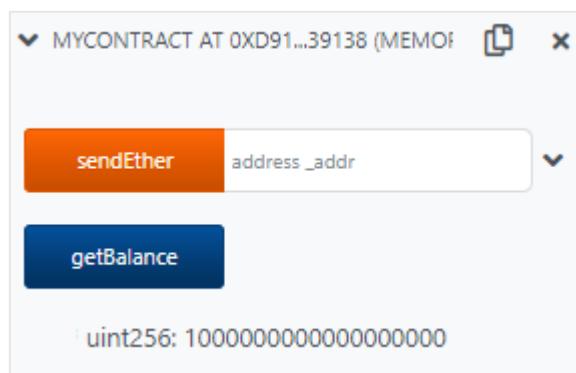
하지만 컨트랙트가 처음부터 이더를 가질 수는 없으므로 외부 계정, 즉 EOA가 이 컨트랙트로 이더를 전송해야 합니다.

이 경우에 MyContract의 생성자에 “payable”이라는 지시어를 붙이고 배포할 때 이더를 같이 보낼 수 있습니다. _addr 역시 이더를 받기 때문에 payable로 지정되었습니다.

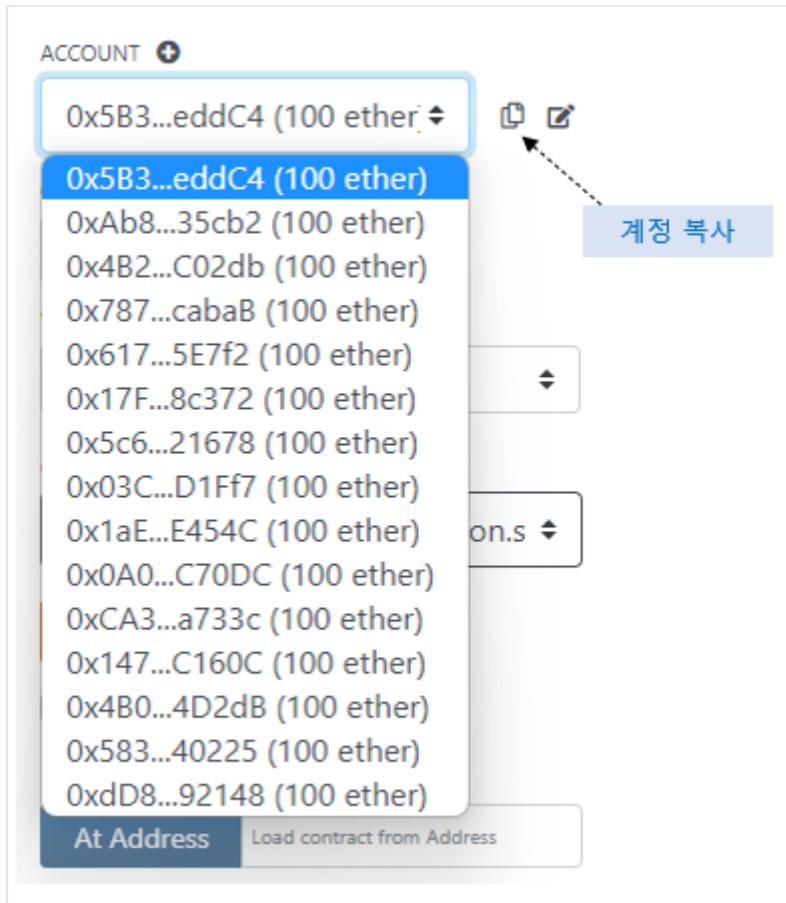
실행 배포 메뉴에서 아래와 같이 VALUE 필드에 전송할 이더를 입력하고 Deploy를 클릭합니다.



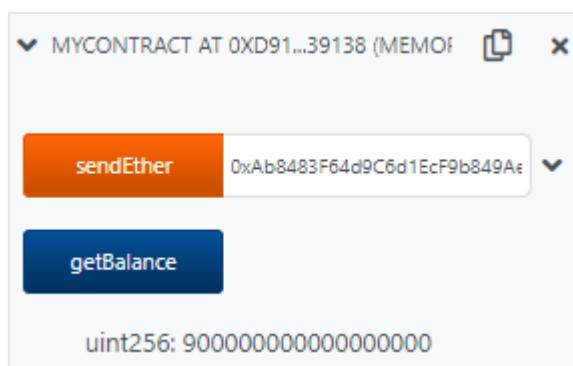
이제 getBalance를 클릭해보면 받은 이더 수량이 1,000,000,000,000,000,000 wei로 표시됩니다.



이제 리믹스의 ACCOUNT 항목의 계정 중 하나를 복사하여 sendEther의 입력 필드에 넣고 함수를 실행합니다.



테스트 계정 0xAB84...의 잔액이 0.1 이더 늘어나고 반대로 MyContract의 잔액은 0.1 이더가 감소하여 900,000,000,000,000,000 wei가 됩니다.



계정이 컨트랙트(CA)인 경우에는 컨트랙트가 제공하는 외부 함수를 호출할 수 있는 call, delegatecall, staticcall과 같은 “저수준(low-level)” 호출 메소드를 제공합니다. 솔리디티 함수를 설명할 때 조금 더 자세하게 다루기로 하겠습니다.

address 타입은 balance 외에도 여러 속성과 메소드를 제공하고 있습니다. 전체 속성과 메소드는 아래 링크를 참고하기 바랍니다.

<https://docs.soliditylang.org/en/latest/units-and-global-variables.html#address-related>

- contract 타입

우리가 컨트랙트를 작성하면 그 컨트랙트는 하나의 타입이 될 수 있습니다. 어떤 타입의 변수를 선언하는 것과 마찬가지로 컨트랙트를 타입의 변수를 선언할 수 있습니다. 예를 들어 Greeter라는 컨트랙트가 있다고 하면 다음과 같이 변수 c의 타입으로 사용할 수 있습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    Greeter c;

    function foo() public returns (string memory){
        c = Greeter(0xf96B30bFD86739D715Aa5F56d139377b73dC94b3);
        return c.greet();
    }
}

abstract contract Greeter {
    function greet() virtual public view returns (string memory);
}
```

address 타입을 contract 타입으로 캐스팅이 가능하고 그 반대로 가능합니다. Greeter라는 컨트랙트 타입으로 컨트랙트의 계정 주소를 변환하면 그 컨트랙트의 함수를 호출할 수 있습니다. 즉 컨트랙트 타입의 멤버 메소드는 그 컨트랙트가 구현한 외부 호출 가능한 함수가 되는 것입니다.

컨트랙트 내에서 새로운 컨트랙트를 생성할 수도 있습니다.

```
contract MyContract {
    constructor(uint256 _v) {
        ...
    }
}

MyContract c = new MyContract(2000);
```

```
address addr = address(c);
```

- enum

enum은 열거형 타입으로 부호 없는 정수 0부터 시작하여 256개의 멤버를 가질 수 있습니다. 타입 캐스팅은 명시적으로 해주어야 합니다. enum으로 정의한 범위를 벗어나는 숫자를 캐스팅하는 경우는 오류가 발생합니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract {

    enum Switch { OFF, ON }

    function foo(uint8 _v) public pure returns (bool) {
        bool result = false;
        if (_v == uint8(Switch.ON)) {
            result = true;
        }
        return result;
    }
}
```

2.4.2. Reference 타입

Reference 타입에는 다음과 같은 것들이 있습니다. 배열의 경우 고정길이 또는 가변길이 배열 모두 사용 가능합니다.

배열(고정, 가변)	<code>T[], T[k]</code>
문자열	<code>string</code>
바이트 문자열	<code>bytes</code>
구조체	<code>struct</code>
해시 테이블	<code>mapping</code>

Reference 타입은 그 타입이 데이터가 저장되는 위치 "Data location"을 지정해주어야 하는데 자바와 비교하면 힙(heap) 영역에 객체 타입의 데이터가 저장되는 것과 유사합니다. 만약 위치를 지정하지 않으면 컴파일 오류가 발생합니다. 예외적으로 상태변수를 선언하는 경우에는 Data location이 항상 storage이므로 생략할 수 있습니다.

<code>storage</code>	영구 저장(블록체인)
<code>memory</code>	지역 변수, 함수 호출이 끝나면 삭제
<code>calldata</code>	external 함수의 전달인자, 변경불가

예를 들어 배열을 인자로 받는 함수의 경우는 다음과 같이 쓸 수 있습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    function foo(uint256[] memory _arr) public pure returns (uint256) {
        return _arr.length;
    }
}
```

배열 타입은 push와 pop이라는 메소드를 제공합니다. push는 배열의 끝에 원소를 추가하는 것이고 pop은 배열의 마지막 원소를 제거하는 것입니다(길이가 변경). 마지막 원소가 아닌 배열의 원소를 지우는 경우는 delete를 사용합니다. 그러나 이것은 원소의 값을 초기화할 뿐 배열의 길이가 줄어드는 것은 아닙니다.

```
function addElement(uint256 _v) public {
    arr.push(_v);
}
```

```
function removeElement() public {
    arr.pop();
}

function clearElement(uint _index) public {
    delete arr[_index];
}
```

2차원 배열의 경우는 선언할 때 행과 열의 순서가 바뀌기 때문에 주의해야 할 부분이 있습니다. 예를 들어 2×3 배열 $\begin{bmatrix} [1,2,3], [4,5,6] \end{bmatrix}$ 의 경우는 다음과 같이 표현됩니다.

```
uint256[3][2] arr = [[1,2,3], [4,5,6]];
```

그러나 원소를 가져올 때는 다시 행과 열을 원래대로 참조해야 해야 합니다. 행의 인덱스는 0에서 1 까지 존재하고 열의 인덱스는 0에서 2까지 존재합니다.

```
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 4;
arr[1][1] = 5;
arr[1][2] = 6;
```

Reference 타입의 데이터는 처음 변수에 넣은 다음 다시 다른 변수에 할당하는 경우 차이가 있습니다. 예를 들어 다음과 같은 `copyRefMemory()` 함수와 `copyRefStor()` 두 함수를 비교해 보겠습니다.

두 함수 모두 상태변수로 선언된 배열 `arr`을 내부에 선언된 변수에 할당합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    uint256[5] public arr = [1,2,3,4,5];

    function copyRefMemory() public view returns (uint256[5] memory){
        uint256[5] memory arrLocal = arr;
        arrLocal[0] = 100;
        return arrLocal;
    }

    function copyRefStor() public returns (uint256[5] memory){
        uint256[5] storage arrRef = arr;
```

```

        arrRef[0] = 100;
        return arrRef;
    }
}

```

memory로 지정된 배열 arrLocal에 상태 변수(storage)의 값을 할당하면 새로운 복사본이 만들어집니다. 즉 arrLocal의 값을 변경한다고 해도 원래 arr의 값은 바뀌지 않습니다. 그러나 storage로 지정한 변수 arrRef에 상태 변수를 넣으면 같은 곳을 참조하기 때문에 arrRef의 값을 바꾸면 arr의 값도 변경됩니다.

상태 변수들 사이의 할당은 항상 새로운 복사본을 만듭니다. 아래의 경우 arr2의 원소를 변경해도 arr1의 원소는 바뀌지 않습니다.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract {
    uint256[5] public arr1 = [1,2,3,4,5];
    uint256[5] public arr2 = arr1;
    ...
}

```

string과 bytes 타입은 특별한 배열입니다. string은 길이와 인덱스 속성이 있지만 bytes는 가변 길이 바이트 배열이라고 생각할 수 있습니다. Value 타입 고정길이 바이트 배열 bytes1 ... bytes32와 혼동하지 않도록 주의해야 합니다.

string은 UTF-8 문자셋으로 저장됩니다. 한글은 다음과 같이 저장될 수 있습니다.

```

string public name;
name = unicode"홍길동";

```

- mapping 타입

솔리디티에는 mapping이라는 특별한 Reference 타입이 있습니다. 이 타입은 해시 테이블과 유사한 key-value 장부의 형태를 가지고 있기 때문에 자주 사용됩니다. 형식은 다음과 같습니다.

```
mapping(_KeyType => _ValueType) _VariableName
```

`_KeyType`으로 `Value` 타입과 `bytes`, `string` 등의 타입이 올 수 있지만 `mapping`이나 `struct`처럼 복합 타입은 불가능합니다. `_ValueType`은 모든 타입이 가능합니다.

`_KeyType`으로 저장되는 키 데이터는 그 값의 keccak256 해시 값이므로 정확한 키를 알아야 대응되는 값을 조회할 수 있습니다. 또 임의의 키로 값을 조회하는 경우 `_ValueType`의 초기값이 항상 나오게 되어 있습니다. 마치 모든 가능한 키가 저장되어 초기화된 것처럼 동작합니다.

예를 들어서 `mapping(address=>uint256)` 타입으로 아래와 같은 데이터가 저장되어 있다고 하면 실제로 저장한 적이 없는 임의의 키 값인 `0xf96B30bFD86739D715Aa5F56d139377b73dC94b3`을 넣으면 0이 나오게 됩니다.

keccak256(0x547d73355A851079E0395aDB2C647821b74C7eAF)	1000
keccak256(0x577D296678535e4903D59A4C929B718e1D575e0A)	5000

```
mapping(address => uint256) booking;
uint256 v = booking[0xf96B30bFD86739D715Aa5F56d139377b73dC94b3]; // v = 0
```

● 구조체(struct)

구조체는 사실상 사용자 정의 타입입니다. 기존의 타입들을 모아서 새로운 타입을 정의할 수 있습니다. `struct` 타입은 배열의 원소나 `mapping` 타입의 값으로 들어갈 수 있고 구조체 멤버로 배열이나 `mapping` 타입을 가질 수 있습니다.

구조체에 값을 넣는 것은 `{..}`을 사용하여 각 필드에 값을 넣고 구조체 타입으로 캐스팅하는 방법을 사용할 수 있습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    struct Certification {
        string certId;
        string certName;
        uint regDate;
    }

    struct UserInfo {
        string userId;
        string name;
        mapping (uint => Certification) certs;
```

```

}

mapping(address => UserInfo) public users;

function registerUser(string memory _name, string memory _userId) external {
    users[msg.sender].userId = _userId;
    users[msg.sender].name = _name;
}

function registerCert(uint _certNo, string memory _certId,
                     string memory _certName, uint _regDate) external {
    UserInfo storage u = users[msg.sender];
    u.certs[_certNo] = Certification(
        {certId: _certId, certName: _certName, regDate: _regDate});
}

function getUserCertInfo(address _addr, uint _certNo) public view returns (string memory) {
    Certification memory c = users[_addr].certs[_certNo];
    return c.certId;
}

}

```

구조체에 값을 할당하는 다른 방법으로 각 멤버 필드들의 값을 하나씩 넣을 수도 있습니다.

```

function registerCert(uint _certNo, string memory _certId,
                      string memory _certName, uint _regDate) external {
    UserInfo storage u = users[msg.sender];
    u.certs[_certNo].certId = _certId;
    u.certs[_certNo].certName = _certName;
    u.certs[_certNo].regDate = _regDate;
}

```

구조체 타입을 선언하면 각 멤버 필드의 타입에 맞게 초기화됩니다. 예를 들어 uint256과 string 타입의 필드는 각각 0과 빈 문자열 "" 값을 가지게 됩니다. 한 가지 예외는 구조체 내에 mapping 타입의 멤버는 값이 저장된 후에는 삭제하더라도 초기화가 되지 않는다는 점입니다. 아래와 같은 삭제 함수의 경우를 보겠습니다.

```

function removeUser(address _addr) external {
    delete users[_addr];
}

```

`delete`로 `users` 목록에서 특정 계정주소의 정보를 지우게 되면 `UserInfo` 타입의 멤버인 `userId`와 `name`은 `""`으로 초기화되지만 `mapping` 타입으로 되어 있는 `certs`는 삭제 전에 저장한 값이 계속 남게 됩니다.

				삭제 전	삭제 후
users	0xf96B...	userId		U001	""
		name		Kate	""
		certs	1	certId	C001
				certName	OCP11g
				regDate	1641449090

2.5. constant와 immutable

상태 변수가 `constant` 또는 `immutable`로 선언되는 경우는 배포 이후에는 그 값을 변경할 수 없습니다. `constant`로 선언된 변수는 컴파일 시점에 그 값으로 대체되고 `immutable`은 생성자 실행 시점에 그 값으로 대체합니다. 상태 변수지만 컨트랙트의 스토리지에 저장되는 것은 아닙니다.

`constant`은 선언할 때 할당한 값이 컴파일 시점에 그대로 대체되기 때문에 컨트랙트의 스토리지 영역이나 블록의 데이터의 값을 넣을 수 없습니다. 예를 들어 다음과 같은 경우는 불가능합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {
    uint256 constant v = block.number;
    ...
}
```

그러나 `immutable`에서는 생성자에서 값을 할당하는 것으로 가능합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {
    uint256 immutable v;

    constructor() {
        v = block.number;
    }
}
```

2.6. 제어문

반복문(for, while, do while)과 조건문(if-else)는 자바스크립트와 거의 같습니다. 아래 코드는 모두 같은 결과를 리턴합니다.

```
uint256 s;

for (uint i=0; i<=10; i++) {
    s = s + i;
}

uint256 i;
while(i<=10) {
    s = s + i;
    i++;
}

do {
    s = s + i;
    i++;
} while (i<=10) ;
```

if-else 문은 다음과 같은 형식으로 작성합니다.

```
if (<condition>) {
    ...
} else if (<condition> ) {
    ...
} else {
    ...
}
```

break와 continue의 기능도 동일합니다. 예를 들어 다음과 같은 코드의 결과를 비교해보겠습니다. foo(5)를 호출하면 반복문의 i=5에서 반복문을 중지하게 되므로 s는 5가 됩니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    uint256 public s;

    function foo(uint256 _v) public {
        for (uint256 i; i<10; i++) {
            if (i == _v) {
                break;
            }
            s++;
        }
    }
}
```

```
    }
}
```

continue의 경우는 foo(5)의 결과는 9가 됩니다. continue를 만나면 다시 반복문의 처음으로 돌아가기 때문입니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    uint256 public s;

    function foo(uint256 _v) public {
        for (uint256 i; i<10; i++) {
            if (i == _v) {
                continue;
            }
            s++;
        }
    }
}
```

2.7. 함수

함수는 컨트랙트와 외부를 연결시켜주는 다리와 같은 역할을 합니다. 애플리케이션에서는 보통 ABI(Application Binary Interface)를 통해서 함수를 호출합니다(6장에서 설명합니다). 여기서는 컨트랙트 내에서 함수가 어떻게 사용되는지 살펴보겠습니다.

함수의 형식은 대략 다음과 같습니다.

function	함수명	(전달인자)	Visibility	State mutability	Modifier	returns (리턴 타입)
			external	pure		
			public	view		
			internal	payable		
			private			

Visibility, State mutability, Modifier의 순서는 바꿀 수 있지만 관례적으로 위의 순서를 따릅니다. 함수가 리턴이 없는 경우에는 returns 구문이 생략됩니다.

함수의 리턴 값은 여러 개가 될 수 있고 리턴 변수를 지정할 수도 있습니다. 예를 들어서 다음과 같이 쓸 수도 있습니다.

```
function arithmetic(uint _a, uint _b) public pure returns (uint o_sum, uint o_product){  
    o_sum = _a + _b;  
    o_product = _a * _b;  
}  
  
(uint v1, uint v2) = arithmetic(10, 20);
```

- 함수의 가시성(Visibility)

가시성이란 함수에 적용되는 상속 또는 외부 호출 가능여부를 나타내는 지시어입니다. 함수에서는 가시성을 생략할 수 없고 상태변수에서 생략하는 경우는 internal이 디폴트입니다. 상태변수를 public으로 선언하면 자동으로 동일한 이름의 상태 변수 읽기 함수가 생성됩니다.

가시성	외부호출	내부호출	상속	상태 변수	함수
external	O	X	O	X	O
public	O	O	O	O	O
internal	X	O	O	O	O
private	X	O	X	O	O

여기서 한 가지 주의할 점은 상태 변수가 private 또는 internal이라고 해서 이더리움에 저장된 데이터를 외부에서 볼 수 없다는 것을 의미하는 것은 아니라는 것입니다. 상태 변수의 가시성은 단지 위에서 설명한 호출과 상속에만 관련되는 것이고 컨트랙트에 저장된 데이터는 스토리지 슬롯에 직접 접근하여 얼마든지 그 값을 볼 수 있으므로 중요한 데이터를 저장하지 않는 것이 바람직합니다.

- 함수의 상태 변경여부(State mutability)

해당 함수가 상태변수를 변경하는 함수인지 지정합니다. 생략해도 되지만 컴파일 경고를 받을 수 있습니다. view나 pure 함수에서 읽기나 변경을 시도하는 경우에는 컴파일 오류가 발생합니다. 상태 접근을 제한하는 정도는 payable, view, pure 순서로 강화됩니다. 지정하지 않은 함수는 "nonpayable"이라고 하기도 합니다.

view	상태 변수 읽기.
pure	상태 변수 읽기와 쓰기 불가
payable	상태(잔액)을 변경(이더 수신)

이더리움에서 상태 변경(트랜잭션)에 해당하는 것은 다음과 같은 것들이 있습니다.

- ✓ 상태변수 쓰기
- ✓ 이벤트 발생하기
- ✓ 컨트랙트 생성하기
- ✓ selfdestruct 호출
- ✓ 이더 송금
- ✓ view나 pure로 지정되지 않은 함수 호출
- ✓ 저수준 호출(low-level call)
- ✓ 상태 변경 opcode를 사용하는 인라인 어셈블리

솔리디티에서는 함수도 function 타입으로 간주하기 때문에 함수를 파라미터로 전달할 수 있습니다. 예를 들면 다음과 같은 것이 가능합니다.

```

function getArrayElementAdd(uint[] memory a1) external pure returns (uint[] memory) {
    return map(a1, fnAdd);
}

function getArrayElementSquare(uint[] memory a2) external pure returns (uint[] memory) {
    return map(a2, fnSquare);
}

function map(uint[] memory arr, function (uint) pure returns (uint) f)
internal pure returns (uint[] memory result) {
    result = new uint[](arr.length);
    for (uint i=0; i<arr.length; i++) {
        result[i] = f(arr[i]);
    }
}

function fnAdd(uint a) internal pure returns (uint) {
    return a+a;
}

function fnSquare(uint a) internal pure returns (uint) {
    return a*a;
}

```

● 함수 호출

상태를 변경하는 트랜잭션의 경우 EOA계정이 시작하기 때문에 처음 설정한 가스(gas limit)가 소모될 때까지 실행됩니다. 이렇게 시작된 트랜잭션은 다른 컨트랙트의 함수를 호출할 수도 있습니다. 보통 컨트랙트 사이에 일어나는 호출은 "internal call"이라고 합니다. 실행 도중 가스가 떨어지면 실행은 중단되고 그동안 변경한 모든 상태들은 다시 원래 상태로 되돌아갑니다.

호출되는 컨트랙트(C)는 C.function(x)와 같은 형태로 함수를 호출합니다. 호출하는 쪽에서는 호출하려는 상대 컨트랙트의 함수 형식을 알아야 합니다. 예를 들어 Caller 컨트랙트가 Callee 컨트랙트의 함수를 호출한다고 하면 아래와 같이 할 수 있습니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Callee {

    function increase(uint256 _v) external pure returns (uint256) {
        return _v * 100;
    }

    function foo(uint256 _x) external pure returns (uint256) {
        return _x;
    }
}

contract Caller {

```

```

Callee callee;

constructor(address _addr) {
    callee = Callee(_addr);
}

function f(uint _val) public view returns (uint256) {
    return callee.increase(_val);
}
}

```

생성자에 Callee 컨트랙트의 주소를 전달해주어야 하므로 당연히 Callee 컨트랙트가 먼저 배포되어야 합니다. 다른 파일에 작성된다면 import를 하거나 함수 정의만 있는 interface를 작성하여 호출할 수 있습니다.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Caller {

    Callee callee;

    constructor(address _addr) {
        callee = Callee(_addr);
    }

    function f(uint _val) public view returns (uint256) {
        return callee.increase(_val);
    }
}

interface Callee {
    function increase(uint256 _v) external pure returns (uint256);
}

```

address 타입의 변수가 컨트랙트 주소일 때 아래와 같이 함수 호출이 가능한 “저수준 호출(low-level call)” 메소드를 사용할 수 있습니다.

call	일반적인 함수 호출
delegatecall	Callee의 함수를 실행하여 Caller의 상태 변수를 변경
staticcall	상태 변경을 하지 않는 함수 호출

저수준 호출에서는 “함수 셀렉터(selector)”와 함수의 인자를 인코딩해서 전달해야 합니다. 또 호출 성공여부와 리턴 값을 받아서 실패한 경우 반드시 revert를 해야 하므로 주의할 필요가 있습니다.

함수 셀렉터는 bytes4 타입의 16진수 문자열인데 예를 들어 함수명이 increase이고 uint256 타입의

인자를 받는다면 아래와 같이 계산할 수 있습니다. 또는 함수 타입의 멤버 메소드인 selector로 쉽게 구할 수도 있습니다.

```
bytes4(keccak256("increase(uint256)")) // 0x30f3f0db
this.increase.selector
```

그런데 솔리디티는 호출 데이터를 쉽게 만들 수 있는 내장 함수를 제공합니다. 예를 들어 위에서 작성한 함수 호출을 저수준 호출로 바꾸면 다음과 같습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Caller {
    address calleeAddr;

    constructor(address _addr) {
        calleeAddr = _addr;
    }

    function f(uint _val) public view returns (uint256) {
        bytes memory payload = abi.encodeWithSelector(
            bytes4(keccak256(bytes("increase(uint256)"))), _val);
        (bool success, bytes memory result) = calleeAddr.call(payload);

        if (!success) {
            revert();
        }
    }
}
```

저수준 호출에서 주의할 점은 call() 호출 후에 리턴되는 bool 타입의 success를 검사해야 한다는 것입니다. 만약 실패의 경우에 revert()를 하지 않으면 정상적으로 트랜잭션이 종료되어 revert된 상태만 되돌아가고 그 전에 발생한 변경들은 반영됩니다. 따라서 실패하면 revert()를 반드시 호출해야 합니다.

abi.encodeWithSelector() 함수는 솔리디티에서 제공하는 인코딩 함수입니다. 함수 셀렉터와 전달인자 값을 받아서 call() 함수에서 필요한 호출 데이터를 생성합니다. 더 간단하게 사용할 수 있는 함수는 abi.encodeWithSignature()입니다. 다음 두 개는 같은 결과를 리턴합니다.

```
abi.encodeWithSelector(bytes4(keccak256(bytes("increase(uint256)"))), _val);
abi.encodeWithSignature("increase(uint256)", _val);
```

저수준 호출에서는 gasLimit을 설정하거나 이더를 함께 보낼 수 있습니다. 아래와 같이 gas와 value 필드에 값을 넣으면 되겠습니다.

```
(bool success, bytes memory result)
= calleeAddr.call{gas: 3e6, value: 5e9}(abi.encodeWithSignature("increase(uint256)", _val));
```

- delegatecall

delegatecall은 일반적으로 라이브러리 컨트랙트의 함수를 호출할 때 사용합니다. 업그레이드 가능한(upgradable) 컨트랙트를 작성할 때도 이용하기도 합니다.

delegatecall을 사용할 때는 조심해야 하는데 왜냐하면 실행 코드는 delegatecall로 호출된 컨트랙트(Callee)에서 실행되지만 데이터는 호출한 컨트랙트(Caller)가 변경되기 때문입니다. 이 경우에 두 컨트랙트의 상태변수들의 배치가 일치해야 잘못된 데이터가 들어가지 않습니다.

다음 예제를 보면서 delegatecall을 사용할 때 주의할 점을 알아보겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Callee {
    uint256 public k;
    uint256 public x;

    function increase(uint256 _v) external {
        k = _v*100;
        x = 2**8;
    }
}

contract Caller {
    uint256 public k;
    uint8 public x;
    address public calleeAddr;

    constructor(address _addr) {
        calleeAddr = _addr;
    }

    function f(uint256 _val) external {
        bytes memory payload = abi.encodeWithSignature("increase(uint256)", _val);
        (bool success, ) = calleeAddr.delegatecall(payload);

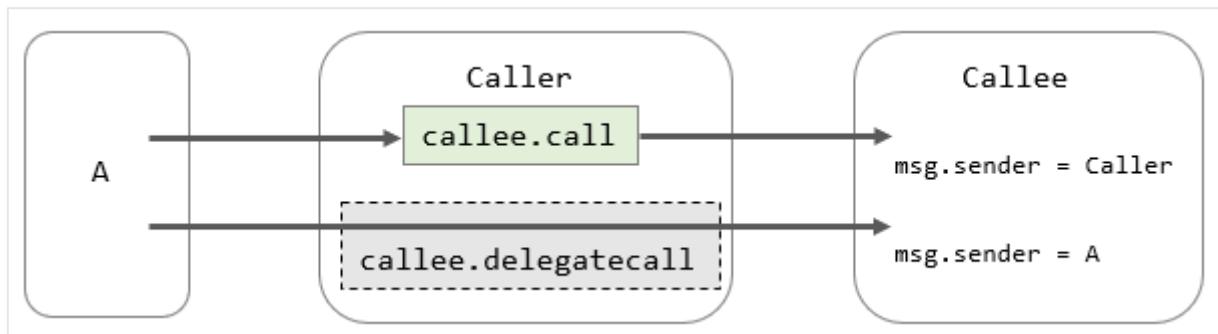
        if (!success) {
            revert();
        }
    }
}
```

Caller 컨트랙트에서 Callee의 increase() 함수를 delegatecall()로 호출했습니다. 이 경우에 Callee 컨트랙트의 increase() 함수가 실행되지만 이 함수가 변경하는 값은 자기 자신의 상태 변수가 아닌 Caller 컨트랙트의 스토리지에 있는 상태 변수입니다. f(100)을 실행한 경우에 delegatecall() 이후의 값은 다음과 같습니다.

컨트랙트	k	x
Callee	0	0
Caller	10000	0

여기서 주목해야 할 점은 Caller의 x 값입니다. Callee의 increase() 함수는 $x = 2^{**8} = 256$ 을 대입했습니다. 하지만 정상적으로 변경된 k 값에 비해 x는 그대로 0입니다. 그런데 Caller의 x는 uint8이고 Callee는 uint256입니다. 상태 변수의 타입이 다르고 256이라는 값은 uint8에 저장될 수 없습니다. 이 경우 오버플로우가 발생하면서 0이 되는 것입니다(2의 보수 연산에 의해 2^{**8} 은 8비트에서 0과 같습니다). Callee가 코드를 실행하지만 자신의 상태변수들은 변경되지 않습니다.

call()과 delegatecall()의 차이점 중 하나는 함수를 호출한 주체인 "msg.sender"가 다르다는 것입니다. call()을 통해서 함수를 호출하는 경우 이것을 호출하는 것은 Caller 컨트랙트 계정입니다. 그러나 Caller가 실행하는 delegatecall()은 Callee에게 코드 실행만을 위임하기 때문에 호출 주체는 원래 Caller를 호출한 계정이 됩니다.



2.8. receive와 fallback

이더리움의 계정들은 서로 이더를 주고받을 수 있습니다. 앞에서 address 타입을 payable로 지정하고 transfer()라는 기본 메소드를 사용하여 이더를 송금한 예제를 기억할 것입니다. 그런데 컨트랙트가 임의의 계정으로부터 이더를 받기 위해서는 특별한 함수 receive()가 필요합니다.

컨트랙트에 receive()라는 함수가 작성되어 있으면 그 컨트랙트에 이더를 송금할 수 있습니다. receive()는 앞에 function이라는 키워드 없이 정의됩니다.

```
receive() external payable {}
```

receive()는 일반적인 함수가 아니기 때문에 몇 가지 규칙이 있습니다.

- ✓ external payable로 선언해야 합니다.
- ✓ 전달인자와 리턴 값을 가질 수 없습니다.
- ✓ 데이터(msg.data)를 받을 수 없습니다.
- ✓ 함수 내에서 사용할 수 있는 가스는 2300 가스로 제한됩니다(stipend).

receive()는 EOA가 송금하거나 컨트랙트에서 transfer() 또는 send()를 사용하여 이더를 보낼 때 실행됩니다. 다만 이스탄불 하드포크(2019년 12월)에서 제기된 문제점, 즉 opcode의 가스 소비량은 언제든 조정될 수 있으므로 가스 제한을 하는 transfer()와 send()의 사용을 자제하는 것이 좋습니다. 후반부의 보안 코딩에서 자세하게 다룰 것입니다.

receive()와 유사한 함수로 fallback()이 있습니다. receive()가 이더 수신 전용이라면 fallback()은 다른 목적으로 사용하는 함수입니다. 두 가지 형태가 있습니다.

```
fallback() external {}

fallback(bytes calldata _input) external returns (bytes memory _output) {}
```

receive()와의 차이점은 bytes 타입의 데이터를 전달받고 리턴 값을 가질 수도 있다는 것입니다. 이더를 받는 용도로 사용하려면 payable을 지정해도 되지만 이 경우에는 receive()를 쓸 것을 권장하고 있습니다. fallback payable은 receive와 마찬가지로 2300 가스 제한이 있습니다.

fallback()은 컨트랙트에 작성되어 있지 않은 함수가 호출된 경우, 말 그대로 "fallback"으로 실행되는 함수입니다. 예를 들어 다음과 같이 호출에서 fallback이 있으면 fallback 함수가 실행됩니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Caller {

    function f() external {

        address callee = 0x0FC5025C764cE34df352757e82f7B5c4Df39A836;
```

```

        bytes memory payload = abi.encode("Hello", 1000);

        (bool success, ) = callee.call(payload);

        if (!success) {
            revert();
        }
    }

contract Callee {
    bytes public data;
    string public s;
    uint256 public v;

    fallback(bytes calldata _input) external returns (bytes memory) {
        data = _input;
        (s, v) = abi.decode(_input, (string, uint256));
        return _input;
    }
}

```

2.9. modifier

함수의 실행을 조건에 따라 제한할 필요할 때 쓸 수 있는 modifier가 있습니다. 흔히 말하는 컨트랙트의 접근 제어(Access Control)의 용도로 사용합니다. 예를 들어 다음과 같이 modifier를 정의하고 여러 함수에 적용할 수 있습니다.

```

modifier onlyOwner {
    require (msg.sender == owner, "Only owner can call this function.");
    _;
}

function f() external onlyOwner { ... }

function g() external { ... }

function h() external onlyOwner { ... }

```

onlyOwner라는 modifier는 require의 조건 검사 후에 “_” 위치에서 함수를 실행합니다. 이 경우에는 msg.sender == owner가 true일 때만 함수 f()와 h()를 실행할 것입니다.

modifier는 한 개 이상이 적용할 수 있습니다. 여러 개가 있으면 순서대로 적용되어 모두 true일 때만 함수를 실행합니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

```

```

address owner;
uint256 public v;

constructor() {
    owner = msg.sender;
}

modifier myModifier1(address _addr) {
    require(_addr == owner, "Only owner");
    _;
}

modifier myModifier2(uint256 _v) {
    require(_v <= 10000, "Less than 10K");
    _;
}

function foo(uint256 _v) public myModifier1(msg.sender) myModifier2(_v) {
    v = _v;
}
}

```

2.10. assert, require, revert

조건의 true, false 여부를 검사하여 예외(exception)을 발생시키는 구문으로 assert와 require가 있습니다. 형식은 다음과 같습니다.

```

assert(<condition>);

require(<condition>, <revert reason>);

```

assert는 내부적인 오류를 검사할 때 사용합니다. 예를 들어 분모를 0하는 나눗셈과 같이 반드시 지켜져야 하는 규칙을 확인할 때 쓸 수 있습니다. assert로 조건식의 결과가 false라면 Panic(uint256)이라는 예외가 "0x01"을 리턴합니다. Panic(uint256)의 예외코드는 아래에서 확인할 수 있습니다.

<https://docs.soliditylang.org/en/v0.8.10/control-structures.html#panic-via-assert-and-error-via-require>

외부에서 컨트랙트를 호출할 때 전달되는 값을 검사하는 경우에는 보통 require가 많이 사용됩니다. require는 두 번째 파라미터로 오류 발생 원인 "revert reason"을 줄 수 있습니다. 앞서 modifier 예제에서 본 것처럼 다음과 같이 사용할 수 있습니다.

```

require (msg.sender == owner, "Only owner can call this function.");

```

require가 발생시키는 예외는 Error(string)입니다. revert reason에 넣은 문자열이 예외처리 구문으로 전달됩니다. assert와 require는 예외가 발생하면 그동안 발생했던 상태 변경을 모두 되돌리고 남은 가스를 반환합니다.

revert는 명시적으로 코드에서 예외를 발생시킬 때 사용합니다. require와 동일한 기능을 하도록 다음과 같이 작성할 수 있겠습니다. revert 역시 Error(string) 예외를 발생시킵니다.

```
if (msg.sender != owner) {
    revert("Only owner can call this function.");
}
```

솔리디티 0.8.4부터는 사용자 정의 예외를 만드는 것이 가능합니다. 이렇게 만든 예외를 발생시키려면 revert를 사용합니다. 다음 코드처럼 error Unauthorized()라는 예외를 정의하면 어떤 조건을 검사하고 revert로 예외를 발생시키면 되겠습니다.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

error Unauthorized();

contract VendingMachine {
    address payable owner = payable(msg.sender);

    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        owner.transfer(address(this).balance);
    }
    // ...
}
```

2.11. 예외처리(try-catch)

try-catch는 솔리디티 0.6.0부터 도입된 예외처리 구문입니다. 다른 프로그래밍 언어와 마찬가지로 코드 실행 중에 예외가 발생했을 때 프로그램을 중단하지 않고 준비된 예외처리를 실행할 때 사용할 수 있습니다.

현재 0.8.10에서는 try-catch를 외부 호출에만 적용할 수 있습니다. 아래 예제를 보도록 하겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity 0.8.10;

contract Callee {
```

```

uint[5] arr = [1,2,3,4,5];

function print(uint _index) public view returns (uint) {
    require(_index < arr.length, "index out of bound");
    return arr[_index];
}

contract Caller {

    Callee callee;
    string public errData;

    constructor(address _addr) {
        callee = Callee(_addr);
    }

    function get(uint _index) public returns (uint, bool) {

        try callee.print(_index) returns (uint v) {
            return (v, true);

        } catch Error(string memory _err) {
            errData = _err;
            return (0, false);
        }
    }
}

```

try 다음에 callee.print()라는 외부 컨트랙트의 external 함수를 호출했습니다. 문제가 없다면 try{...} 내의 코드가 실행되어 리턴이 될 것이고 callee.print()의 require에 의해 예외가 발생한다면 catch{...} 블록이 실행될 것입니다. 여기서는 revert reason을 상태 변수에 저장하는 일을 수행합니다. Callee 컨트랙트에서 require()에서 예외가 발생하면 catch에서는 Error(string)으로 받게 됩니다.

Panic(uint256)도 같은 식으로 예외처리를 할 수 있습니다. 아래 예시처럼 0으로 나눗셈을 하는 경우에 Panic(uint256) 예외가 발생하므로 catch에서는 Panic(uint256)을 잡아야 합니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Callee {

    function f(uint256 _v) external pure returns (uint256) {
        return 10000000 / _v;
    }
}

contract Caller {

    Callee callee;
    uint256 public errorCode;

    constructor(address _addr) {
        callee = Callee(_addr);
    }
}

```

```

function foo(uint256 _v) public returns (uint256, bool) {

    try callee.f(_v) returns (uint256 result) {
        return (result, true);
    } catch Panic(uint code) {
        errorCode = code;
        return (0, false);
    }
}

```

2.12. 이벤트(event)

이벤트는 트랜잭션 처리 히스토리를 블록체인에 저장할 때 사용합니다. 그래서 이벤트 “로그(log)”라고 부르기도 합니다. 트랜잭션의 최종 결과는 “영수증”에 해당하는 리시트(receipt)로 만들어지는데, 이벤트는 리시트의 로그 항목에 기록되므로 과거의 기록을 조회할 때도 이벤트 로그를 이용할 수 있습니다.

예를 들어 컨트랙트가 받은 이더의 로그를 기록하고 싶다면 다음과 같이 할 수 있습니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    address owner;
    event Deposit(address _from, uint _value);

    constructor() {
        owner = msg.sender;
    }

    receive() external payable {
        emit Deposit(msg.sender, msg.value);
    }

    function disable() external {
        selfdestruct(payable(owner));
    }
}

```

`Deposit`이라는 이벤트를 정의하고 이벤트가 발생했을 때 `address` 타입의 `_from`과 `uint256` 타입의 `_value`를 로그에 남깁니다. `_from`은 `msg.sender`이므로 이 컨트랙트에 이더를 송금한 계정이며 `_value`는 `msg.value` 즉 송금된 이더의 수량입니다. 컨트랙트에 이더가 송금될 때마다 이벤트가 발생하여 로그가 블록에 기록될 것입니다.

이벤트가 발생하는 곳은 `receive` 함수입니다. 이벤트를 발생시킬 때는 `emit`이라는 키워드를 사용합니

다. 리믹스 콘솔에서 이벤트 로그를 볼 수 있습니다.

```
[  
 {  
   "from": "0xD4Fc541236927E2EAf8F27606bD7309C1Fc2cbee",  
   "topic": "0xe1fffcc4923d04b559f4d29a8bfc6cda04eb5b0d3c460751c2402c5c5cc9109c",  
   "event": "Deposit",  
   "args": {  
     "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  
     "1": "20000000000000000000",  
     "_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  
     "_value": "20000000000000000000"  
   }  
 }]  
 ]
```

topic에 있는 값은 이벤트 시그너처에 해당하는 keccak256("Deposit(address,uint256)")입니다. args에 해당 이벤트가 발생했을 때 전달된 from과 value 값이 저장된 것을 알 수 있습니다.

이벤트를 정의할 때 전달하는 변수를 `indexed`로 설정하면 발생된 이벤트 중에 특정한 조건에 맞는 것을 필터링할 수 있습니다. 앞의 예제에서 `_value`에 `indexed`를 붙이는 경우에는 특정 수량을 보낸 이벤트를 필터링할 수 있게 됩니다(최대 3개까지 `indexed` 가능).

```
event Deposit(address _from, uint indexed _value);
```

예를 들어 100 gwei를 송금하는 경우에 이벤트 로그는 다음과 같이 생성됩니다. 아래 그림은 대표적인 이더리움 블록 탐색기인 이더스캔(<https://etherscan.io/>)에서 로그를 조회한 것입니다.

Topics의 첫 번째 값은 이벤트 시그너처이고 indexed가 붙은 _value의 값은 두 번째에 저장됩니다. 이 값을 디코딩하면 100 gwei가 됩니다(디코딩할 때 리믹스에서 web3.eth.abi.decodeParameter를 이용). indexed가 없는 from은 Data에 저장됩니다.

애플리케이션에서는 다음과 같이 필터링해서 조회할 수 있습니다. 리믹스는 Ethers.js에서 제공하는 함수를 사용할 수 있습니다. 이더리움의 테스트넷을 JsonRpcProvider()로 지정하고 getLogs()를 이용하여 과거의 이벤트 로그를 가져올 수 있습니다. getLogs()에 필터를 정의하여 넘겨주면 해당 이벤트 로그를 찾게 됩니다.

자바스크립트는 파일명에서 오른쪽 클릭하고 Run을 선택하면 실행됩니다.

```
const provider = new ethers.providers.JsonRpcProvider("https://eth-ropsten.alchemyapi.io/v2/..");

const filter = {address: "0x1F4189e2E745b341f73664039557644ba1206AE8",
    topics: [null, "0x000000000000...000000000000174876e800"],
    fromBlock: "earliest"};

(async () => {
    const logs = await provider.getLogs(filter);
    console.log(logs);

})();
```

실행 결과를 콘솔에서 확인하면 다음과 같습니다.

2.13. 상속과 인터페이스

상속은 객체지향 언어의 특징 중 하나입니다. 상속을 통해 이미 잘 만들어진 다른 컨트랙트의 기능을 그대로 사용하거나 확장할 수 있습니다. 또 인터페이스를 정의하고 여러 방법으로 구현할 수 있도록 하여 컨트랙트의 “다형성(polymorphism)”을 지원하고 다중 상속도 가능합니다.

예를 들어 컨트랙트를 작성하면서 관리자 기능이 필요하다면 직접 구현해도 되지만 누군가 잘 만들어 놓은 Ownable이라는 컨트랙트가 있다면 이것을 상속받을 수 것입니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Ownable {

    address private owner;

    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    constructor () {
        owner = msg.sender;
    }

    function isOwner() public view returns (bool) {
        return msg.sender == owner;
    }
}
```

이 컨트랙트를 내가 작성하는 MyContract에서 상속을 받으려면 다음과 같이 합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "./Ownable.sol";

contract MyContract is Ownable {

    construct() {
        ...
    }

    function foo() public {
        ...
    }
}
```

상속은 is라는 키워드를 사용합니다. is 다음에 상속을 받는 부모 컨트랙트의 이름을 적어주면 되겠습

니다. 컴파일이나 배포를 할 때는 자식 컨트랙트만 하면 됩니다.

함수, modifier, 상태 변수들이 상속되는데 함수의 가시성(Visibility)을 설명할 때 이야기한 것처럼 이 것에 따라 상속의 가능 여부가 결정됩니다. `private`으로 정의한 함수는 상속할 수 없고 동일한 이름의 상태 변수를 자식 컨트랙트에서 선언할 수 없습니다(`private`은 제외).

가시성	상속
<code>external</code>	O
<code>public</code>	O
<code>internal</code>	O
<code>private</code>	X

상속된 함수나 modifier를 자식 컨트랙트에서 변경하고 싶은 경우도 있을 것입니다. 이런 경우에는 재정의(override)를 할 수 있습니다. 재정의를 하기 위해서는 부모 컨트랙트에서 그 함수를 `virtual`로 선언해야 하고 자식 컨트랙트에서는 `override`를 붙여야 합니다.

또 재정의하면서 상태 변경 여부(State mutability)를 바꿀 수도 있는데 상태 접근을 제한하는 순서에 준합니다. 예를 들어 “`nonpayable`(아무것도 지정되지 않은)”을 `view`나 `pure`로 변경할 수 있습니다. 그러나 `payable`은 바꿀 수 없습니다.

함수의 가시성은 `external`에서 `public`으로 수정하는 것이 허용됩니다. 부모 함수를 다시 호출하려면 `super`를 사용하면 되겠습니다.

아래 예제를 보겠습니다. 부모 함수가 “`nonpayable`”이지만 상속받은 컨트랙트에서는 `pure` 함수로 변경할 수 있습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Parent {
    function foo() virtual external {}
}

contract Child is Parent{
    function foo() override public pure {}
}
```

다른 함수에서 재정의된 함수를 다시 상속받아서 재정의할 수도 있습니다. 이 경우에도 `virtual`로 선언되어야 합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract GrandFather {
    function foo() virtual public {}
}

contract Father is GrandFather {
    uint256 public b;
    function foo() override virtual public {
        b = b + 100;
    }
}

contract Child is Father {
    uint256 public c;
    function foo() override public {
        c = c + 1000;
    }
}
```

솔리디티는 다중 상속이 가능하기 때문에 여러 개의 컨트랙트를 상속받을 수 있습니다. 만약 부모 컨트랙트들이 동일한 이름의 함수를 둘 다 가지고 있다면 이것을 재정의할 때 override 다음에 부모 컨트랙트를 모두 명시해야 합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Parent1 {
    function foo() virtual public {...}
}

contract Parent2 {
    function foo() virtual public {...}
}

contract Child is Parent1, Parent2 {
    function foo() override(Parent1, Parent2) public view {}
}
```

부모 컨트랙트의 생성자가 파라미터를 받아야 하는 경우에는 자식 컨트랙트에서 넣어주면 됩니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Parent {

    uint256 private a;
    constructor(uint256 _a) {
        a = _a;
    }
}
```

```

        function get() internal view returns (uint256) {
            return a;
        }
    }

contract Child is Parent(100) {

    function foo() public view returns (uint256) {
        return get();
    }
}

```

초기화할 때 정해지는 상수가 아니라면 자식 컨트랙트의 생성자를 통해 값을 전달할 수 있습니다.
constructor() 옆에 부모 컨트랙트를 써주고 값을 전달해주면 되겠습니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Parent {

    uint256 private a;
    constructor(uint256 _a) {
        a = _a;
    }

    function get() internal view returns (uint256) {
        return a;
    }
}

contract Child is Parent {

    constructor(uint256 _v) Parent(_v){}

    function foo() public view returns (uint256) {
        return get();
    }
}

```

다중 상속의 경우는 `is` 키워드의 가까운 쪽부터 오른쪽으로 방향으로 차례로 생성자가 실행됩니다.
예를 들어 다음과 같은 경우에 Father 컨트랙트의 생성자가 먼저 실행된 후 Mother가 실행됩니다.
constructor()에 지정한 순서와는 상관없습니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Father {
    constructor() { }
}

contract Mother {
    constructor() { }
}

```

```
}

contract Child is Father, Mother {
    constructor() Mother() Father() { }
}
```

modifier도 재정의가 가능합니다.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract Parent {
    modifier foo() virtual {_;}
}

contract Child is Parent {
    modifier foo() override {_;}
}
```

- 추상(abstract) 컨트랙트

컨트랙트 앞에 abstract가 붙으면 그 컨트랙트는 추상 컨트랙트가 됩니다. 하나라도 구현부가 없는 함수가 존재하면 abstract를 반드시 붙여야 합니다. 당연히 구현되지 않은 함수는 상속을 통해서 구현되어야 하기 때문에 virtual로 정의합니다. 모두 구현된 함수들만 있어도 abstract로 지정할 수 있습니다.

추상 컨트랙트는 단독으로 배포될 수 없고 상속을 위해서 작성되는 경우가 많습니다.

- 인터페이스

인터페이스는 추상 컨트랙트와 유사하지만 모든 함수들의 구현부가 없다는 점이 다릅니다. 그리고 contract 대신에 interface라는 키워드를 사용합니다. 상속과 마찬가지로 is 키워드를 사용하여 구현 컨트랙트를 작성하면 됩니다.

인터페이스는 상태 변수와 modifier를 가질 수 없고 생성자도 당연히 없습니다. 그리고 함수들은 모두 external로 정의되어야 합니다. 예를 들어 ERC-20이라는 토큰 표준은 다음과 같은 인터페이스로 정의되어 있습니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
interface IERC20 {

    function name() public view returns (string); // optional
    function symbol() public view returns (string); //optional
    function decimals() public view returns (uint8); // optional

    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

NFT 토큰 표준인 ERC-721과 ERC-1155도 인터페이스로 정의되어 있습니다. NFT 표준 인터페이스는 다음 3장에서 자세히 살펴보겠습니다.

2.14. 기타

- 예약어와 내장함수

솔리디티의 예약어와 내장함수는 다음과 같은 것들이 있습니다. 전체 목록은 아래 링크를 참고하기 바랍니다.

<https://docs.soliditylang.org/en/v0.8.10/units-and-global-variables.html>

이더 단위	ether, gwei, wei	
시간 단위	seconds, minutes, hours, days, weeks	
블록 정보	block.number	현재 블록번호
	block.timestamp	블록 타임스탬프
	block.gaslimit	블록 가스한도
	block.basefee	현재 블록의 기본 수수료
	block.coinbase	채굴보상 계정
	gasleft()	남은 가스량
트랜잭션 정보	msg.sender	함수(트랜잭션) 호출 계정
	msg.value	송금된 이더 수량
	msg.data	함수호출 데이터
	tx.origin	트랜잭션 시작 계정(EOA)
ABI 인코딩, 디코딩 해시 함수	abi.decode	ABI 인코딩
	abi.encode	ABI 디코딩
	abi.encodeWithSelector	함수 셀렉터와 전달인자 인코딩
	abi.encodeWithSignature	함수 셀렉터와 전달인자 인코딩
	abi.encodePacked	ABI 인코딩(Packed)
	keccak256	해시 함수
	sha256	해시 함수
	ripemd160	해시 함수
	ecrecover	전자서명검증

- 라이브러리

다른 프로그래밍 언어와 마찬가지로 솔리디티로 라이브러리를 만들 수 있습니다. 라이브러리는 contract 대신 library라는 키워드를 사용합니다. 예를 들어 오픈제펠린의 SafeMath라는 라이브러리 컨트랙트는 다음과 같이 되어 있습니다.

```
pragma solidity ^0.5.0;
```

```

library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0, "SafeMath: division by zero");
        uint256 c = a / b;
        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}

```

라이브러리를 컨트랙트에서 사용하는 방법은 두 가지가 있습니다. 예를 들어 위의 SafeMath 라이브러리를 다음과 같이 쓸 수 있습니다.

```

pragma solidity ^0.5.0;

import "./SafeMath.sol";

contract MyContract {

    function foo(uint _a, uint _b) external pure returns (uint256) {
        return SafeMath.add(_a, _b);
    }
}

```

두 번째 방식은 using for 구문을 사용하는 것입니다. using for는 라이브러리를 특정 데이터 타입에만

적용해서 마치 그 타입이 제공하는 기본 메소드처럼 사용할 수 있게 합니다. 위의 예제를 using for 구문으로 변경하면 아래와 같습니다.

```
pragma solidity ^0.5.0;

import "./SafeMath.sol";

contract MyContract {
    using SafeMath for uint256;

    function foo(uint _a, uint _b) external pure returns (uint256) {
        return _a.add(_b);
    }
}
```

라이브러리는 상태 변수를 가질 수 없지만 구조체 타입을 정의할 수는 있습니다. 다음 예제를 보겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

library LibUser {

    struct UserInfo {
        uint256 userId;
        string userName;
        uint256 regDate;
    }

    function addUser(UserInfo storage _self,
                    uint256 _userId,
                    string memory _userName) internal {
        _self.userId = _userId;
        _self.userName = _userName;
        _self.regDate = block.timestamp;
    }
}
```

라이브러리 컨트랙트 LibUser에서 구조체인 UserInfo를 정의했습니다. addUser()라는 함수의 첫 번째 전달인자로 UserInfo 타입의 _self를 정의했습니다. 이것은 addUser()를 호출할 때 전달해주는 파라미터가 아니라 using for 구문을 UserInfo 타입의 변수에 적용할 때 그 변수를 가리키는 참조 변수입니다. 이 라이브러리를 사용하는 코드를 보겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "./MyLib.sol";
```

```

contract MyContract {

    LibUser.UserInfo user;
    mapping (uint256 => LibUser.UserInfo) users;

    using LibUser for LibUser.UserInfo;

    function setUser(uint256 _userId, string memory _userName) external {
        user.addUser(_userId, _userName);
        users[_userId] = user;
    }

    function getUser(uint256 _userId) public view returns (LibUser.UserInfo memory) {
        return users[_userId];
    }

}

```

using for 구문으로 라이브러리에 정의된 구조체 UserInfo 타입을 지정했습니다. setUser() 함수에서는 UserInfo 타입의 변수 user에서 addUser() 함수를 호출합니다. 그런데 첫 번째 파라미터가 빠져 있습니다. 왜냐하면 그것은 라이브러리를 호출한 MyContract의 상태변수인 user를 가리키는 참조이기 때문입니다. 라이브러리에서 제공하는 모든 함수를 UserInfo 타입에 적용되도록 했고 특히 구조체와 같은 레퍼런스 타입의 경우는 첫 번째 파라미터를 그 타입을 가리키도록 지정합니다.

- selfdestruct

스마트 컨트랙트가 이더리움에 배포되면 수정이나 삭제를 할 수 없습니다. 하지만 컨트랙트 기능을 영구 정지할 수 있는 방법이 있는데 그것이 selfdestruct()입니다. 이 코드를 넣을 때는 modifier를 이용하여 관리자와 같은 특정 계정만이 실행할 수 있도록 해야 합니다.

selfdestruct()를 호출할 때 전달해주는 값은 계정 주소인데 컨트랙트가 중지되기 전에 컨트랙트가 소유한 이더를 그 계정으로 전송합니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {

    receive() external payable {}

    function kill() external {
        selfdestruct(payable(0x5B38...ddC4));
    }
}

```

2.15. 보안 코딩

스마트 컨트랙트는 화폐로 교환 가능한 가상 자산을 거래하는 경우가 많기 때문에 보안에 신경을 써야 합니다. 더구나 이더리움에 한 번 배포된 컨트랙트의 코드는 일반적으로 수정할 수 없으므로 문제점이 발견되는 경우는 매우 큰 피해를 입을 수도 있습니다.

실제로 이더리움에서는 지금도 보안 사고들이 종종 발생하고 있는데 대부분은 스마트 컨트랙트의 취약점을 이용하는 해킹 사고들입니다. 특히 2016년에 발생한 "The DAO" 해킹으로 인하여 수백억에 달하는 자산이 탈취당했고 결과적으로 이더리움 클래식(ETC)과 현재 이더리움(ETH)으로 분열되기도 했습니다.

그래서 스마트 컨트랙트를 기반으로 하는 블록체인 서비스들은 컨트랙트의 보안 취약점을 다양한 방법으로 검사하고 분석해주는 "감사(audit)"를 필수적으로 받게 되고 이러한 일을 전문적으로 해주는 기업들도 있습니다(4장에 나오는 오픈제펠린이 그런 기업들 중 한 곳입니다). 이제부터 솔리디티로 컨트랙트를 작성할 때 주의해야 할 점을 살펴보도록 하겠습니다.

2.15.1. 컨트랙트 중단 함수

프로그램을 아무리 잘 만든다고 해도 예상치 못한 곳에서 문제가 발생할 수 있습니다. 완벽하게 보이는 소프트웨어도 아직 드러나지 않은 버그들이 있다고 생각해야 합니다. 특히 이더리움과 같은 퍼블릭 블록체인에서 보안 사고가 터지면 블록체인의 불변성 때문에, 미리 대비해 놓지 않으면 손쓸 수 없는 상황을 지켜볼 수밖에 없습니다.

만약 심각한 문제를 발견한 경우 이를 분석하여 해결책을 찾기 전까지 컨트랙트의 기능을 일시적으로 중지시켜야 할 필요가 있을 것입니다. 즉 보안 사고가 발생했을 때는 컨트랙트의 주요한 함수들이 호출되더라도 실행되지 않도록 하는 장치를 마련해 두는 것이 보안 측면에서 필요합니다.

앞에서 배운 modifier를 적절하게 활용하여 특정 조건(플래그)이 설정되었을 때 함수의 실행을 막는 코드를 미리 작성해 놓는 것이 바람직합니다. 또는 아예 컨트랙트 전체를 비활성화하고 컨트랙트가 보유한 자산을 안전한 계정으로 이체시킬 수 있는 selfdestruct() 함수도 필요할 수 있습니다.

물론 이러한 함수들의 실행은 특정 계정, 권한을 가진 계정만이 실행할 수 있도록 해야 합니다. 예를 들어 다음과 같은 modifier를 만들 수 있겠습니다(오픈제펠린 Pausable 컨트랙트에서 발췌).

```
modifier whenNotPaused() {
    require(!paused(), "Pausable: paused");
}
```

이름에서 알 수 있는 것처럼 정상적으로 컨트랙트의 함수가 실행될 수 있는지 검사하는 modifier입니다. 이것이 적용된 함수는 paused()가 false 즉 중지되지 않은 경우에만 함수를 실행할 것입니다. 반대로 아래와 같이 중지된 경우에만 실행되는 함수도 있을 것입니다.

```
modifier whenPaused() {
    require(paused(), "Pausable: not paused");
    _;
}
```

예를 들어 어떤 자산을 전송하는 transfer()라는 함수가 있다면 다음과 같이 modifier를 적용할 수 있습니다. 비상 상황이 발생하면 paused()가 true가 되도록 하여 transfer() 함수의 실행을 막을 수 있게 되는 것입니다.

```
function transfer(address to, uint256 value) external whenNotPaused {
    _transfer(to, value);
}
```

2.15.2. 권한과 역할

컨트랙트 중단 함수처럼 함수의 기능을 정지하는 것은 상당히 중요한 일입니다. 특정 권한을 가지는 계정만이 실행할 수 있어야 합니다. 컨트랙트 함수를 실행할 수 있는 계정(또는 계정들의 그룹)을 지정하는 것도 컨트랙트 보안에 매우 필수적입니다.

가장 흔하게 사용되는 방법은 컨트랙트가 배포되는 시점 다시 말해서 컨트랙트의 constructor()가 실행되는 시점에 배포 계정을 저장하고 modifier를 이용하여 함수를 호출하는 계정과 비교하여 실행을 제한하는 것입니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(owner == msg.sender, "Caller is not the owner");
        _;
    }
}
```

```

    }

    function transferOwnership(address newOwner) public onlyOwner {
        ...
    }
}

```

상태변수 owner에 배포 계정을 저장하고 modifier onlyOwner를 정의했습니다. onlyOwner는 다시 transferOwnership() 함수에 적용되어 권한을 이전할 때 반드시 현재 권한을 가지고 있는 계정만이 실행할 수 있도록 만들었습니다.

권한과 역할에 대해서는 4장 오픈제펠린 AccessControl 컨트랙트에서 좀 더 자세히 다루게 될 것입니다.

2.15.3. 인출(Withdrawal) 패턴

스마트 컨트랙트에 자산을 예치할 수 있다면 당연히 인출이 가능하도록 작성해야 합니다. receive() 함수에서 설명한 것처럼 이더를 받기만 하고 인출하는 코드가 없다면 대형 보안 사고가 될 것입니다. 그런데 인출 함수를 작성할 때도 주의해야 점들이 있습니다.

사용자가 컨트랙트에 자신의 자산을 예치했다가 나중에 다시 인출하는 경우 직접 인출하게 하는 방식인 “pull” 패턴을 쓰는 것이 바람직합니다. 이와는 반대로 컨트랙트 관리자가 확인 후에 어떤 함수를 실행하여 일괄적으로 자산을 분배하는 형태가 있는데 이것을 “push” 패턴이라고 합니다.

push 패턴은 사용자들에게 이더와 같은 자산을 전송해주는 함수를 작성하는 것입니다. 예를 들어 아래와 같은 형태가 될 것입니다.

```

function transfer() public onlyOwner {
    for (uint i=0; i<list.length; i++) {
        address payable addr = payable(list[i]);
        if (addr != owner && vault[addr]>0) {
            addr.transfer(vault[addr]);
        }
    }
}

```

위의 함수는 반복문 안에서 list 배열에 담긴 계정 주소로 이더를 전송하고 있습니다. 이 경우에 트랜잭션(송금)에 필요한 가스는 21,000 이상 소요되므로 1000개 이상의 계정에 이더를 전송해야 한다면

블록 가스 제한(block gas limit)에 걸려서 중간에 실행이 중단될 가능성이 있습니다. 더구나 이 함수가 컨트랙트로부터 자산을 인출할 수 있는 유일한 함수라면 예치된 이더는 영원히 컨트랙트에 묶이게 됩니다.

반복문 안에서 상태를 변경시키는 함수를 실행하는 경우는 무척 조심해야 합니다. 그래서 위험을 줄이기 위해 자산 인출은 사용자가 스스로 각자 인출하는 패턴을 권장합니다. 다음과 같은 인출 메소드를 사용자들이 실행할 수 있도록 제공하는 것입니다.

```
function withdraw() external {
    require(msg.sender != address(0));
    require(vault[msg.sender]>0);

    uint256 amount = vault[msg.sender];
    vault[msg.sender] = 0;
    payable(msg.sender).transfer(amount);
}
```

반복문이 아니더라도 push 패턴을 악용하는 사례가 있기 때문에 조심해야 합니다. 예를 들어 다음과 같은 코드를 살펴보겠습니다(오픈제펠린의 "Forever King"에서 발췌).

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract King {
    address payable king;
    uint public prize;
    address payable public owner;

    constructor() payable {
        owner = payable(msg.sender);
        king = payable(msg.sender);
        prize = msg.value;
    }

    receive() external payable {
        require(msg.value >= prize || msg.sender == owner);
        king.transfer(msg.value);
        king = payable(msg.sender);
        prize = msg.value;
    }

    function king() public view returns (address payable) {
        return king;
    }
}
```

이 컨트랙트는 receive() 함수를 통해서 이더를 받을 때 더 많은 이더를 전송하는 계정을 상태 변수 king에 저장하고 이전에 king이었던 계정이 예치했던 이더를 돌려줍니다. 다시 말해서 현재 prize 변수의 값보다 큰 수량의 이더를 보낸 계정을 새로운 king으로 삼는 것입니다.

그런데 이 컨트랙트를 무용지물로 만드는 방법이 있습니다. 아무리 현재 prize보다 더 많은 이더를 전송해도 king이 바뀌지 않고 이전 계정을 영원히 king으로 남게 하는 것입니다.

아래 ForeverKing 컨트랙트를 작성해서 King 컨트랙트로 이더를 송금하겠습니다. ForeverKing 컨트랙트를 배포할 때 King 컨트랙트의 주소와 현재 King 컨트랙트의 prize 값보다 충분히 큰 이더를 함께 전송합니다(나중에 King으로 이더를 보내기 위해).

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract King {
    ...
}

contract ForeverKing {
    King public target;

    constructor(address payable _addr) payable {
        target = King(_addr);
    }

    function send(uint256 _v) external {
        (bool bOk, ) = address(target).call{value: _v}("");
        if (!bOk) {
            revert();
        }
    }
}
```

send() 함수를 호출하면 King 컨트랙트로 이더가 전송됩니다. 전달하는 value 값은 현재 prize에 있는 값보다 크게 해야 합니다(wei 단위로 환산하여 넘겨주어야 합니다). King 컨트랙트의 receive() 로직을 보면 prize 보다 클 때만 king을 교체하기 때문입니다.

ForeverKing 컨트랙트로 king이 교체된 후에는 다른 계정이 현재 prize 값보다 큰 수량의 이더를 보내더라도 ForeverKing을 대체할 수 없습니다. 이유가 무엇일까요?

receive() 로직을 보면 이전 계정에 transfer() 함수를 호출하여 이더를 전송하는 것을 볼 수 있습니다. 그런데 ForeverKing에는 이더를 받는 함수가 없습니다. 의도적으로 만들지 않은 것입니다. 따라서 transfer() 함수에서 오류가 발생하면서 king을 교체하는 로직은 항상 실패하게 됩니다. 결국 king은 영원히 ForeverKing 컨트랙트가 되는 것입니다.

이렇게 직접 push 패턴으로 이더를 전송하는 경우에는 조심해야 할 점들이 있습니다. 다음에 설명할 재진입 문제도 유사한 맥락에 있습니다.

2.15.4. 재진입(Reentrancy) 문제

address 타입이 제공하는 transfer()와 send() 함수로 이더를 전송하는 경우는 수신하는 컨트랙트의 receive() 함수 내에서 가용한 가스는 2300 이내라는 제약이 있습니다. 이런 제약 조건은 소위 말하는 “재진입” 공격에 대한 방어 수단이 됩니다. 왜냐하면 2300 가스로는 아래와 같은 일들을 할 수 없기 때문입니다.

- ✓ 이더 보내기
- ✓ 상태 변수에 값을 저장하기
- ✓ 다른 컨트랙트의 함수 호출하기

transfer()와 send()를 쓰지 않고 이더를 보낼 수 있는 방법은 저수준 호출 call()을 사용하는 것인데 이 경우에는 재진입(Reentrancy)이라는 보안 문제가 생길 수 있습니다. 다음 컨트랙트를 통해서 재진입이 어떻게 발생하는지 살펴보겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Donation {
    mapping(address => uint256) balances;

    function donate(address _to) public payable {
        balances[_to] += msg.value;
    }

    function balanceOf(address _addr) public view returns (uint balance) {
        return balances[_addr];
    }

    function withdraw(uint256 _amount) public {
        if (balances[msg.sender] >= _amount) {
            (bool b0k,) = msg.sender.call{value: _amount}("");
            if (!b0k) {
                revert();
            }
            unchecked{ balances[msg.sender] -= _amount; }
        }
    }
}
```

```

    }

    receive() external payable {}

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}

```

donate()를 호출하여 특정 계정에 이더를 기부하면 기부를 받은 계정은 자신에게 기부된 이더를 withdraw() 함수를 호출하여 인출할 수 있습니다. 기부를 받는 계정들과 기부 수량은 mapping 타입의 balances 장부에 기록이 되므로 받은 수량 이내에서만 인출할 수 있습니다.

그런데 인출할 때 사용된 이더 송금을 위해 call()을 사용했습니다. call()은 transfer()나 send()와는 다르게 2300 가스 제한 없이 트랜잭션을 시작할 때의 가스가 그대로 전달되므로 남은 가스가 충분하다면 다른 코드를 실행할 수 있습니다.

***0.8.0 이후부터 오버/언더플로우가 발생하면 예외가 발생합니다. 이 예제에서는 재진입 문제를 명확히 보여주기 위해 편의상 unchecked 블록을 사용하여 언더플로우가 일어나는 것을 방지했습니다.**

공격자는 이러한 점을 악용하여 다음과 같은 컨트랙트를 작성할 수 있습니다.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./Donation.sol";

contract Attacker {

    address payable owner;
    uint256 public v = 0.5 ether;
    Donation public donation;

    constructor(address payable _addr) payable {
        owner = payable(msg.sender);
        donation = Donation(_addr);
    }

    function donate() external {
        donation.donate{value: v}(address(this));
    }

    receive() external payable {
        if (address(donation).balance >= v) {
            donation.withdraw(v);
        }
    }

    function kill() external {
        selfdestruct(owner);
    }
}

```

```
}

function getBalance() public view returns (uint) {
    return address(this).balance;
}

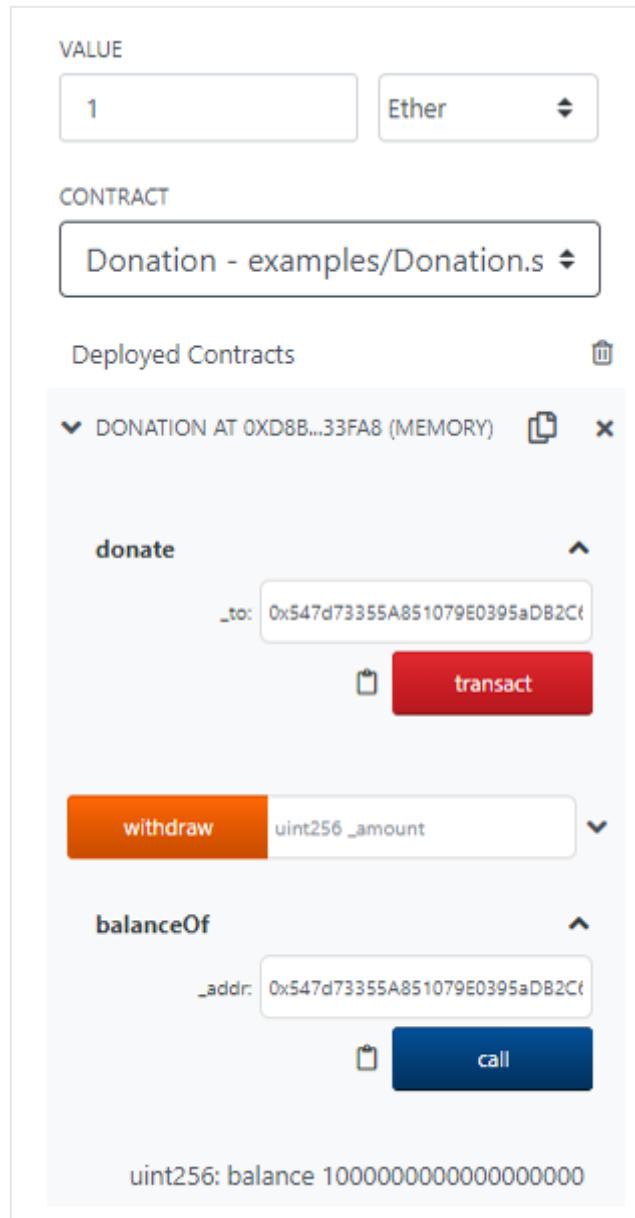
}
```

Attacker 컨트랙트에서 주의 깊게 봐야 할 부분은 바로 receive() 함수입니다. 이더를 받는 함수 안에서 다시 Donation 컨트랙트의 인출 함수인 withdraw() 함수를 호출하고 있습니다. 이 코드가 어떻게 Donation 컨트랙트에 기부된 이더를 무단으로 인출할 수 있는지 차례로 실행해보면서 그 결과를 분석해보겠습니다.

우선 리믹스에서 두 컨트랙트를 배포합니다. Donation 컨트랙트를 먼저 배포해야 합니다. 배포된 주소를 가지고 Attacker 컨트랙트를 배포합니다. Attacker 컨트랙트를 배포할 때는 나중에 사용할 1이더 정도를 함께 전송합니다.

정상적으로 Donation 컨트랙트에 기부를 합니다. 리믹스를 이용하여 donate() 함수를 호출합니다. 기부를 받게 되는 계정은 임의의 계정을 넣어도 상관없습니다. VALUE 항목에 기부할 수량을 넣고 기부 받을 계정을 입력하고 donate()를 클릭합니다.

예를 들어 계정 0x547d...7eAF에게 1이더를 기부한다면 위와 같이 입력하고 `donate()`를 호출하면 되겠습니다. 이런 식으로 여러 계정으로부터 기부를 받게 되면 Donation 컨트랙트에 이더가 예치되고 그 기록은 `balances` 장부에 기록됩니다. 각 계정의 인출 가능한 수량은 `balanceOf()`로 조회할 수 있습니다.



기부를 받은 계정들은 장부에 기록된 기부 수량 이상을 인출할 수 없는 조건이 적용되어 있기 때문에 부정 인출이 일어날 수 없는 것처럼 보입니다.

```
function withdraw(uint256 _amount) public {
    if (balances[msg.sender] >= _amount) {
```

```

        (bool bOk, ) = msg.sender.call{value: _amount}("");
        if (!bOk) {
            revert();
        }

        unchecked{ balances[msg.sender] -= _amount; }
    }
}

```

이제 Attacker 컨트랙트가 어떻게 한도를 넘어서 인출할 수 있는지 보겠습니다. 공격자는 Attacker 컨트랙트의 `donate()` 함수를 실행합니다. 이 함수는 Donation 컨트랙트의 `donate()` 함수를 호출합니다. 그런데 기부를 받는 계정은 Attacker 컨트랙트 자신으로 되어 있습니다. 여기서는 0.5이더를 자신에게 기부하도록 되어 있습니다(배포할 때 함께 전송된 1이더를 사용).

```

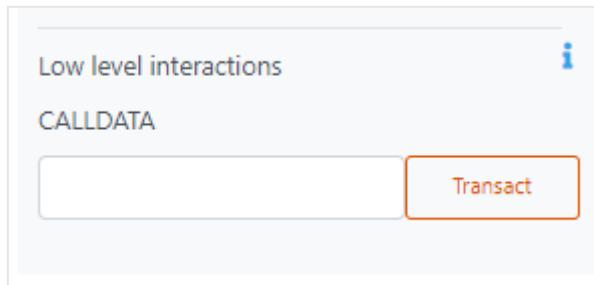
function donate() external {
    donation.donate{value: v}(address(this));
}

```

지금까지는 컨트랙트의 규칙을 따르는 정상적인 과정입니다. 자기 자신에게 기부를 하더라도 컨트랙트에서 그 이상을 인출할 수는 없을 것입니다.

그런데 Attacker 컨트랙트를 통해 이렇게 자신에게 기부된 이더를 인출해보겠습니다. Donation 컨트랙트의 인출 메소드인 `withdraw()` 함수를 호출할 때 `receive()` 함수 안에서 수행합니다.

리믹스에서 `receive()`를 실행하는 방법은 Attacker 컨트랙트의 아래 메뉴에 있는 "Low level interactions"을 이용하면 됩니다. 아무것도 입력하지 않고 Transact 버튼을 클릭하면 `receive()` 함수가 호출됩니다.



이렇게 이더 전송 함수를 호출하면 어떤 일이 벌어질까요? Donation 컨트랙트의 로직은 `withdraw()` 함수가 호출되면 이 함수에서는 다시 함수를 호출한 `msg.sender`에게 요청한 수량 `_amount`의 이더를 전송하도록 되어 있습니다.

```
(bool bOk, ) = msg.sender.call{value: _amount}("");
```

call()을 사용하여 receive()를 호출했기 때문에 앞서 설명한 것처럼 트랜잭션을 시작면서 실행되고 남은 가스가 계속 전달됩니다. 그런데 msg.sender는 Attacker 컨트랙트가 되므로 다시 receive() 함수가 호출되면서 계속 withdraw() 함수로 “재진입”합니다. 이것은 단일 트랜잭션으로 실행되는데 Donation 컨트랙트의 잔액이 Attacker가 인출해가는 이더 수량보다 작아질 때까지 계속됩니다.

예를 들어 Donation에 기부된 이더의 총수량이 2.5이더라고 하면 Attacker 컨트랙트가 0.5이더씩 인출을 시작한 후에는 0이 될 때까지 인출이 계속됩니다. 이더스캔에서 이 트랜잭션을 조회해보면 어떤 일이 벌어졌는지 알 수 있습니다.

Type Trace Address	From	To	Value
call_0_1	0x985f19869d03d69780...	0x3da54bc1e3235d2cb9...	0 Ether
call_0_1_1	0x3da54bc1e3235d2cb9...	0x985f19869d03d69780...	0.5 Ether
call_0_1_1_1	0x985f19869d03d69780...	0x3da54bc1e3235d2cb9...	0 Ether
call_0_1_1_1_1	0x3da54bc1e3235d2cb9...	0x985f19869d03d69780...	0.5 Ether
call_0_1_1_1_1_1	0x985f19869d03d69780...	0x3da54bc1e3235d2cb9...	0 Ether
call_0_1_1_1_1_1_1	0x3da54bc1e3235d2cb9...	0x985f19869d03d69780...	0.5 Ether
call_0_1_1_1_1_1_1_1	0x985f19869d03d69780...	0x3da54bc1e3235d2cb9...	0 Ether
call_0_1_1_1_1_1_1_1_1	0x3da54bc1e3235d2cb9...	0x985f19869d03d69780...	0.5 Ether
call_0_1_1_1_1_1_1_1_1_1	0x985f19869d03d69780...	0x3da54bc1e3235d2cb9...	0 Ether
call_0_1_1_1_1_1_1_1_1_1_1	0x3da54bc1e3235d2cb9...	0x985f19869d03d69780...	0.5 Ether

0x3Da5...가 Donation 컨트랙트이고 0x985F...가 Attacker입니다. 인출이 일어나고 있지만 재진입으로 인하여 Attacker 계정의 잔액을 여전히 0.5이더 상태로 판단하기 때문에 이렇게 연속적인 인출이 가능한 것입니다.

만약 call()을 사용하여 이더를 전송하지 않고 transfer() 또는 send() 함수를 사용했다라면 어떻게 되었을까요? 즉 withdraw() 함수를 다음과 같이 작성하는 것입니다.

```
function withdraw(uint256 _amount) public {
    if (balances[msg.sender] >= _amount) {
        payable(msg.sender).transfer(_amount);
        unchecked{ balances[msg.sender] -= _amount; }
    }
}
```

2300 가스 제한이 있기 때문에 receive() 함수 안에서는 다른 컨트랙트의 함수를 호출할 수 없습니다. 따라서 Attacker 컨트랙트에서 withdraw() 함수를 다시 호출하는 것은 실패할 것입니다.

이더를 보낼 때 call() 대신에 transfer()를 사용하는 것이 일반적이었지만 이스탄불 하드포크 이후에는 이러한 기조가 다소 변경되었습니다. 2019년 이스탄불 하드포크에서 EVM의 opcode에 책정된 가스를 변경하면서 하드포크 전에는 2300 이내였던 코드가 2300을 넘어버리는 결과를 초래했기 때문입니다.

그래서 EVM opcode의 가스는 이더리움 클라이언트가 업데이트되면서 여러 가지 이유에 의해 조정될 수 있기 때문에 2300 가스 제한이 있는 transfer()나 send()를 사용하는 것이 바람직하지 않다는 주장이 제기되었습니다. 스마트 컨트랙트를 가스에 의존적으로 작성되면 오히려 향후 동작에 문제가 생길 수 있다는 것입니다.

그렇다면 해결책은 무엇일까요? 다시 call()을 사용하는 것입니다. 다만 이 경우에 재진입 문제는 소위 말하는 “Checks-Effects Interactions” 패턴으로 해결해야 합니다. 앞서 예제에서 call()을 그대로 사용하면서 다음과 같이 변경할 수 있습니다.

```
function withdraw(uint256 _amount) public {
    if (balances[msg.sender] >= _amount) {
        unchecked{ balances[msg.sender] -= _amount; }

        (bool bOk, ) = msg.sender.call{value: _amount}("");
        if (!bOk) {
            revert();
        }
    }
}
```

수정된 코드를 보면 이더를 송금하기 전에 장부 balances의 잔액을 먼저 차감했습니다. 이렇게 되면 재진입이 발생하더라도 장부 상의 데이터는 변경된 상태이기 때문에 인출 한도 이내에서만 전송이 이루어지게 됩니다. 다시 말해서 상태 변수를 먼저 변경하고 그 다음에 외부 전송을 하는 순으로 코드를 작성해야 합니다.

2.15.5. 대리자(proxy) 컨트랙트

“대리자” 컨트랙트는 호출 데이터를 받아서 다른 컨트랙트 함수에 전달하는 중개 컨트랙트인데 저수준 호출 call()을 통해서 대리자 컨트랙트를 거쳐 다른 컨트랙트의 함수를 실행하는 것을 말합니다. 다음과 같은 코드를 보겠습니다.

MyProxy 컨트랙트의 `callByProxy()` 함수는 `_addr` 컨트랙트에 `_payload`를 전달합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyProxy {

    function callByProxy(address _addr, bytes memory _payload) public returns (bytes memory) {
        (bool bOk, bytes memory result) = _addr.call(_payload);
        if (!bOk) {
            revert();
        }
        return result;
    }
}
```

예를 들어 아래와 같은 MyContract의 sum() 함수를 호출하려면 호출 데이터(payload)를 만들어야 합니다. 이것은 abi.encodeWithSignature("sum(uint256,uint256)", _a, _b)를 사용하면 됩니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

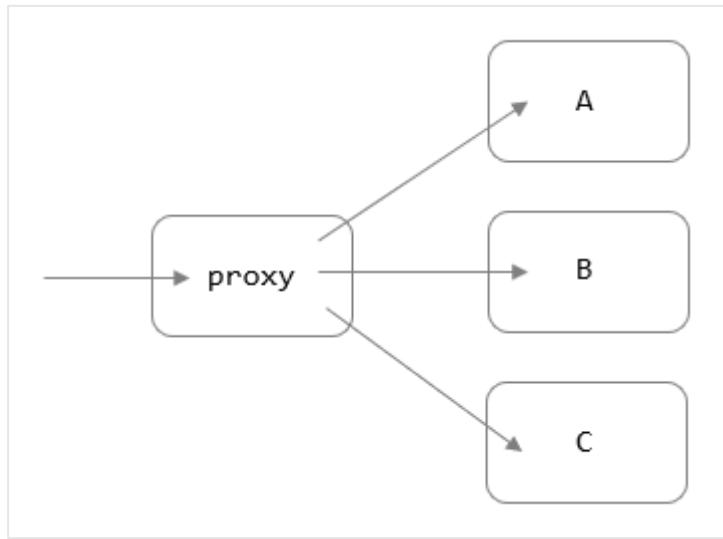
contract MyContract {

    function sum(uint256 _a, uint256 _b) external pure returns (uint256) {
        return _a + _b;
    }
}
```

a = 100, b = 500의 경우 훈련 데이터는 아래와 같습니다(너무 길어서 개행되어 있습니다).

`_addr`에 MyContract의 계정 주소를, `_payload`에 위에서 만든 호출 데이터를 `callByProxy()`에 전달하면 MyContract의 `sum()` 함수가 다시 호출되고 bytes 타입의 리턴 값을 디코딩하면 600이라는 값을 얻을 수 있습니다.

그런데 이렇게 대리자 컨트랙트를 통해 다른 컨트랙트의 함수를 실행할 수 있는 것이 언뜻 효율적으로 보일 수도 있습니다. 그래서 어떤 목적에 의해 대리자 컨트랙트를 앞에 두고 이 컨트랙트에서 여러 컨트랙트의 함수를 실행할 수 있도록 권한을 주는 경우가 있습니다.



그러나 이것은 보안적인 측면에서 바람직하지 않습니다. 사실 컨트랙트의 외부 함수는 누구든지 실행할 수 있고 더구나 대리자 컨트랙트는 임의의 함수 호출 데이터를 전달할 수 있으므로 대리자 컨트랙트에게 권한을 주는 것은 위험할 수 있습니다.

2.15.6. tx.origin과 msg.sender

`tx.origin`은 트랜잭션을 처음 시작한 계정으로 항상 EOA가 됩니다. `msg.sender`는 처음 트랜잭션을 시작한 계정일 수도 있지만 중간에 컨트랙트를 경유할 경우 컨트랙트 계정으로 바뀔 수 있습니다. 그런데 `tx.origin`과 `msg.sender`가 동일하다고 생각하고 `tx.origin`을 권한을 검사할 때 사용하는 것은 위험합니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract MyContract {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(owner == tx.origin, "Caller is not the owner");
       _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}

```

transferOwnership() 함수를 실행할 수 있는 것은 onlyOwner를 통과한 계정으로 제한했습니다. 그런데 owner를 tx.origin과 비교하고 있습니다. 공격자가 다음과 같은 컨트랙트를 작성한다고 가정해보겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract Attacker {

    address badUser;
    MyContract target;

    constructor(address _addr) {
        badUser = msg.sender;
        target = MyContract(_addr);
    }

    receive() external payable {
        target.transferOwnership(badUser);
    }
}
```

만약 공격자가 교묘한 방법을 동원하여 MyContract의 배포 계정이 Attacker 컨트랙트에게 이더를 전송하도록 유도하여 성공한다면 receive() 함수가 받으면서 transferOwnership()이 실행될 것이고 tx.origin은 배포 계정이 됩니다. 이 경우 modifier onlyOwner를 무사히 통과하게 될 것이고 파라미터로 전달된 badUser 계정으로 owner가 변경될 것입니다.

2.16. 정리

2장에서는 이더리움 스마트 컨트랙트 프로그래밍 언어 솔리디티의 기본 문법과 컨트랙트 작성 시 유의할 점을 살펴보았습니다. 처음에 언급한 것처럼 솔리디티는 아직 개발 중인 언어이기 때문에 변경되거나(없어지거나) 추가되는 부분들이 자주 발생합니다. 따라서 공식 홈페이지를 자주 방문하여 어떤 것들이 바뀌었는지 따라가는 노력이 필요합니다.

다행스럽게도 리믹스와 같은 편리하고 훌륭한 도구가 있기 때문에 변화가 많은 솔리디티를 배우는데 큰 도움이 될 것입니다.

NFT(Non-Fungible Token) 표준



3장에서는 NFT 표준 ERC-721과 ERC-1155에 대한 내용을 다루겠습니다. NFT 컨트랙트는 이를 표준에 따라 작성하게 됩니다.

3.1. 이더리움의 표준

이더리움은 스마트 컨트랙트를 실행할 수 있는 애플리케이션 플랫폼으로서, 다양한 탈중앙화 애플리케이션(Decentralized Application, Dapp)을 구동하기 위해 여러 가지 표준을 정하고 있습니다. 표준을 정한다는 것은 컨트랙트의 세부 구현이 서로 다를 수는 있어도 그 인터페이스는 공통적으로 약속되어 있다는 것을 의미합니다.

예를 들어 메타마스크와 같은 범용 지갑 소프트웨어를 통해 토큰을 전송할 때 호출해야 하는 컨트랙트의 함수가 어떤 개발팀은 “sendToken”으로, 어떤 개발팀은 “transfer”로 이름을 붙인다면 컨트랙트마다 호출 함수를 관리해야 하므로 매우 비효율적일 것입니다.

이더리움에서는 이렇게 애플리케이션 레벨의 표준들을 정할 때 이더리움 개선 제안(EIP)이라는 공개 토론을 통해 결정합니다. EIP 포럼에서는 이더리움의 생태계 전분야에 걸쳐 개발자와 연구자들, 기술 전문가들이 모여 다양한 제안과 의견을 나누게 됩니다. 관심있는 독자들은 아래 링크를 방문해 보기 바랍니다.

<https://eips.ethereum.org/>

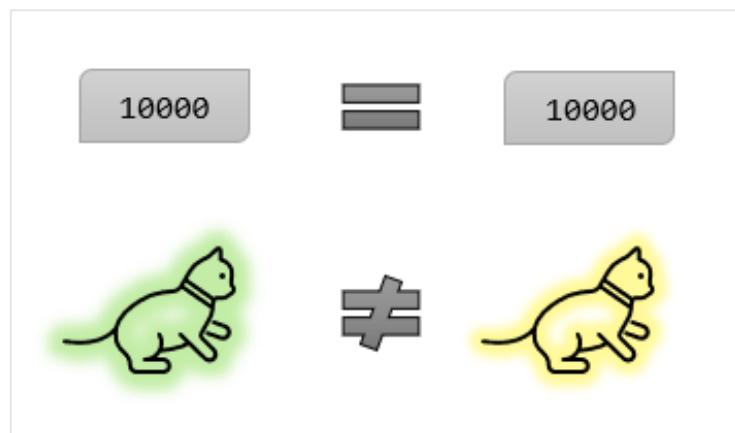
3.2. 토큰 표준

토큰이라고 하는 것은 “Internet of Value”라는 블록체인의 역할에서 핵심적인 요소라고 할 수 있습니다. 왜냐하면 유무형의 가치를 자유롭게 거래하기 위해서는 반드시 토큰화라는 과정을 거쳐 블록체인 상에서 유통할 수 있는 가상 자산으로 만들어야 하기 때문입니다.

이더리움의 토큰 표준은 EIP의 ERC(Ethereum Request for Comments)라는 카테고리에 포함되어 있습니다. 가장 많이 알려진 것은 화폐처럼 사용할 수 있도록 설계된 ERC-20이라는 토큰 표준입니다. 이더리움에서 먼저 이러한 표준을 정하고 많은 Dapp들이 이 표준에 맞추어 개발되고 사용되었기 때문에 이더리움 외의 다른 블록체인에서도 ERC-20이라는 이름과 유사한 이름(예를 들어 바이낸스 스마트 체인에서는 BEP-20)으로 동일한 표준들이 존재합니다.

ERC-20은 화폐처럼 서로 교환 가능한(fungible) 토큰입니다. 내가 가진 만원권 지폐와 여러분들이 가진 만원권 지폐는 가치가 서로 동일하기 때문에 맞바꾸어도 아무 문제가 없습니다. 그런데 내가 키우고 있는 고양이와 옆집에 사는 앤리스가 키우는 고양이는 어떨까요? 같은 고양이라서 교환 가능하다고 말할 수 있을까요?

내가 키우는 고양이는 단순한 고양이 그 이상의 가치가 있을 것입니다. 그 가치가 나에게만 의미가 있는 것일 수도 있지만 만약 희귀 품종이라면 시장에서의 가치 또한 높을지도 모릅니다. 앤리스의 고양이도 나름의 가치를 가지고 있을 것입니다. 분명한 것은 둘 다 고양이지만 동일한 가치는 아니라는 사실입니다.



고유한 가치를 지니고 있어서 서로 교환할 수 없는 토큰을 교환할 수 없는(Non-fungible) 토큰, NFT라고 합니다. 세상에 단 하나밖에 없는 유일무이한 가치를 지닌 예술품(예를 들어 다빈치의 모나리자 같은)을 토큰화하는 경우에는 대체 불가능한 토큰이라고 말할 수도 있을 것입니다.

ERC-20 토큰에서는 수량이 많을수록 더 많은 가치를 소유한 것이 되지만 NFT에서는 상대적으로 수

량은 그렇게 큰 의미가 없습니다. 수량보다는 개별 토큰의 고유한 가치가 더 중요합니다.

그런데 여기서 유념해야 할 것은 NFT가 가치 그 자체를 저장하고 있다는 것은 아니라는 점입니다. 레오나르도 다빈치의 모나리자를 NFT로 만든다고 했을 때 NFT가 저장하고 있는 것은 모나리자에 대한 소유권 정보(데이터)이지, 모나리자 그 자체는 아니라는 말입니다. 물론 NFT 내에 NFT가 표시하고 있는 디지털 이미지나 문자를 저장할 수는 있겠지만 일반적으로 NFT는 부동산 등기 문서처럼 소유권 정보만을 가지고 있습니다.

이더리움의 NFT 표준은 이러한 소유권 정보를 어떻게 블록체인에 기록할 것인지 규정하고 있는 스펙입니다. 이제부터 NFT 토큰 표준인 ERC-721과 ERC-1155에 대해 알아보겠습니다.

3.3. ERC-721

이더리움의 NFT 표준은 ERC-721에 정해져 있습니다. ERC-721은 고유한 번호가 부여된 토큰에 대한 소유권 정보를 저장합니다(고유한 번호란 이더리움 전체가 아니라 해당 컨트랙트 내에서 고유하다는 의미). 또 소유권 이전과 위임 등의 기능들도 정의하고 있습니다.

이제부터 ERC-721 표준에서 정의하고 있는 인터페이스를 하나씩 살펴보겠습니다. 일반적으로 인터페이스는 구현부가 없는 함수들을 정의한 것을 말합니다. 즉 다양하게 구현할 수 있지만 서로 약속된 함수명과 전달인자(함수 시그너처)를 통해 데이터를 주고받을 수 있도록 하는 것입니다.

ERC-721 스펙은 다음 링크에서 찾을 수 있습니다.

<https://eips.ethereum.org/EIPS/eip-721>

3.3.1. 기본 인터페이스

앞서 NFT에 대한 특징을 설명한 것처럼 ERC-721이 저장할 수 있는 데이터는 다음과 같이 세상의 거의 모든 유무형의 가치에 대한 소유권 정보입니다.

- 실물 자산 – 부동산, 미술품 등
- 가상 수집품 – 디지털 이미지, 동영상 등
- “Negative” 자산 – 채무, 채권 등

ERC-721 표준은 솔리디티로 작성된 인터페이스로 정의되어 있습니다. 우선 필수적으로 구현해야 하는 인터페이스는 다음과 같습니다.

```
pragma solidity ^0.4.20;

interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);

    event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);

    event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);

    function balanceOf(address _owner) external view returns (uint256);

    function ownerOf(uint256 _tokenId) external view returns (address);

    function safeTransferFrom(address _from,
                            address _to,
                            uint256 _tokenId,
                            bytes data) external payable;

    function safeTransferFrom(address _from, address _to, uint256 _tokenId) external payable;

    function transferFrom(address _from, address _to, uint256 _tokenId) external payable;

    function approve(address _approved, uint256 _tokenId) external payable;

    function setApprovalForAll(address _operator, bool _approved) external;

    function getApproved(uint256 _tokenId) external view returns (address);

    function isApprovedForAll(address _owner, address _operator) external view returns (bool);
}
```

표준이 정해진 시점의 솔리디티 컴파일러 버전 ^0.4.20과 현재 이 글이 작성되는 시점의 컴파일러 버전과 차이가 날 수 있기 때문에 사용하는 버전에 맞추어 변경하는 것은 허용됩니다. 예를 들어 safeTransferFrom 함수는 ^0.8.0 기준으로 다음과 같이 변경해야 합니다.

```
function safeTransferFrom(address _from,
                        address _to,
                        uint256 _tokenId,
                        bytes calldata data) external payable;
```

각 함수의 상태 변경 여부(State mutability)는 준수해야 하지만 상황에 따라 payable, non-payable, view, pure 순으로 변경이 허용됩니다. 인터페이스의 특성상 함수들이 모두 external로 선언되어 있는

데 이것을 public으로 변경해도 무방합니다.

처음보면 다소 복잡해 보이지만 차근차근 보면 크게 어려운 내용은 없습니다. 3개의 이벤트와 9개의 함수가 정의되어 있고 부가적으로 ERC-165 인터페이스도 함께 구현해야 합니다. ERC-165라는 인터페이스는 컨트랙트가 어떤 표준 인터페이스를 구현한 것인지를 확인하기 위한 "Standard Interface Detection" 규약입니다. 이것을 통해 해당 컨트랙트가 ERC-721 컨트랙트인지를 판단할 수 있습니다.

```
pragma solidity ^0.4.20;

interface ERC165 {
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

예를 들어 다음과 같이 구현할 수 있겠습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "./ERC721.sol";
import "./ERC165.sol";

contract MyNFT is ERC721, ERC165 {

    function supportsInterface(bytes4 interfaceID) external view override returns (bool) {
        return interfaceID == type(ERC721).interfaceId; // 0x80ac58cd
    }

    //TODO ERC-721 implement
    //
}
```

여기서 `type(ERC721).interfaceId` 은 ERC-721에 정의된 9개의 함수들의 셀렉터를 XOR를 한 값으로 4 바이트의 `0x80ac58cd`라는 값이 나옵니다(16진수 2자리가 1 바이트). 이것이 ERC-721의 인터페이스 식별자가 되는 것입니다. 외부에서 `supportsInterface()`를 호출하여 `true`가 리턴되면 해당 컨트랙트는 ERC-721을 구현한 컨트랙트로 판단할 수 있겠습니다.

다수의 인터페이스를 구현한 경우에는 다음과 같이 OR 연산자를 사용합니다.

```
function supportsInterface(bytes4 interfaceId) external view override(ERC165, IERC165) returns (bool) {
    return
        interfaceId == type(ERC721).interfaceId ||
        interfaceId == type(ERC721Metadata).interfaceId ||
        super.supportsInterface(interfaceId);
}
```

다음에는 표준에 정의된 함수와 이벤트에 대해 알아보겠습니다.

- `balanceOf(address _owner) returns (uint256)`

`_owner`로 지정된 계정이 소유한 NFT의 수량을 리턴합니다. `address(0)`에 대해서는 유효하지 않은 것으로 처리해야 합니다.

- `ownerOf(uint256 _tokenId) returns (address)`

토큰 번호 `_tokenId`를 가진 계정 주소를 리턴합니다. 만약 소유자 계정이 `address(0)`라면 유효하지 않은 것으로 처리해야 합니다.

- `safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data)`

`_from` 계정이 소유한 토큰 번호 `_tokenId` 토큰을 `_to` 계정으로 전송합니다. 즉 소유권을 이전하는 함수입니다. 임의의 계정이 이 함수를 실행할 수도 있으므로 제약조건을 반드시 넣어야 합니다.

`_from` 계정은 토큰의 소유자이거나 소유자로부터 위임받은 계정이 되어야 하고, `_to` 계정이 `address(0)`이거나 존재하지 않는 `_tokenId` 경우에는 예외를 발생시켜야 합니다. 함수명에 “`safe`”를 붙인 것은 잘못된 전송을 되돌릴 수 있도록 구현해야 한다는 의미도 있습니다.

특히 `_to` 계정이 컨트랙트인 경우 `onERC721Received`라는 함수를 호출하도록 해야 합니다. 일반적으로 토큰이 컨트랙트로 전송되는 경우에는 컨트랙트에 영원히 귀속되지 않도록 조심해야 합니다. ERC-721 표준에서는 상대 컨트랙트에 대해서도 `ERC721TokenReceiver`라는 인터페이스를 구현하도록 규정하고 있습니다. 마지막 인자인 `data`는 어떤 데이터를 함께 전송할 필요가 있을 때 사용합니다.

- safeTransferFrom(address _from, address _to, uint256 _tokenId)

safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data)과 동일한 기능의 함수로 다만 마지막 data 인자가 없습니다.

- transferFrom(address _from, address _to, uint256 _tokenId)

이 함수 역시 safeTransferFrom과 기능적으로 동일합니다. 토큰을 소유한 계정이 직접 _to 계정에게 소유권을 이전하는 함수입니다. safeTransferFrom과 transferFrom은 모두 기능적으로 동일하므로 하나를 작성하여 호출하는 형태가 됩니다. 예를 들어서 다음과 같은 형태가 될 수 있겠습니다.

```
function transferFrom(address _from, address _to, uint256 _tokenId) public payable {
    ...
}

function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes memory data) public payable {

    transferFrom(_from, _to, _tokenId);

    if (_to.isContract()) {
        bytes4 result = ERC721TokenReceiver(_to).onERC721Received(msg.sender,
            _from,
            _tokenId,
            data);
        require(result == bytes4(keccak256("onERC721Received(address,address,uint256,bytes)")));
    }
}

function safeTransferFrom(address _from, address _to, uint256 _tokenId) public payable {
    safeTransferFrom(_from, _to, _tokenId, "");
}
```

그 다음은 위임(증개)과 관련된 함수들입니다. 위임은 내가 가진 토큰의 처분을 다른 계정에게 맡긴다는 것을 의미합니다. 다른 계정이란 보통 컨트랙트 계정을 말합니다.

- approve(address _approved, uint256 _tokenId)

_approved 계정에게 토큰 번호 _tokenId 토큰의 소유권을 처분할 수 있는 권한을 위임합니다. 이 함

수를 호출하는 계정은 당연히 토큰 소유자이거나 위임을 받은 계정이 되어야 합니다. `_approved` 계정이 `address(0)`가 되면 유효하지 않은 호출로 처리되어야 합니다.

- `setApprovalForAll(address _operator, bool _approved)`

소유자가 가진 모든 토큰을 증개할 수 있는 계정 `_operator`를 지정합니다. 위임 여부는 `bool` 타입의 `_approved`로 전달받으므로 `false`로 하면 언제든지 위임을 철회할 수 있습니다. 증개 계정은 하나 이상 지정하는 것이 가능해야 합니다. 예를 들어 다음과 같이 구현할 수 있습니다.

```
mapping(address => mapping(address => bool)) operators;

function setApprovalForAll(address _operator, bool _approved) external {
    operators[msg.sender][_operator] = _approved;
    emit ApprovalForAll(msg.sender, _operator, _approved);
}
```

- `getApproved(uint256 _tokenId) returns (address)`

`approve` 함수에 의해 위임된 계정(증개 계정)을 조회하는 함수입니다. 위임되지 않은 토큰의 경우는 `address(0)`를 리턴합니다.

- `isApprovedForAll(address _owner, address _operator) returns (bool)`

`setApprovalForAll` 함수를 통해 지정된 증개 계정의 위임 상태 여부를 `true/false`로 리턴합니다.

그 다음에는 3개의 이벤트입니다.

- `Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId)`

토큰이 전송될 때 발생합니다. 토큰 전송이라는 것은 결국 소유권의 변경을 의미하는데 새로 생성되거나 없어질 때도 이벤트가 발생하도록 정의합니다.

- Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId)

토큰에 대한 위임 계정(approved)이 변경되거나 새로 지정될 때마다 발생하는 이벤트입니다. approve 함수를 실행할 때 발생합니다.

- ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved)

토큰을 위임하여 중개자(operator)가 지정될 때 발생하는 이벤트입니다. setApproveForAll 함수가 호출될 때 발생합니다.

3.3.2. 확장 인터페이스

ERC-721 표준은 기본 인터페이스 외에도 몇 가지 확장 인터페이스를 함께 정의하고 있습니다.

대표적인 것은 ERC-721 컨트랙트가 다른 컨트랙트에 토큰을 보낼 때 상대 컨트랙트가 구현해야하는 ERC721TokenReceiver 인터페이스입니다.

```
interface ERC721TokenReceiver {
    function onERC721Received(
        address _operator,
        address _from,
        uint256 _tokenId,
        bytes _data) external returns(bytes4);
}
```

onERC721Received 함수의 리턴 값은 자신의 함수 셀렉터인데 결국 상수 값(0x150b7a02)이라고 할 수 있습니다. 예를 들면 다음과 같이 구현할 수 있겠습니다.

```
function onERC721Received(
    address _operator,
    address _from,
    uint256 _tokenId,
    bytes calldata _data) override external returns(bytes4) {
    return this.onERC721Received.selector; // 0x150b7a02;
}
```

토큰을 전송한 후 상대 컨트랙트의 onERC721Received를 호출합니다. 리턴 값이 일치하지 않는 경우에는 예외를 발생시켜서 전송을 취소해야 합니다.

NFT는 일반적으로 소유권 정보만을 저장하는 것이므로 이 NFT가 어떤 자산에 대한 소유권 정보인지 알 수 있습니다. 이 때 사용하는 것이 NFT의 메타정보입니다. ERC-721 표준은 메타정보에 대한 표준을 ERC721Metadata로 정의하고 있습니다.

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external view returns (string _name);
    function symbol() external view returns (string _symbol);
    function tokenURI(uint256 _tokenId) external view returns (string);
}
```

NFT는 그 자체의 고유한 가치가 의미가 있기 때문에 이름(name)이나 심볼(symbol)은 그렇게 중요하지 않습니다. 그래서 ERC721Metadata 인터페이스는 필수 구현 인터페이스는 아닙니다.

메타정보 인터페이스에서 가장 중요한 것은 tokenURI입니다. 토큰의 여러 속성들, 그리고 이미지 정보 등을 JSON 형태로 외부에 저장할 수 있는데 이것을 가리키는 URI(Uniform Resource Identifier) 정보만을 컨트랙트에 저장하는 것입니다. 왜냐하면 메타정보 자체를 이더리움 내에(온체인) 저장하는 것은 비효율적이고 비용도 많이 발생하기 때문입니다. JSON의 형태는 다음과 같습니다.

```
{
    "title": "She is Art",
    "name": "She is Art",
    "type": "object",
    "imageUrl": "https://ipfsgateway.makersplace.com/ipfs/QmXK...LMfr",
    "description": "Made with love and digital tools",
    "attributes": [{"trait_type": "Creator", "value": "Bit Error"}],
    "properties": {"name": {"type": "string", "description": "She is Art"}, "description": {"type": "string", ...}}
}
```

메타정보를 저장하고 있는 JSON 파일은 중앙서버(클라우드)에 저장될 수 있고 아니면 분산 파일 시스템의 일종인 IPFS(InterPlanetary File System)에 저장될 수도 있습니다. 또 메타정보 내에 있는 imageUrl 역시 중앙서버나 IPFS를 가리킬 수 있습니다. IPFS는 다수의 파일 서버들이 모여 저장 공간을 제공하는 분산 네트워크이기 때문에 여기에 메타정보를 저장하는 것을 권장합니다.

개인이 IPFS를 쉽게 활용할 수 있는 서비스가 있습니다. 피나타(Pinata)와 NFT 스토리지 등이 대표적인 서비스입니다.

NFT Storage	https://nft.storage/
Pinata	https://www.pinata.cloud/

NFT를 조회할 때 필요한 확장 인터페이스는 ERC721Enumerable에서 정의합니다.

```
interface ERC721Enumerable /* is ERC721 */ {  
    function totalSupply() external view returns (uint256);  
    function tokenByIndex(uint256 _index) external view returns (uint256);  
    function tokenOfOwnerByIndex(address _owner, uint256 _index)  
        external view returns (uint256);  
}
```

totalSupply()는 발행된 토큰의 총수량을 나타내고 tokenByIndex()는 인덱스(보통 일련번호)로 토큰 번호를 리턴하는 함수입니다. tokenOfOwnerByIndex()는 소유자 계정과 인덱스를 전달하면 토큰 번호를 리턴합니다.

ERC-721에서는 NFT를 어떻게 발행하는지에 대한 표준은 정하지 않습니다. 토큰을 발행하는 것을 흔히 민트 “mint”라고 말합니다(반대로 없어지는 것은 소각 “burn”). NFT 발행 로직은 구현하기 나름이며, 토큰 식별자에 해당하는 uint256 타입의 토큰 번호를 어떻게 만들어야 하는지도 정해져 있지 않습니다. 다만 토큰 번호는 동일한 컨트랙트 내에서 중복되지 않게 생성해야 합니다. 토큰 번호는 이더리움 내에서 컨트랙트 주소+토큰 번호 쌍으로 유일하게 식별될 수 있습니다.

3.4. ERC-1155

앞서 살펴본 ERC-721 표준은 보통 단일한 종류의 NFT를 구현합니다. 이를테면 어떤 디지털 아트 작가의 작품들을 NFT로 발행한다면 ERC-721 컨트랙트 하나면 충분할 것입니다.

그런데 게임 아이템을 NFT로 발행하려고 한다면 어떻게 될까요? 게임 내에서 플레이 되는 아이템들이 각각 고유한 디지털 카드로 발행되고 카드를 일정수량 모으면 해당 아이템의 레벨이 올라가고 그에 따라 마법이 강해지도록 하고 싶다면 NFT를 어떻게 설계하는 것이 바람직할까요? 아이템별로 ERC-721 표준의 컨트랙트를 만들어야 할까요?

이러한 문제를 해결하기 위해서 수량의 개념이 포함된, 그러니까 ERC-20 토큰과 ERC-721 토큰을 결합한 형태의 ERC-1155라는 표준이 정해지는 계기가 되었습니다. ERC-1155는 “semi-fungible” 토큰이라고 말합니다.

ERC-1155는 다중 토큰(Multi Token) 표준이라는 이름으로 기술되어 있습니다.

<https://eips.ethereum.org/EIPS/eip-1155>

3.4.1. 기본 인터페이스

ERC-721 표준과 마찬가지로 ERC-165 인터페이스를 구현해야 합니다. ERC-1155의 인터페이스 식별자는 0xd9b67a26입니다.

기본 인터페이스는 이벤트 4개와 6개의 함수로 정의되어 있습니다.

```
pragma solidity ^0.5.9;

interface ERC1155 /* is ERC165 */ {

    event TransferSingle(address indexed _operator,
                        address indexed _from,
                        address indexed _to,
                        uint256 _id,
                        uint256 _value);

    event TransferBatch(address indexed _operator,
                        address indexed _from,
                        address indexed _to,
                        uint256[] _ids,
                        uint256[] _values);

    event ApprovalForAll(address indexed _owner,
                        address indexed _operator,
                        bool _approved);

    event URI(string _value, uint256 indexed _id);

    function safeTransferFrom(address _from,
                            address _to,
                            uint256 _id,
                            uint256 _value,
                            bytes calldata _data) external;

    function safeBatchTransferFrom(address _from,
                                address _to,
                                uint256[] calldata _ids,
                                uint256[] calldata _values,
                                bytes calldata _data) external;

    function balanceOf(address _owner, uint256 _id) external view returns (uint256);

    function balanceOfBatch(address[] calldata _owners,
                            uint256[] calldata _ids) external view returns (uint256[] memory);

    function setApprovalForAll(address _operator, bool _approved) external;

    function isApprovedForAll(address _owner, address _operator) external view returns (bool);
}
```

이제 각 함수와 이벤트에 대해 알아보겠습니다.

- `safeTransferFrom(address _from, address _to, uint256 _id, uint256 _value, bytes calldata _data)`

`_from` 계정이 소유한 토큰 번호 `_id` 토큰을 수량 `_value` 만큼 `_to` 계정으로 전송합니다. 만약 `_to` 계정이 `address(0)`이라면 revert 해야 합니다. 당연히 `_from` 계정은 토큰의 소유자 또는 위임 계정이어야 하고 소유 수량은 전송하려는 `_value`보다 커야 합니다.

마지막 인자인 `data`는 부가적인 정보를 넘겨줄 필요가 있을 때 사용합니다. 함수 내에서 변경하지 못하도록 `calldata`로 지정되어 있습니다.

- `safeBatchTransferFrom(address _from, address _to, uint256[] calldata _ids, uint256[] calldata _values, bytes calldata _data)`

`safeTransferFrom`과 마찬가지로 토큰 소유권을 이전하는 함수입니다. 함수 이름 "Batch"가 들어간 것으로부터 알 수 있듯이 여러 유형의 토큰을 한 번에 전송하는 함수입니다(수수료를 절약할 수 있습니다). 당연한 이야기지만 보내려는 토큰 유형의 개수, 즉 `_ids` 배열의 길이와 수량 `_values` 배열의 길이는 같아야 합니다.

- `balanceOf(address _owner, uint256 _id) returns (uint256)`

`_owner` 계정이 소유한 토큰 번호 `_id`의 수량을 리턴합니다.

- `balanceOfBatch(address[] calldata _owners, uint256[] calldata _ids) returns (uint256[] memory)`

다수의 `_owners` 계정 배열이 소유한 토큰 번호 `_ids` 배열인 토큰 수량을 조회합니다. 리턴되는 배열은 순서대로 (`_owner, _id`) 쌍이 가진 수량이 됩니다.

- `setApprovalForAll(address _operator, bool _approved)`

토큰 소유자가 이 함수를 호출하면 소유한 모든 토큰을 지정된 계정 `_operator`에게 위임합니다. `_approved`가 `true`이면 위임, `false`로 하면 위임을 철회합니다.

- `isApprovedForAll(address _owner, address _operator) returns (bool)`

특정 소유자 `_owner`가 위임한 `_operator`의 현재 위임상태를 `true` 또는 `false`로 리턴합니다.

다음은 4개의 이벤트입니다.

- `TransferSingle(address indexed _operator, address indexed _from, address indexed _to, uint256 _id, uint256 _value)`

`safeTransferFrom`을 통해 토큰이 전송되었을 때 발생하는 이벤트입니다. 신규 토큰이 발행되거나 없어질 때도 이벤트가 발생합니다. 새로 발행되는 경우 `_from` 계정은 `address(0)`가 되고 소각하는 경우에는 `_to` 계정이 `address(0)`가 되어야 합니다.

- `TransferBatch(address indexed _operator, address indexed _from, address indexed _to, uint256[] _ids, uint256[] _values)`

`safeBatchTransferFrom`을 통해 토큰이 전송할 때 발생시키는 이벤트입니다. 토큰의 발행과 소각은 `safeTransferFrom`과 동일한 규칙이 적용됩니다.

- `ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved)`

`setApprovalForAll` 함수가 실행될 때 발생시키는 이벤트입니다.

- `URI(string _value, uint256 indexed _id)`

토큰 번호 `_id` 토큰의 메타정보가 저장된 곳을 가리키는 `URI` `_value`가 변경되었을 때 발생하는 이벤트입니다. 메타정보 인터페이스는 뒤에 나오는 `ERC1155Metadata_URI`에 정의되어 있습니다.

3.4.2. ERC1155TokenReceiver

ERC-1155에서도 컨트랙트로 토큰이 전송되는 경우 그 컨트랙트가 구현해야 할 함수가 있습니다.

```
pragma solidity ^0.5.9;
```

```

interface ERC1155TokenReceiver {

    function onERC1155Received(address _operator,
                                address _from,
                                uint256 _id,
                                uint256 _value,
                                bytes calldata _data) external returns(bytes4);

    function onERC1155BatchReceived(address _operator,
                                    address _from,
                                    uint256[] calldata _ids,
                                    uint256[] calldata _values,
                                    bytes calldata _data) external returns(bytes4);
}

```

_to 계정이 컨트랙트라면 safeTransferFrom의 경우 onERC1155Received를, safeBatchTransferFrom에서는 onERC1155BatchReceived를 구현해야 합니다. 리턴해야 하는 값은 자신들의 함수 셀렉터로, 각각 0xf23a6e61, 0xbc197c81입니다.

예를 들어 safeBatchTransferFrom은 다음과 같이 구현될 수 있습니다.

```

function safeBatchTransferFrom(address _from,
                               address _to,
                               uint256[] calldata _ids,
                               uint256[] calldata _values,
                               bytes calldata _data) external {

    require(_to != address(0), "Should be non-zero.");
    require(_ids.length == _values.length, "Both array length should be same");
    require(_from == msg.sender || operators[_from][msg.sender] == true, "Should be owner or operator");

    for (uint256 i=0; i<_ids.length; ++i) {
        uint256 id = _ids[i];
        uint256 value = _values[i];
        balances[id][_from] = balances[id][_from] - value;
        balances[id][_to] = balances[id][_to] + value;
    }

    emit TransferBatch(msg.sender, _from, _to, _ids, _values);

    if (_to.isContract()) {
        bytes4 result = ERC1155TokenReceiver(_to).onERC1155BatchReceived(msg.sender,
                                                                           _from, _ids, _values, _data);
        require(result == 0xbc197c81);
    }
}

```

3.4.3. ERC1155Metadata_URI

ERC-721과 마찬가지로 토큰 번호 _id의 메타정보(JSON 파일)는 외부에 저장되고 컨트랙트에는 그것

을 가리키는 URI 정보만을 저장합니다. 이 정보를 조회하는 함수를 구현하도록 합니다.

```
interface ERC1155Metadata_URI {
    function uri(uint256 _id) external view returns (string memory);
}
```

ERC-1155 표준에서는 토큰 번호에 따라 URI 정보를 가질 수 있습니다. 그런데 컨트랙트 내에 모든 토큰의 URI의 정보를 각각 저장하지 않고 기본 URI와 토큰번호 _id를 애플리케이션에서 조합할 수 있도록 되어 있습니다. 메타정보 URI 가 아래와 같다고 하면

<https://token-cdn-domain/{id}.json>

애플리케이션에서는 {id}를 _id 값으로 대체합니다. _id는 uint256이고 이것을 0x가 없는 16진수로 바꾸어 대체합니다. 예를 들어 314592번 토큰의 메타정보 URI는 다음과 같습니다.

<https://token-cdn-domain/0000000000000000...000000000000000000004cce0.json>

ERC-1155 표준에서도 토큰 발행에 대한 내용은 없습니다. 토큰 발행은 전적으로 구현하기 나름입니다. 기본적으로는 토큰 유형을 정하고 각 토큰이 가질 수 있는 수량을 정하여 발행하면 됩니다. 예를 들어 다음과 같이 생성자에서 발행할 수도 있습니다. 아래 예제에서 발행한 토큰은 컨트랙트를 배포한 계정이 소유하게 됩니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";

contract GameItems is ERC1155 {

    uint256 public constant COPPER = 0;
    uint256 public constant CRYSTAL = 1;
    uint256 public constant ELDER_SWORD = 2;
    uint256 public constant KNIFE = 3;
    uint256 public constant WAND = 4;

    constructor() public ERC1155("https://game.example/api/item/{id}.json") {
        _mint(msg.sender, COPPER, 10**18, "");
        _mint(msg.sender, CRYSTAL, 10**27, "");
        _mint(msg.sender, ELDER_SWORD, 1, "");
        _mint(msg.sender, KNIFE, 10**9, "");
        _mint(msg.sender, WAND, 10**9, "");
    }
}
```

ERC-1155 표준은 토큰의 name과 symbol을 포함하지 않습니다. symbol은 서로 다른 토큰이 같은 것을 사용할 수도 있고 또 name은 메타정보에 넣을 수도 있기 때문에 컨트랙트에 저장하는 것이 의미가 없기 때문입니다.

ERC-721은 ERC721Enumerable이라는 인터페이스를 규정하고 있지만 ERC-1155에서는 표준을 간소하게 유지하기 위해 생략했습니다. 관련된 정보들은 이벤트 로그를 이용하도록 권장하고 있습니다. TransferSingle, TransferBatch, 그리고 URI 이벤트가 발생할 때마다 이러한 정보를 별도의 데이터베이스에 저장할 수 있습니다.

두 표준을 간단히 비교하면 다음과 같습니다.

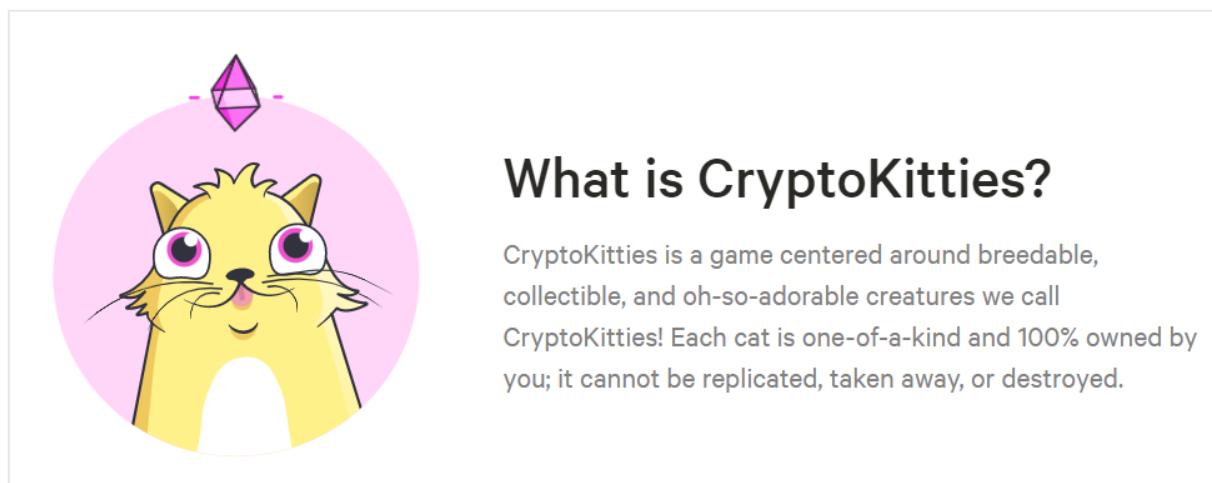
	ERC-721	ERC-1155
Non-fungible	2^{256}	2^{256}
Fungible	X	2^{256} (NFT 1개당)
배치(batch) 전송	X	O

3.5. NFT의 유형

이번에는 어떤 유형의 NFT들이 존재하는지 알아보도록 하겠습니다. 물론 여기에 제시된 것 외에도 향후 새로운 형태의 NFT가 출현할 가능성도 많습니다.

3.5.1. 디지털 수집품

이더리움에서 가장 인기를 끌었던 NFT 중 하나는 아마 "크립토키티(CryptoKitties)"일 것 같습니다. 2017년 말 암호화폐 붐으로 떠들썩하던 시기에 큰 관심을 끌었고 덕분에 폭주하는 트래픽을 제대로 처리하지 못하면서 이더리움의 성능 이슈가 부각되기도 했습니다.



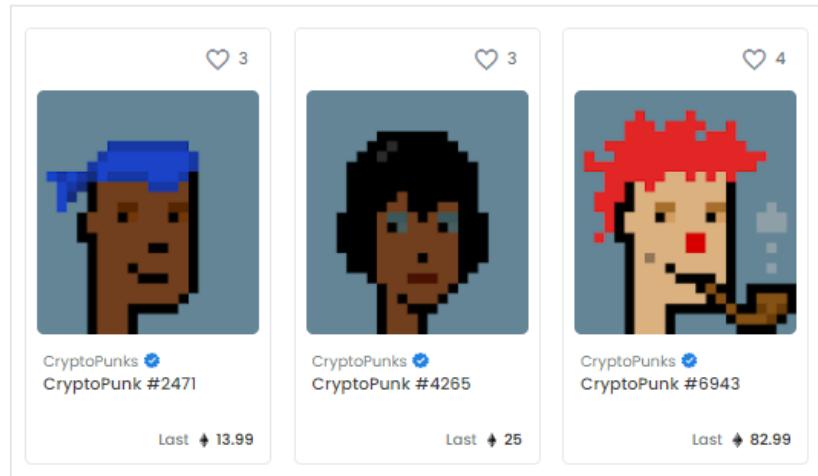
크립토키티는 소위 말하는 "디지털 수집품"의 대명사로 지금 유행하는 수많은 "크립토" 수집품의 시초라고 할 수 있을 것 같습니다. 다양한 특징들을 조합하고 서로 "교배"시키고, 그렇게 만들어진 "품종"을 거래하는 것은 여전히 유효한 NFT의 모델입니다.

하지만 누구나 쉽게 복제 가능한 디지털 이미지는 원본의 의미, 그 가치에 관한 근본적인 의문이 생깁니다. 소유적인 가치와 존재적인 가치에 대한 철학적인 질문처럼 혼란스럽습니다. 디지털 컨텐츠는 원본과 복제품 구별이 불가능하기 때문에 오히려 소유권 정보가 더 의미가 있다는 주장도 일리는 있습니다.

비슷한 유형으로, 역시 2017년에 나온 거래되는 라바 랩스의 "크립토펑크(Cryptopunk)"가 있습니다. 크립토펑크는 대표적인 픽셀아트 이미지 NFT입니다.

크립토펑크는 가상공간에서 자신의 정체성을 나타내는 아바타의 성격이 있습니다. 또 고가에 거래되다 보니 그것을 소유한 사람의 경제적 지위와 연결되기도 합니다. 흔히 말하는 "프로파일" 또는 "디

지털 플렉스”로 표현하는 과시용 NFT가 바로 크립토펑크라고 할 수 있을 것 같습니다. 마치 람보르기니를 몰고 나타나면 그 사람에 대한 평가가 달라지는 것처럼 말입니다.



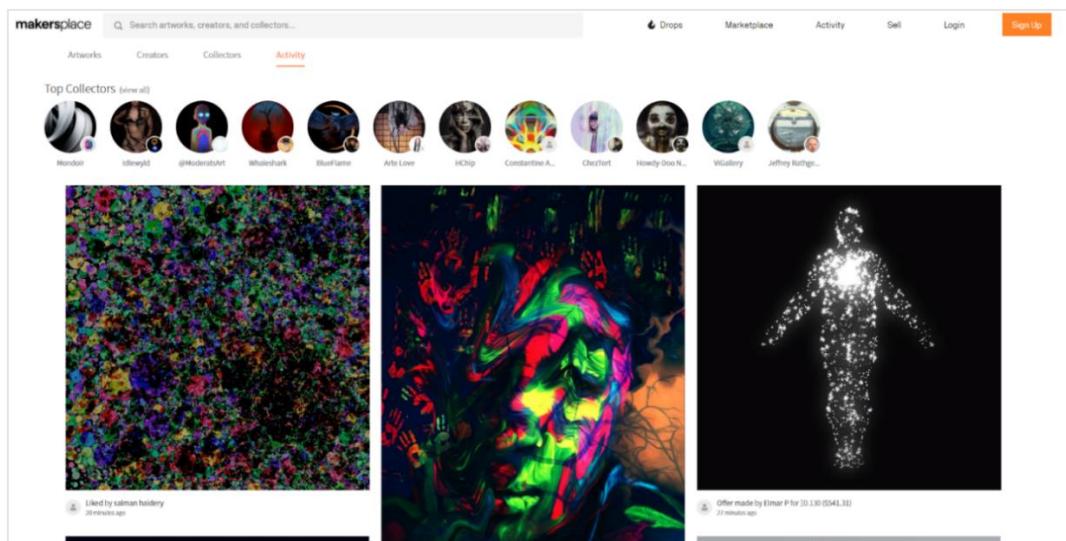
크립토펑크를 시초로 그동안 디지털 페르소나를 표방하는 수많은 NFT가 나왔습니다. 사람이나 동물(특히 원숭이), 로봇, 만화 캐릭터, 상상의 존재를 형상화하고 다양한 악세사리를 결합시킨 디지털 수집품들이 쏟아지고 있습니다. 그러나 한편으로는 이들 NFT들이 어떤 가치를 유지하고 나름의 생태계를 발전시켜 나갈 것인지는 숙제로 남습니다.

3.5.2. 디지털 아트

NFT의 또 다른 활용으로, 기성작가나 디지털 아트 분야에서 활동하는 작가들이 자신의 작품을 NFT로 만들 수 있습니다. 예술가들이 직접 스마트 컨트랙트나 애플리케이션을 개발하지 않고도 오픈마켓에서 제공하는 기능을 통해 자신의 작품을 전세계 사람들에게 전시하고 판매할 수 있습니다.

당연한 이야기지만 현실에서 유명한 작가들의 작품은 이미 NFT 시장에서도 그야말로 터무니없는 가격에 판매되고 있습니다. 디지털 아트 NFT 역시 작품 그 자체 보다는 소유권을 구매하는 것이므로 일반적인 예술품 경매와는 조금 다른 측면이 있습니다.

실제 원본을 구매자에게 전달하는 과정이 생략될 수 있으므로 저작권을 무시하고 허락없이 해당 작품의 NFT를 임의로 발행하는 경우도 있습니다. 작품과 NFT 사이의 연관성이 전혀 없는, 가치 없는 NFT를 만들어낼 수 있는 것입니다.



작가가 제출한 작품이 원본이 아닌 경우도 있습니다. NFT 마켓플레이스의 중앙 서버에 크기가 작은 사본이 저장되어 있기도 하고, 탈중앙화 파일 시스템(IPFS)에 업로드 되더라도 실제 원본이 아닐 가능성은 충분히 있습니다.

만약 이미지를 저장한 서버가 사라진다면 내가 구매한 작품은 어떻게 되는 것일까요? 소유권을 증명한다는 것과 작품 그 자체를 소장한다는 것은 어떤 차이가 있을까요? 또 NFT를 가지고 있다고 해서 그것이 현실에서 법적인 보호를 받을 수 있을까요?

3.5.3. 제너레이티브 NFT

보통 이미 완성되어 존재하는 디지털 작품에 대해 NFT를 발행하는 것이 일반적이라고 생각하지만 발행 직전까지도 존재하지 않는 NFT를 판매하는 경우도 있습니다.

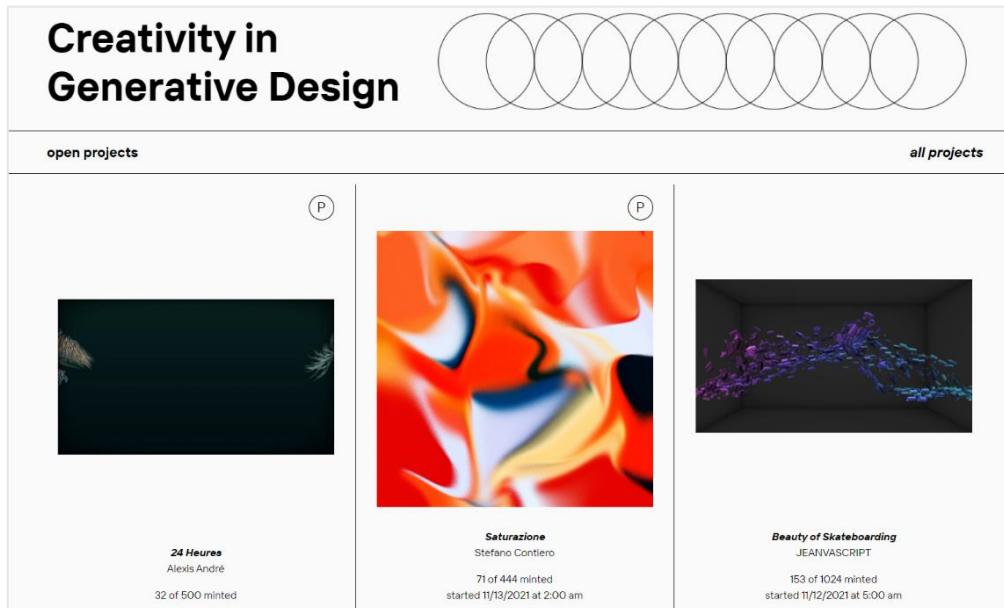
가장 유명한 제너레이티브(Generative) NFT 플랫폼은 “아트 블록(Art Blocks)”입니다. 디지털 아트 분야의 작가들과 전문가들이 알고리즘에 의해 생성된 작품을 평가하고 큐레이션 하는 플랫폼입니다. 작가들이 작품을 구성하는 요소들을 알고리즘으로 미리 만들어 두고 NFT 발행 시점에 랜덤하게 그림이 생성되는 것입니다.

<https://www.artblocks.io/>

확률적으로 동일한 작품이 나올 수 없고 또 어떤 결과물이 만들어질지도 모릅니다. 이렇게 만들어진 작품들 중에 특히 전문가들이 높게 평가하는 작품들은 고가에 판매되고 있습니다.

이러한 예술의 창작 과정은 이전에는 없던 새로운 것입니다. 작가들은 작품을 만드는 것이 아니라

프로그램을 만들고 실제 작품은 수집가들에 의해 NFT로 발행, "mint"되는 것입니다.



3.5.4. 유틸리티 NFT

이제까지는 소유(때로는 플렉스나 시세 차익)가 목적이었다면 이번에는 실용적인 NFT를 살펴보겠습니다. 이러한 유형의 NFT와 가장 비슷한 형태는 영화표입니다.

영화표는 정해진 시간과 장소에서 영화를 볼 수 있는 권리를 제공합니다. 유일하며 영화 시작 전까지는 가치가 유지되고 나중에는 그 경험을 기념하는 증표가 될 수도 있습니다.

"스토너 캣츠(Stoner Cats)"라는 애니메이션 프로젝트(<https://www.stonercats.com/cats>)가 있습니다. 애니메이션을 제작하면서 이것을 볼 수 있는 티켓을 NFT로 만들어서 판매했습니다. 단순히 시청할 수 있는 권한만을 주는 것이 아니라 여러 가지 혜택을 부여한 토큰이었습니다. 이더리움을 만든 비탈릭 부테린이 성우로 참여하면서 화제가 되기도 했습니다.

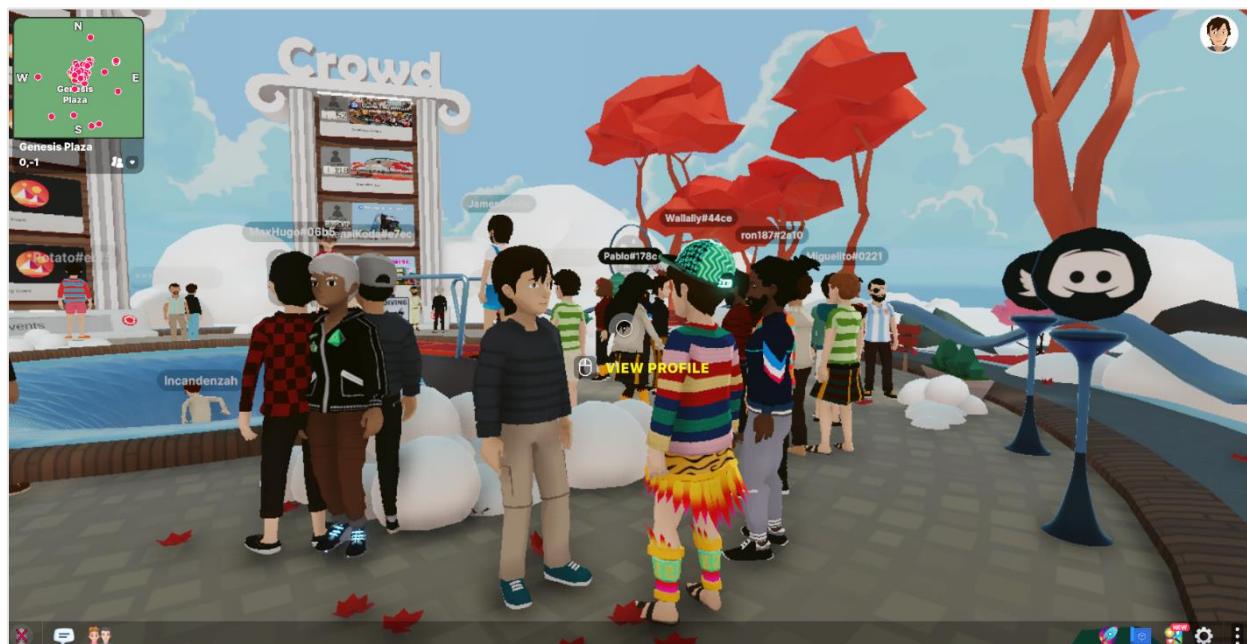
블록체인은 유무형의 가치를 토큰화하여 자유롭게 거래할 수 있도록 설계된 플랫폼입니다. 각 산업 분야, 특히 엔터테인먼트 분야에서는 이러한 유무형의 가치를 마케팅에 활용할 수 있는 기회가 상대적으로 많습니다.

이를 테면 아이돌 그룹이 NFT를 발행하고 그 NFT에 어떤 가치, 한정판 음반을 구매할 수 있는 기회, 공연장 특별석 제공 등의 혜택이 주어지는 NFT를 발행한다면? 물론 그런 혜택을 누리려는 사람들도 많겠지만 단기 차익을 추구하는 투기 등의 부작용이 생길 가능성도 있긴 합니다. 하지만 그것은 유명 가수의 콘서트나 한국시리즈 마지막 결승전 티켓에서도 벌어지는 일이기도 합니다.

3.5.5. 게임과 메타버스

처음부터 가상세계인 게임 분야에서는 역시 태생적으로 가상자산인 NFT가 매우 유용하게 사용될 수 있습니다. 물론 과거에도 게임 아이템을 사적으로 거래하는 경우가 있었지만 아이템 소유권이라는 관점에서 보면 소유권을 완벽하게 보장해줄 수 있는 퍼블릭 블록체인이야말로 최적의 플랫폼이라고 할 수 있습니다. 더구나 글로벌한 마켓플레이스에서 자유롭게 거래가 이루어진다면 게임과 NFT의 시너지 효과는 폭발적일 것이라고 예상하는 것은 어렵지 않습니다.

게임 내의 아이템을 NFT으로 만든다는 개념은 요즘 가장 많은 관심을 끌고 있는 메타버스까지 확장될 수 있습니다. 현실의 복잡한 소유관계들을 가상세계인 메타버스로 그대로 옮긴다면 당연히 NFT를 떠올릴 수 밖에는 없는 것입니다.



현실의 모든 것들, 정치, 사회, 경제 등의 영역이 DAO(Decentralized Autonomous Organization), 아바타, 가상 부동산, 마켓플레이스 등으로 구현되어 메타버스 안에 들어갈 수 있습니다. 이미 많은 기업들이 그 잠재력을 인식하고 메타버스의 주도권을 잡기 위해 뛰어들고 있습니다. 그 중심에 NFT가 있는 것입니다.

3.5.6. 1차 시장(Primary market)과 2차 시장(Secondary market)

한 번쯤 이름을 들어보았을 크리스티 경매에서 “비플(Beeples)”이라는 작가의 NFT 가 약 700억원이 넘는 금액에 낙찰된 적이 있습니다. 이렇게 판매되는 NFT는 큰 화제가 되고 또 봄으로 이어지는 계

기가 되기도 합니다. 디지털 아트 분야의 유명 NFT를 보면(<https://cryptoart.io/>) 놀라움 그 자체입니다. 작품 자체가 놀랍다고 하기 보다는 그 평가 금액에 놀라는 것입니다.

처음 NFT 가 발행되어 판매되는 시장을 1차 시장이라고 합니다. 1차 시장에서 유명 작가의 NFT들은 굉장히 높은 가격으로 거래가 됩니다.

1차 시장에서 판매된 NFT가 OpenSea(<https://opensea.io/>)와 같은 곳에 다시 매물로 나와서 거래되는 시장을 2차 시장이라고 합니다. 그런데 2차 시장, 즉 재판매 시장에서는 그 거래량이 급격히 줄어든다는 통계가 있습니다. 이것은 투자 측면에서 매우 조심해야 할 부분이고 동시에 NFT의 가치평가에 그만큼 거품이 있다는 이야기도 될 수 있습니다.

그렇게 고가에 구매하는 사람들조차도 그 본질적인 가치 때문에 큰 돈을 지불한 것이 아니라 단지 NFT를 마케팅 수단으로 삼으려는 사업적인 계산에 의한 경우도 있습니다. 소위 말하는 자전 거래도 존재합니다. 실제로 크립토팡크가 6천억원이 넘는 금액에 팔린 적이 있었는데 거래를 추적해보니 매도자와 매수자가 서로 짜고 대출받은 암호화폐를 주고받으면서 그 가치를 부풀린 것이었습니다.

NFT 1차 시장과 2차 시장의 거래비중에 대한 통계는 다양한 크립토 데이터를 분석해서 보여주는 둔(<https://dune.xyz>)에서 찾아볼 수 있습니다.

@rchen8 / SuperRare primary vs. secondary volume – <https://dune.xyz/queries/7700>

3.6. 정리

이번 장에서는 NFT의 표준과 NFT가 실제로 어떻게 구현되어 시장에 나타나는지 간단하게 알아보았습니다. 여기서 다룬 NFT 표준 ERC-721과 1155는 NFT와 관련된 애플리케이션을 개발할 때 반드시 숙지해야 할 표준 스펙들입니다. 인터페이스에 정의된 함수들이 요구하는 것들을 잘 지켜서 구현해야만 다른 애플리케이션들과 수월하게 연동될 수 있습니다.

오픈제펠린(OpenZeppelin)



4장에서는 솔리디티 스마트 컨트랙트 개발에 가장 많이 활용되고 있는 오픈제펠린의 컨트랙트에 대해 알아보겠습니다.

4.1. 컨트랙트

오픈제펠린은 스마트 컨트랙트와 관련된 소프트웨어를 만드는 개발팀입니다. 또 보안 취약점을 분석하는 컨트랙트 감사(audit)도 같이 수행하고 있습니다. 다음 링크를 방문하면 자세한 정보를 얻을 수 있습니다.

<https://openzeppelin.com/>

오픈제펠린에는 크게 두 가지 솔루션이 있습니다. 하나는 컨트랙트 구현체와 라이브러리를 모아놓은 “컨트랙트(Contracts)”이고 또 다른 하나는 “디펜더(Defender)”라고 하는 컨트랙트 개발 및 운영 관리 플랫폼입니다. 여기서는 오픈제펠린의 컨트랙트를 중심으로 살펴보겠습니다.

현재 시점을 기준으로 오픈제펠린 컨트랙트는 4.x까지 출시되었습니다. 재사용 가능한 컴포넌트와 라이브러리, 유ти리티, 이더리움 토큰 표준을 구현한 컨트랙트 그리고 단위 테스트 라이브러리까지 다양한 개발 자원들을 제공합니다.

오픈제펠린이 제공하는 컨트랙트들을 사용하려면 컨트랙트 자체를 수정해서 활용하는 것이 아니라 상속을 받아서 쓰는 것을 권장하고 있습니다. 2장에서 설명한 것처럼 상속을 통해서 기능을 확장하는 것입니다.

예를 들어서 접근 제어 기능을 제공하는 AccessControl 컨트랙트를 상속받고 사용할 필요가 없는 함수 revokeRole()이 있다면 다음과 같이 작성하면 되겠습니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";

contract MyAccessControl is AccessControl {
    function revokeRole(bytes32, address) public override {
        revert("MyAccessControl: cannot revoke roles");
    }
}
```

4.2. 접근 제어(Access Control)

4.2.1. Ownable

우선 오픈제펠린에서 가장 많이 활용되는 접근 제어 기능부터 살펴보겠습니다. 오픈제펠린 컨트랙트는 npm으로 설치할 수 있습니다.

```
npm install @openzeppelin/contracts
```

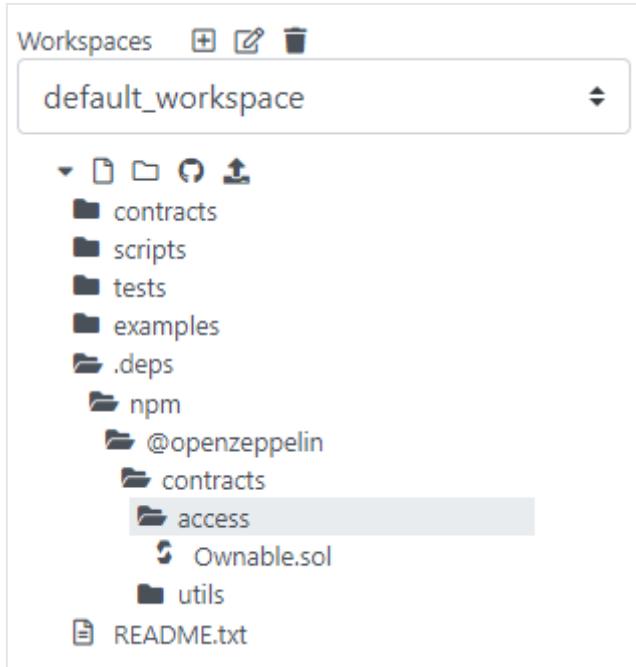
또는 2장에서 배운 리믹스를 사용할 수 있습니다. 리믹스에서는 별도의 설치 과정 없이 import를 하면 자동으로 웹브라우저 내부에 설치가 됩니다.

예를 들어 아래와 같이 작성한 후 컴파일을 실행하면 자동으로 .deps 폴더가 생성되면서 Ownable 컨트랙트가 설치되기 때문에 수동으로 복사할 필요가 없습니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";

contract MyContract is Ownable {
    function foo() public {
    }
}
```



Ownable이라는 이름에서 알 수 있듯이 컨트랙트를 소유하는, 즉 컨트랙트를 관리하는 기능을 제공하는 기본 컨트랙트입니다. Ownable 컨트랙트의 전체 소스는 다음과 같습니다.

```
// SPDX-License-Identifier:MIT
// OpenZeppelin Contracts v4.4.1 (access/Ownable.sol)

pragma solidity ^0.8.0;

import "../utils/Context.sol";

abstract contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    constructor() {
        _transferOwnership(_msgSender());
    }

    function owner() public view virtual returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }

    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
    }
}
```

```

        _transferOwnership(newOwner);
    }

    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}

```

생성자에서는 컨트랙트를 배포하는 계정을 `_owner`라는 `private` 상태 변수에 저장합니다. 또 소유 계정을 바꿀 수 있는 `transferOwnership()`이라는 함수를 제공합니다. 당연히 이 함수는 `onlyOwner`라는 modifier가 적용되어 있고 현재 `_owner` 계정만이 실행할 수 있습니다.

`renounceOwnership()`는 소유권을 `address(0)`으로 지정함으로써 컨트랙트의 소유권을 아무도 갖지 않도록(소유권 포기) 하는 것입니다.

모든 함수들이 `virtual`로 정의되어 있기 때문에 상속을 받는 컨트랙트에서 재정의가 가능합니다. 구현되지 않은 함수가 없음에도 불구하고 `abstract`라고 되어 있는 것은 이 컨트랙트들이 단독으로 배포될 수 없다는 것을 의미합니다. 즉 `Ownable`이나 `Context` 컨트랙트는 상속에 의해서만, 자식 컨트랙트가 배포될 때만 사용될 수 있습니다.

4.2.2. 역할(Role) 기반의 접근제어

`Ownable` 컨트랙트는 배포 계정이 절대적인 권한을 가지는 형태였다면 역할에 따라서 컨트랙트의 특정 기능을 허용하거나 제한할 수 있는 여러 레벨의 관리자들이 존재하는 경우도 있습니다. 예를 들어 토큰 가격을 조정할 수 있는 계정과 컨트랙트를 중지할 수 있는 계정의 역할을 구분할 필요가 있을지도 모릅니다.

오픈제ppelin 컨트랙트는 솔리디티의 객체 지향적인 특징을 활용하여 확장 가능한 구조로 설계되어 있습니다. 역할 기반의 접근제어도 그런 관점에서 `IAccessControl`이라는 인터페이스로 정의되어 있고 그것을 구현한 `AccessControl` 컨트랙트를 제공합니다.

`IAccessControl`에 정의된 인터페이스는 아래와 같습니다.

```

// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (access/IAccessControl.sol)

pragma solidity ^0.8.0;

interface IAccessControl {
    event RoleAdminChanged(bytes32 indexed role,

```

```

        bytes32 indexed previousAdminRole,
        bytes32 indexed newAdminRole);
event RoleGranted(bytes32 indexed role,
                  address indexed account,
                  address indexed sender);

event RoleRevoked(bytes32 indexed role,
                  address indexed account,
                  address indexed sender);

function hasRole(bytes32 role, address account) external view returns (bool);

function getRoleAdmin(bytes32 role) external view returns (bytes32);

function grantRole(bytes32 role, address account) external;

function revokeRole(bytes32 role, address account) external;

function renounceRole(bytes32 role, address account) external;
}

```

AccessControl에서는 role은 역할을 나타내는 문자열을 해시한 값으로 나타냅니다. 예를 들어 토큰 발행 역할을 "MINTER_ROLE"이라고 정한다면 컨트랙트 상에서의 역할 이름은 다음과 같이 표현할 수 있겠습니다.

```
bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
```

이제부터 각 함수들이 구현해야 할 것들이 무엇인지 살펴보도록 하겠습니다.

- `hasRole(bytes32 role, address account) returns (bool)`

계정 account가 지정된 역할 role을 가지고 있는지 여부를 true 또는 false로 리턴합니다.

- `getRoleAdmin(bytes32 role) returns (bytes32)`

특정 역할을 관리하는 역할(이것을 "admin role"이라고 합니다)을 리턴합니다. 모든 역할에는 그 역할을 관리하는 계정을 설정할 수 있습니다.

접근 제어를 위해서는 역할을 부여할 수 있는 관리자가 있어야 합니다. 예를 들어 토큰 발행 역할을 누군가에게 부여하려면 그런 역할을 부여할 수 있는 사람 역시 어떤 권한을 가지고 있어야 합니다. AccessControl 컨트랙트는 처음에 관리자 역할 DEFAULT_ADMIN_ROLE을 "0x00"로 설정하여 임의의 계정이 역할을 부여할 수 있도록 되어 있습니다.

그러나 특정 계정에 admin role을 주는 경우에는, 예를 들어 생성자에서 DEFAULT_ADMIN_ROLE을 배포 계정으로 설정하면 배포 계정만이 역할을 부여하거나 회수하는 권한을 가지게 됩니다.

```
constructor() {
    // Grant the contract deployer the default admin role: it will be able
    // to grant and revoke any roles
    _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
}
```

- `grantRole(bytes32 role, address account)`

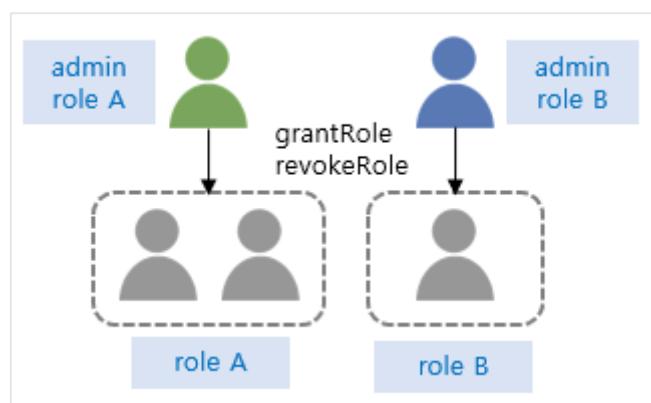
계정 `account`에게 `role`을 부여합니다. 최초로 해당 역할을 부여받는 경우에는 `RoleGranted` 이벤트를 발생해야 합니다. 이 함수를 실행하려면 그 역할을 관리하는 권한 즉 `admin role`을 가지고 있어야 합니다.

예를 들어 `grantRole()`에는 항상 modifier `onlyRole`이 걸려있기 때문에 특정 `admin role`을 가진 계정이 있는 경우에는 그 계정만이 이 함수를 실행할 수 있습니다.

```
function grantRole(bytes32 role, address account)
    public
    override
    onlyRole(getRoleAdmin(role)) {
    ...
}
```

- `revokeRole(bytes32 role, address account)`

`grantRole()`과 반대로 역할을 회수하는 함수입니다. `RoleRevoked` 이벤트를 발생시켜야 합니다.



- renounceRole(bytes32 role, address account)

이 함수를 호출하는 계정은 반드시 account와 동일해야 합니다. 즉 account가 스스로 role을 반납하는 함수입니다. 이 경우는 admin role을 가진 계정이 다른 사람에 의해 잘못 사용되는 경우(정보 유출 등의 원인) 즉 grantRole()이나 revokeRole()이 정상적으로 동작할 수 없는 경우를 대비한 함수입니다.

- RoleGranted, RoleRevoked, RoleAdminChanged

RoleGranted와 RoleRevoked 이벤트는 역할을 부여하거나 회수할 때 발생하고 RoleAdminChanged는 admin role을 변경할 때 발생합니다.

이제 구현체인 AccessControl 컨트랙트를 살펴보겠습니다.

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (access/AccessControl.sol)

pragma solidity ^0.8.0;

import "./IAccessControl.sol";
import "../utils/Context.sol";
import "../utils/Strings.sol";
import "../utils/introspection/ERC165.sol";

abstract contract AccessControl is Context, IAccessControl, ERC165 {
    struct RoleData {
        mapping(address => bool) members;
        bytes32 adminRole;
    }

    mapping(bytes32 => RoleData) private _roles;

    bytes32 public constant DEFAULT_ADMIN_ROLE = 0x00;

    modifier onlyRole(bytes32 role) {
        _checkRole(role, _msgSender());
        ;
    }

    function supportsInterface(bytes4 interfaceId) public view
        virtual override returns (bool) {
        return interfaceId ==
            type(IAccessControl).interfaceId || super.supportsInterface(interfaceId);
    }

    function hasRole(bytes32 role, address account) public view override returns (bool) {
        return _roles[role].members[account];
    }

    function _checkRole(bytes32 role, address account) internal view {
        if (!hasRole(role, account)) {
            revert(
                string(
                    abi.encodePacked(
                        "AccessControl: account ",
                        account,
                        " is not authorized for role ",
                        bytes32(role),
                        " because caller ",
                        _msgSender(),
                        " is not the admin role "
                    )
                )
            );
        }
    }
}
```

```

        "AccessControl: account ",
        Strings.toHexString(uint160(account), 20),
        " is missing role ",
        Strings.toHexString(uint256(role), 32)
    )
)
);
}

function getRoleAdmin(bytes32 role) public view override returns (bytes32) {
    return _roles[role].adminRole;
}

function grantRole(bytes32 role, address account)
    public virtual override onlyRole(getRoleAdmin(role)) {
    _grantRole(role, account);
}

function revokeRole(bytes32 role, address account)
    public virtual override onlyRole(getRoleAdmin(role)) {
    _revokeRole(role, account);
}

function renounceRole(bytes32 role, address account) public virtual override {
    require(account == _msgSender(), "AccessControl: can only renounce roles for self");
    _revokeRole(role, account);
}

/* deprecated */
function _setupRole(bytes32 role, address account) internal virtual {
    _grantRole(role, account);
}

function _setRoleAdmin(bytes32 role, bytes32 adminRole) internal virtual {
    bytes32 previousAdminRole = getRoleAdmin(role);
    _roles[role].adminRole = adminRole;
    emit RoleAdminChanged(role, previousAdminRole, adminRole);
}

function _grantRole(bytes32 role, address account) internal virtual {
    if (!hasRole(role, account)) {
        _roles[role].members[account] = true;
        emit RoleGranted(role, account, _msgSender());
    }
}

function _revokeRole(bytes32 role, address account) internal virtual {
    if (hasRole(role, account)) {
        _roles[role].members[account] = false;
        emit RoleRevoked(role, account, _msgSender());
    }
}
}

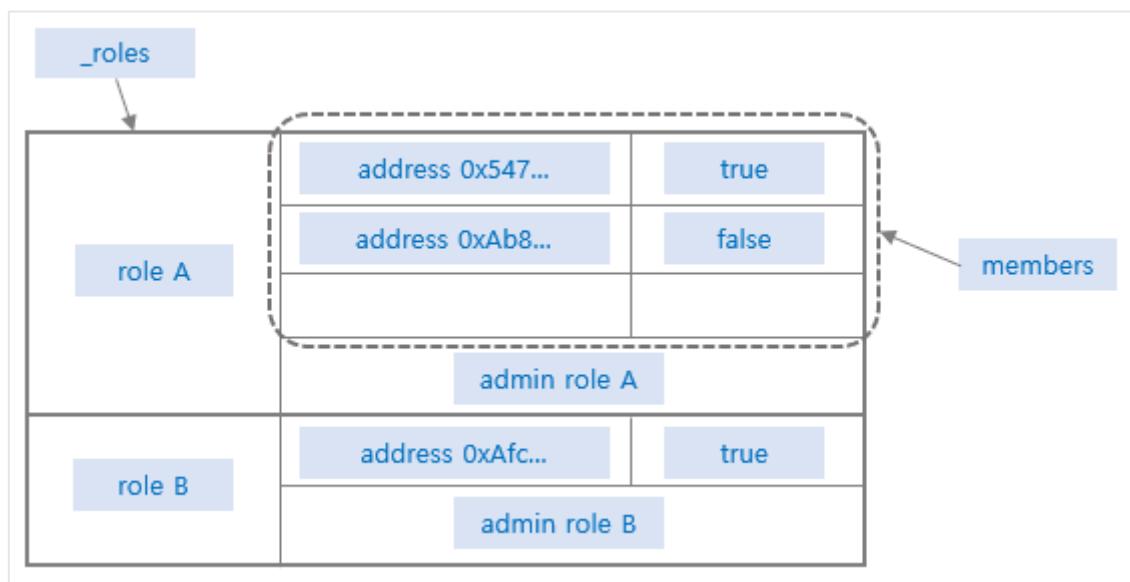
```

먼저 ERC-165 표준인 supportsInterface 통해 IAccessControl 인터페이스를 구현했음을 나타냅니다.
앞서 설명한 인터페이스에 정의된 함수와 이벤트들이 구현되어 있습니다.

각 계정의 역할을 저장하기 위해 mapping 타입 `_roles`를 정의했습니다. 역할은 `RoleData`라는 구조체로 맵핑되어 있습니다. `RoleData` 안에 다시 계정 목록을 나타내는 mapping 타입의 `members`가 있습니다. 계정이 해당 역할을 가지고 있다면 `true`가 나올 것입니다.

```
struct RoleData {
    mapping(address => bool) members;
    bytes32 adminRole;
}

mapping(bytes32 => RoleData) private _roles;
```



`RoleData` 안에 이 역할을 관리하는 `adminRole`이 정의되어 있는데 이 값은 `DEFAULT_ADMIN_ROLE` 즉 `0x00`에 해당합니다(`bytes32`의 초기값).

각 역할들을 나누고 부여하려면 `internal`로 정의된 `_grantRole()`을 생성자에서 호출합니다. 전달된 계정이 이미 그 역할을 가지 있는지 검사하고 없는 경우 역할을 추가합니다.

예를 들어 아래와 같이 코드를 작성할 수 있습니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";

contract MyContract is AccessControl {
```

```

bytes32 public constant ROLE_A_ADMIN = keccak256("ROLE_A_ADMIN");
bytes32 public constant ROLE_A = keccak256("ROLE_A");
bytes32 public constant ROLE_B = keccak256("ROLE_B");
bytes32 public constant ROLE_C = keccak256("ROLE_C");

constructor(address _superAdmin, address _adminA) {

    _grantRole(DEFAULT_ADMIN_ROLE, _superAdmin);
    _grantRole(ROLE_A_ADMIN, _adminA);
    _setRoleAdmin(ROLE_A, ROLE_A_ADMIN);

}
}

```

생성자에서 _superAdmin과 _adminA 계정을 받아서 각각 DEFAULT_ADMIN_ROLE과 ROLE_A_ADMIN 을 부여합니다. _superAdmin은 DEFAULT_ADMIN_ROLE 역할이기 때문에 이후 grantRole을 호출하여 특정 계정에게 역할을 부여할 수 있습니다.

그런데 _setRoleAdmin()을 사용하여 ROLE_A를 부여할 수 있는 계정은 ROLE_A_ADMIN 역할을 가지도록 합니다. 따라서 _adminA 계정만이 다른 계정에게 ROLE_A 역할을 줄 수 있습니다. _superAdmin 은 ROLE_B와 ROLE_C를 부여할 수 있는 권한을 갖게 됩니다.

외부 호출이 가능한 grantRole() 또는 revokeRole()을 실행할 때는 항상 그것을 실행할 수 있는 계정 이 그럴 자격이 있는지 검사합니다. 이들 함수에는 modifier onlyRole()가 걸려있습니다. 파라미터로 넘겨주는 getRoleAdmin(role)은 함수를 호출한 계정(msg.sender)이 해당 역할을 부여하거나 회수할 권한을 가지고 있는지 검사하기 위한 것입니다.

```

modifier onlyRole(bytes32 role) {
    _checkRole(role, _msgSender());
    _;
}

```

checkRole()에서는 hasRole() 함수를 호출하여 역할을 검사합니다. hasRole()은 역할 테이블 _roles에서 전달된 계정이 역할을 가지고 있는지 여부를 true 또는 false로 리턴합니다. revokeRole()과 renounceRole()도 같은 방식으로 동작합니다.

사용자가 정의한 함수에 역할을 적용하려면 require() 조건을 사용하면 되겠습니다.

```

function foo(uint256 _v) external {

    require(hasRole(ROLE_A, msg.sender));
    ...
}

```

다음에는 토큰 인터페이스와 구현체에 대해 알아보겠습니다. 오픈제펠린 토큰 컨트랙트는 크게 ERC-20, ERC-721, ERC-777, ERC-1155를 제공합니다. ERC-20과 ERC-721의 개선 표준이 각각 ERC-777과 ERC-1155입니다. 여기서는 NFT 표준인 ERC-721과 ERC-1155 두 가지를 알아보겠습니다.

4.3. ERC-721

오픈제펠린의 ERC-721 컨트랙트는 표준에서 정의한 모든 인터페이스를 구현한 컨트랙트 모음입니다. 3장에서 ERC-721 표준에 대해 설명한 것처럼 기본 인터페이스 IERC-721과 IERC721MetaData 그리고 IERC721Enumerable의 구현체를 제공합니다.

부가적으로 다양한 경우에 적용할 수 있는 컨트랙트들도 구현되어 있습니다. 토큰 전송을 중단하는 ERC721Pausable과 발행된 토큰을 없앨 수 있는 ERC721Burnable 그리고 ERC721URIStorage는 토큰 별로 메타 데이터를 저장할 수 있는 컨트랙트들입니다.

- _baseURI

ERC721 컨트랙트는 IERC721Enumerable을 제외한 IERC-721과 IERC721MetaData을 구현한 컨트랙트입니다. 이 구현 컨트랙트의 특징 중 하나는 각 토큰의 메타정보를 가리키는 tokenURI()를 생성할 때 URI의 공통적인 부분을 컨트랙트에 _baseURI() 함수로 미리 저장해 놓는다는 것입니다.

보통 메타정보 URI는 인터넷에 노출된 도메인 주소와 토큰 번호로 구성되는 경우가 많은데, 이를테면 메타정보의 위치가 "https://www.myserver.com/{tokenId}"의 형태라면 _baseURI()가 리턴하는 값은 토큰 번호를 제외한 "http://www.myserver.com/"에 해당합니다. 그래서 다음과 같이 구현되어 있습니다. _baseURI()에서 공통적인 URI를 리턴하게 하면 메타정보의 위치를 보다 효율적으로 저장할 수 있게 되는 것입니다.

```
function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
    require(_exists(tokenId), "ERC721Metadata: URI query for nonexistent token");
    string memory baseURI = _baseURI();
    return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI, tokenId.toString())) : "";
}

function _baseURI() internal view virtual returns (string memory) {
    return "";
}
```

- `_safeMint`

ERC-721에서는 NFT를 어떻게 발행하는지에 대한 표준은 정하지 않았지만 오픈제펠린은 `_safeMint()`와 `_mint()` 함수 두 가지를 제공합니다. 함수 이름에서 짐작할 수 있듯이 `_safeMint()`는 발행된 토큰의 소유 계정이 컨트랙트일 때 상대 컨트랙트가 `onERC721Received` 함수를 구현한 컨트랙트인지 검사하여 토큰의 전송 오류를 방지합니다.

```
function _safeMint(address to, uint256 tokenId, bytes memory _data) internal virtual {
    _mint(to, tokenId);
    require(_checkOnERC721Received(address(0), to, tokenId, _data),
        "ERC721: transfer to non ERC721Receiver implementer"
    );
}
```

발행을 위해서는 토큰 번호와 발행된 토큰을 소유하는 계정 주소를 넘겨주어야 합니다. 표준에 따라 소유 계정은 `address(0)`이 될 수 없고 이미 존재하는 토큰이라면 예외를 발생시켜야 합니다. 또 토큰 발행 전후에 처리해야 할 일들을 각각 `_beforeTokenTransfer()`와 `_afterTokenTransfer()`에서 처리할 수 있도록 하고 있습니다.

```
function _mint(address to, uint256 tokenId) internal virtual {
    require(to != address(0), "ERC721: mint to the zero address");
    require(!_exists(tokenId), "ERC721: token already minted");

    _beforeTokenTransfer(address(0), to, tokenId);

    _balances[to] += 1;
    _owners[tokenId] = to;

    emit Transfer(address(0), to, tokenId);

    _afterTokenTransfer(address(0), to, tokenId);
}
```

- `ERC721Pausable`

`_beforeTokenTransfer()`은 `ERC721Pausable` 컨트랙트에서 재정의하는 함수로 어떤 조건에 따라 토큰 발행을 중지시키는 기능으로 사용할 수 있습니다.

`ERC721Pausable`은 `security/Pausable` 컨트랙트를 상속받는데 `Pausable`은 불리언 타입의 `_pause` 상태 변수를 `true`나 `false`로 지정하게 되고 `_beforeTokenTransfer()`에서 그 조건을 검사합니다.

```
abstract contract ERC721Pausable is ERC721, Pausable {
```

```

function _beforeTokenTransfer(
    address from,
    address to,
    uint256 tokenId
) internal virtual override {
    super._beforeTokenTransfer(from, to, tokenId);
    require(!paused(), "ERC721Pausable: token transfer while paused");
}

```

- ERC721Burnable

ERC721Burnable 컨트랙트는 발행된 토큰을 없애는(소각) 기능을 가지고 있습니다. 당연히 토큰의 소유자 또는 위임을 받았는지 검사하는 조건이 있어야 합니다.

```

abstract contract ERC721Burnable is Context, ERC721 {

    function burn(uint256 tokenId) public virtual {
        require(_isApprovedOrOwner(_msgSender(), tokenId),
            "ERC721Burnable: caller is not owner nor approved");
        _burn(tokenId);
    }
}

```

- ERC721URIStorage

이 컨트랙트는 ERC721을 상속받아서 tokenURI()를 재정의한 컨트랙트입니다. 메타정보의 위치를 가리키는 URI를 토큰 번호와 연결시켜 주기 위한 _setTokenURI() 함수가 추가되어 있습니다.

```

function _setTokenURI(uint256 tokenId, string memory _tokenURI) internal virtual {

    require(_exists(tokenId), "ERC721URIStorage: URI set of nonexistent token");
    _tokenURIs[tokenId] = _tokenURI;
}

```

예를 들어 분산 파일 시스템 IPFS에 메타정보를 업로드하는 경우에는 tokenURI가 임의의 컨텐츠 번호일 것이기 때문에 이 값을 전달해서 해당 토큰의 메타정보 URI로 저장할 필요가 있습니다.

아래와 같이 토큰 발행을 하면서 _setTokenURI()를 호출하여 메타정보 위치를 지정합니다. 6장에서 토큰을 발행할 때 ERC721URIStorage 컨트랙트를 이용할 것입니다.

```

function mint(address toAddr, uint256 tokenId, string memory tokenURI)
    public onlyOwner returns (uint256) {

```

```

        uint256 newItemId = tokenId;
        _mint(toAddr, newItemId);
        _setTokenURI(newItemId, tokenURI);
    }
}

```

4.4. ERC-1155

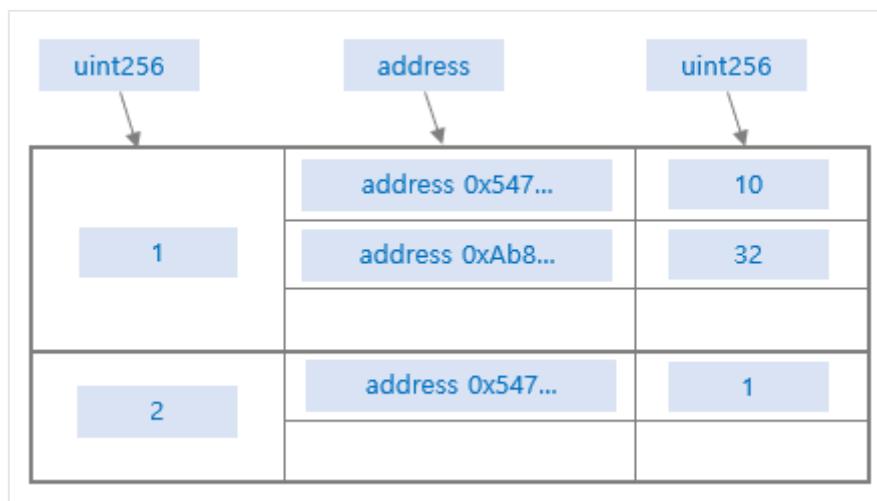
ERC-1155 표준의 특징은 멀티 토큰과 배치 전송이 가능하도록 스펙을 만들었다는 점입니다. NFT의 수량 개념이 있기 때문에, 이를테면 ERC-721의 `balanceOf()`는 중요하지는 않았지만 ERC-1155 토큰은 “semi-fungible”이기 때문에 수량이 의미를 가지게 됩니다.

기본 구현체인 ERC1155 컨트랙트는 `IERC1155`와 `IERC1155MetadataURI`를 구현하고 있습니다. ERC-721과 유사하게 `ERC1155Pausable`과 `ERC1155Burnable` 컨트랙트도 제공됩니다. 이 두 컨트랙트는 ERC-721의 그것과 전달 인자만 차이가 있고 기능은 크게 다르지 않습니다.

토큰의 수납을 기록하는 장부는 조금 복잡한 구조를 가지게 됩니다.

```
mapping(uint256 => mapping(address => uint256)) private _balances;
```

이것을 그림으로 표현하면 아래와 같습니다.



- `_setURI`

ERC1155 컨트랙트는 생성자에서 `_setURI()`를 호출합니다. 표준에서 제시된 것처럼 여러 유형의 토큰들에 대한 메타정보의 URI를 설정할 때 공통 부분을 상태변수 `_uri`에 저장합니다.

```

constructor(string memory uri_) {
    _setURI(uri_);
}

function _setURI(string memory newuri) internal virtual {
    _uri = newuri;
}

```

이렇게 저장된 값은 IERC1155MetadataURI에 정의된 `uri()` 함수로 조회할 수 있습니다. 표준에서는 토큰 번호를 넘겨주도록 되어 있지만 오픈제펠린 구현체에서는 전달인자를 사용하지 않고 `_uri`와 토큰 번호를 조합하여 메타정보 URI를 구성하도록 하고 있습니다.

```

function uri(uint256) public view virtual override returns (string memory) {
    return _uri;
}

```

다시 말해서 컨트랙트는 모든 토큰들의 메타정보 URI를 개별적으로 저장하지 않고 대표 URI만 저장하고 있는 것입니다(구현하기 나름이기 때문에 다른 방식이 얼마든지 있을 수 있겠습니다).

- `_mint`와 `_mintBatch`

"safe"는 붙지 않았지만 토큰 발행 후 다른 컨트랙트로 전송되면 그 컨트랙트는 `onERC1155Received` 함수가 구현되어 있는지 `_doSafeTransferAcceptanceCheck()`에서 검사합니다.

```

function _mint(
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: mint to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, address(0),
        to, _asSingletonArray(id), _asSingletonArray(amount), data);

    _balances[id][to] += amount;
    emit TransferSingle(operator, address(0), to, id, amount);

    _doSafeTransferAcceptanceCheck(operator, address(0), to, id, amount, data);
}

function _doSafeTransferAcceptanceCheck(
    address operator,

```

```

        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) private {

        if (to.isContract()) {
            try IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data)
                returns (bytes4 response) {
                if (response != IERC1155Receiver.onERC1155Received.selector) {
                    revert("ERC1155: ERC1155Receiver rejected tokens");
                }
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("ERC1155: transfer to non ERC1155Receiver implementer");
            }
        }
    }
}

```

`_mintBatch()`는 여러 개의 토큰을 한 번의 호출로 생성하는 함수입니다. 이더리움의 트랜잭션은 함수가 호출될 때마다 기본 트랜잭션 가스가 소모되기 때문에 여러 번 호출하는 경우 비용이 많이 발생합니다. 그래서 이렇게 배치 생성 함수를 작성합니다. 하지만 블록 가스 제한이 있기 때문에 한꺼번에 많은 토큰을 생성하면 트랜잭션이 실패할 수 있습니다.

```

function _mintBatch(
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: mint to the zero address");
    require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, address(0), to, ids, amounts, data);

    for (uint256 i = 0; i < ids.length; i++) {
        _balances[ids[i]][to] += amounts[i];
    }

    emit TransferBatch(operator, address(0), to, ids, amounts);

    _doSafeBatchTransferAcceptanceCheck(operator, address(0), to, ids, amounts, data);
}

```

- ERC1155Supply

오픈제펠린의 ERC1155Supply 컨트랙트는 발행된 각 토큰들의 수량을 파악할 수 있는 기능을 제공

합니다. 이 컨트랙트는 totalSupply() 함수와 토큰이 발행이나 전송될 때 수량 변화를 업데이트하기 위한 _beforeTokenTransfer() 함수를 재정의합니다.

```
pragma solidity ^0.8.0;

import "../ERC1155.sol";

abstract contract ERC1155Supply is ERC1155 {
    mapping(uint256 => uint256) private _totalSupply;

    function totalSupply(uint256 id) public view virtual returns (uint256) {
        return _totalSupply[id];
    }

    function exists(uint256 id) public view virtual returns (bool) {
        return ERC1155Supply.totalSupply(id) > 0;
    }

    function _beforeTokenTransfer(
        address operator,
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) internal virtual override {
        super._beforeTokenTransfer(operator, from, to, ids, amounts, data);

        if (from == address(0)) {
            for (uint256 i = 0; i < ids.length; ++i) {
                _totalSupply[ids[i]] += amounts[i];
            }
        }

        if (to == address(0)) {
            for (uint256 i = 0; i < ids.length; ++i) {
                _totalSupply[ids[i]] -= amounts[i];
            }
        }
    }
}
```

4.5. 보안

4.5.1. Pull Payments

2.15 보안코딩에서 설명한 것처럼 컨트랙트에서 이더를 전송할 때는 push 패턴을 쓰는 것이 바람직합니다. 오픈제펠린 컨트랙트에는 이더 입출금 기능을 구현한 PullPayment 컨트랙트가 있습니다.

```
// SPDX-License-Identifier:MIT
//OpenZeppelin Contracts v4.4.1 (security/PullPayment.sol)

pragma solidity ^0.8.0;

import "../utils/escrow/Escrow.sol";

abstract contract PullPayment {

    Escrow private immutable _escrow;

    constructor() {
        _escrow = new Escrow();
    }

    function withdrawPayments(address payable payee) public virtual {
        _escrow.withdraw(payee);
    }

    function payments(address dest) public view returns (uint256) {
        return _escrow.depositsOf(dest);
    }

    function _asyncTransfer(address dest, uint256 amount) internal virtual {
        _escrow.deposit{value: amount}(dest);
    }
}
```

생성자에서 immutable로 선언된 Escrow 컨트랙트를 생성하는데 이 컨트랙트는 이름에서 알 수 있듯이 안전한 입출금 기능을 제공합니다. mapping 타입의 입출금 장부_deposits와 예치 deposit() 그리고 인출 withdraw() 메소드를 사용할 수 있습니다.

```
// SPDX-License-Identifier:MIT
//OpenZeppelin Contracts v4.4.1 (utils/escrow/Escrow.sol)

pragma solidity ^0.8.0;

import "../../access/Ownable.sol";
import "../Address.sol";

contract Escrow is Ownable {
    using Address for address payable;

    event Deposited(address indexed payee, uint256 weiAmount);
    event Withdrawn(address indexed payee, uint256 weiAmount);
```

```

mapping(address => uint256) private _deposits;

function depositsOf(address payee) public view returns (uint256) {
    return _deposits[payee];
}

function deposit(address payee) public payable virtual onlyOwner {
    uint256 amount = msg.value;
    _deposits[payee] += amount;
    emit Deposited(payee, amount);
}

function withdraw(address payable payee) public virtual onlyOwner {
    uint256 payment = _deposits[payee];
    _deposits[payee] = 0;
    payee.sendValue(payment);
    emit Withdrawn(payee, payment);
}
}

```

이더를 전송할 때 특정 계정 주소로 직접 보내지 않고 `_asyncTransfer()`를 통해서 Escrow 컨트랙트에 보냅니다. 수신 계정이 `withdraw()`를 호출하여 pull 패턴으로 인출합니다. `withdraw()` 함수에서는 “Checks-Effects Interactions” 패턴을 쓴 것을 볼 수 있습니다. 장부의 값을 먼저 변경하고 외부 계정 `payee`에게 송금을 합니다.

송금 메소드 `payee.sendValue()`는 라이브러리 컨트랙트 `Address`에 있는데 다음과 같이 저수준 호출 `call()`을 사용하여 이더를 전송합니다.

```

function sendValue(address payable recipient, uint256 amount) internal {
    require(address(this).balance >= amount, "Address: insufficient balance");
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Address: unable to send value, recipient may have reverted");
}

```

4.5.2. Reentrancy Guard

재진입(Reentrancy) 문제에서 살펴본 것처럼 어떤 컨트랙트가 자신을 호출한 컨트랙트를 다시 호출하는 것은 보안적인 측면에서 위험할 수 있습니다. 오픈제ppelin 컨트랙트에는 재진입 문제를 막기 위한 `ReentrancyGuard`라는 컨트랙트가 있습니다.

```

// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (security/ReentrancyGuard.sol)

```

```

pragma solidity ^0.8.0;

abstract contract ReentrancyGuard {

    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;

       _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }
}

```

이 컨트랙트에는 `nonReentrant`라는 modifier가 정의되어 있는데 이것은 `_status` 값을 검사하여 modifier가 적용된 함수에 한 번 진입한 후에는 재진입이 되지 않도록 합니다.

2.15.4 재진입 문제의 인출 함수는 다음과 같이, “Checks-Effects Interactions” 패턴을 쓰지 않고 modifier를 추가하여 방지할 수 있습니다.

```

// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract Donation {
    ...

    function withdraw(uint256 _amount) public nonReentrant {
        if (balances[msg.sender] >= _amount) {
            (bool b0k, ) = msg.sender.call{value: _amount}("");
            if (!b0k) {
                revert();
            }
            unchecked{ balances[msg.sender] -= _amount; }
        }
    }
}

```

4.6. 정리

4장에서는 오픈제펠린이 제공하는 유용한 컨트랙트 몇 가지를 소개했습니다. 또 토큰 컨트랙트 중에 NFT에 관련된 ERC-721과 ERC-1155의 구현체들을 중심으로 살펴보았습니다. 6장에서 ERC-721 컨트랙트를 구현할 때 여기서 설명된 컨트랙트를 이용하게 될 것입니다.

Dapp 개발환경



이번 장에서는 스마트 컨트랙트 개발을 효율적으로 하기 위한 도구들을 살펴보겠습니다. 가장 많이 사용되는 트러플과 하드햇을 중심으로 설명합니다.

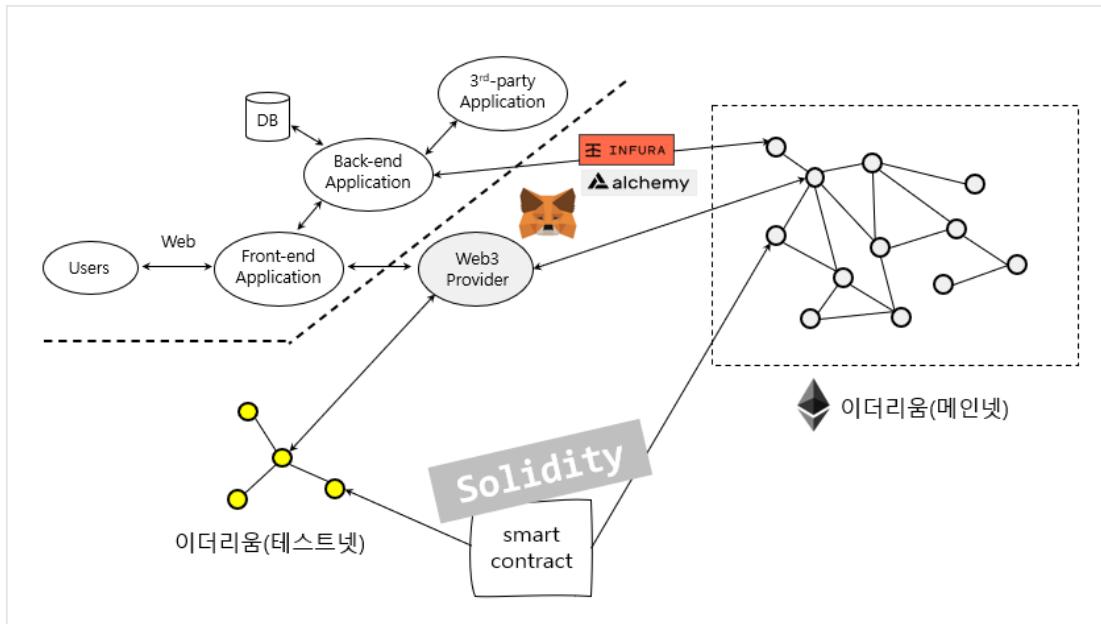
5.1. 탈중앙화 애플리케이션(Dapp)

우리가 사용하는 일반적인 애플리케이션과 인터넷 서비스들은 대부분 관리 주체가 분명합니다. 특정 기업들이 시스템을 만들고 유지하면서 사용자들에게 여러 가지 서비스를 제공합니다.

반면에 탈중앙화 애플리케이션(Decentralized Application, Dapp)은 관리 주체가 모호한 측면이 있습니다. 물론 현실적으로 모든 것을 탈중앙화할 수는 없기 때문에 사용자 경험 측면에서는 속도가 느리고 번거롭다는 것을 제외하면 비슷하게 느낄 수도 있겠습니다. 내부적으로는 스마트 컨트랙트를 통해 데이터를 처리한다는 점에서 차이가 있고 흔히 우리가 사용하는 애플리케이션과 크게 다르지 않습니다.

블록체인으로 전송되는 사용자 트랜잭션들은 어느 한 기업이 관리하는 데이터베이스에 저장되는 것이 아니라 수천대의 노드들에 의해, 소위 말하는 “분산원장”에 기록됩니다. 그래서 블록체인의 데이터는 누구의 것도 아니지만 동시에 누구나 볼 수 있고 가질 수 있습니다. 또 스마트 컨트랙트는 불변성을 가지고 있습니다. 왜냐하면 네트워크를 이루는 수 천대의 컴퓨터 안에 저장된 컨트랙트를 모두 지운다는 것은 사실상 불가능하기 때문입니다.

스마트 컨트랙트는 블록체인에서 실행되는 프로그램이기 때문에 실사용이 가능한 Dapp이 되려면 기존의 프론트엔드 또는 백엔드 시스템과 함께 서로 데이터를 주고받으며 상호작용해야 합니다.

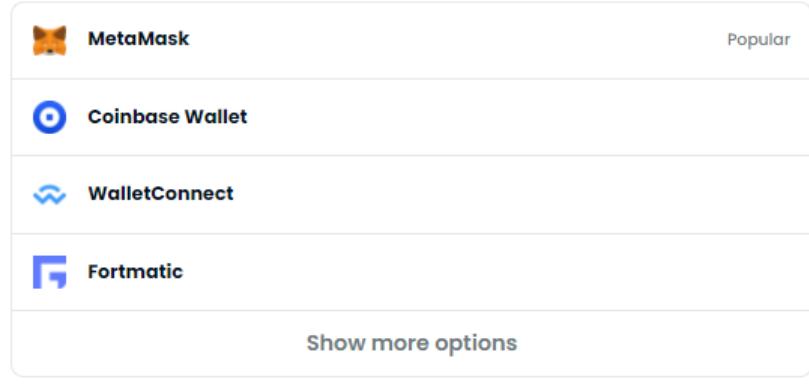


Dapp은 보통 메타마스크와 같은 지갑 소프트웨어가 필요합니다. 지갑 소프트웨어는 “지갑”이라는 단어 때문에 그 안에 토큰이 보관되어 있을 것이라고 착각할 수 있습니다. 하지만 지갑은 전자 서명을 위한 개인키를 생성하고 보관하는 소프트웨어입니다. 사용자가 보유한 토큰은 블록체인에 저장되는 것이지 지갑에 저장되는 것이 아닙니다.

탈중앙화 애플리케이션이기 때문에 자신의 개인키로 자신의 신분을 증명합니다. 예를 들어 NFT 마켓플레이스인 OpenSea는 별도의 회원가입 절차 없이 지갑만으로 즉시 이용이 가능합니다. 지갑으로 전자 서명된 트랜잭션들은 블록에 저장되어 블록체인에 기록됩니다.

Connect your wallet.

Connect with one of our available [wallet](#) providers or create a new one.



Dapp에서는 각 사용자들이 자신의 지갑을 직접 사용할 것은 권장하지만 블록체인 서비스를 제공하는 기업에서 지갑을 대신 관리하기도 합니다. 사용자 경험 측면에서는 이 구조가 편리하다고 생각할 수도 있겠습니다.

인퓨라(Infura)나 알케미(Alchemy)같은 API 서비스를 이용하여 애플리케이션이 이더리움과 데이터를 주고받는 방법도 있습니다. 이렇게 애플리케이션과 블록체인을 이어주는 지갑, API 서비스 등을 "Web3 Provider"라고 합니다.

Web3 Provider가 정해지면 Dapp에서는 이것과 통신하기 위한 라이브러리가 필요합니다. 가장 많이 쓰는 이더리움 자바스크립트 라이브러리에는 Web3.js와 Ethers.js가 있습니다. 결국 요약하자면 Dapp을 만든다는 것은 애플리케이션 구현 관점에서 볼 때 이더리움과 같은 블록체인에 연결시켜주는 Web3 Provider와 애플리케이션에서 이용 가능한 다양한 라이브러리를 결합하여 시스템을 개발하는 것으로 생각할 수 있겠습니다.

5.2. Dapp 개발의 요소

Dapp 개발은 우리가 흔히 알고 있는 시스템 개발과 유사합니다. 일반 사용자가 애플리케이션을 이용하기 위해서는, 대개 웹을 통해 접근하므로 웹 기반의 프론트엔드 애플리케이션이 필요합니다. 또 모든 데이터를 블록체인에 저장할 수 없기 때문에 백엔드 애플리케이션 서버와 데이터베이스도 있어야 합니다.

이번 5장에서는 스마트 컨트랙트 개발을 중심으로 알아보겠습니다. 여기서는 프론트엔드나 백엔드 애플리케이션을 구현하는 기술을 깊게 다루지는 않습니다.

5.2.1. 솔리디티

우리는 이미 2장에서 솔리디티에 대해 알아보았습니다. 스마트 컨트랙트는 백엔드에 속한다고 볼 수 있습니다. 블록체인을 분산 데이터베이스라고 생각한다면 데이터베이스에서 실행되는 저장 프로시저와 비교할 수도 있겠습니다.

이더리움은 솔리디티라는 스마트 컨트랙트 전용 언어로 컨트랙트를 작성합니다. 이더리움이 프로그래밍이 가능한 블록체인 플랫폼으로 가장 먼저 출발했기 때문에 그동안 축적된 개발자원이 풍부하다는 장점을 가지고 있습니다. 최근 유행하는 디파이(De-fi)와 NFT 분야에서도 솔리디티 기반의 스마트 컨트랙트가 많은 부분을 차지하고 있습니다.

Dapp을 개발하려면 스마트 컨트랙트 작성을 위한 프로그래밍 언어와 컴파일러, 테스트, 디버깅을 위

한 개발도구가 필요할 것입니다. 이제부터 어떤 개발도구들이 있는지 살펴보겠습니다.

5.2.2. 자바스크립트 기반 개발도구

2020년에 실시된 이더리움 개발자 설문(<https://trentv.medium.com/ethglobal-developer-survey-report-2020-f7bf4f7cf821>)에 의하면 개발자들이 가장 많이 사용하는 언어는 스마트 컨트랙에서는 솔리디티이고 애플리케이션에서는 자바스크립트로 나타났습니다.

솔리디티 스마트 컨트랙트 개발도구는 여러 가지가 존재하지만 가장 많이 사용되는 것은 트러플(Truffle)과 하드햇(Hardhat)이라는 도구입니다. 아래는 공식 홈페이지입니다.

- 트러플 <https://www.trufflesuite.com/>
- 하드햇 <https://hardhat.org/>

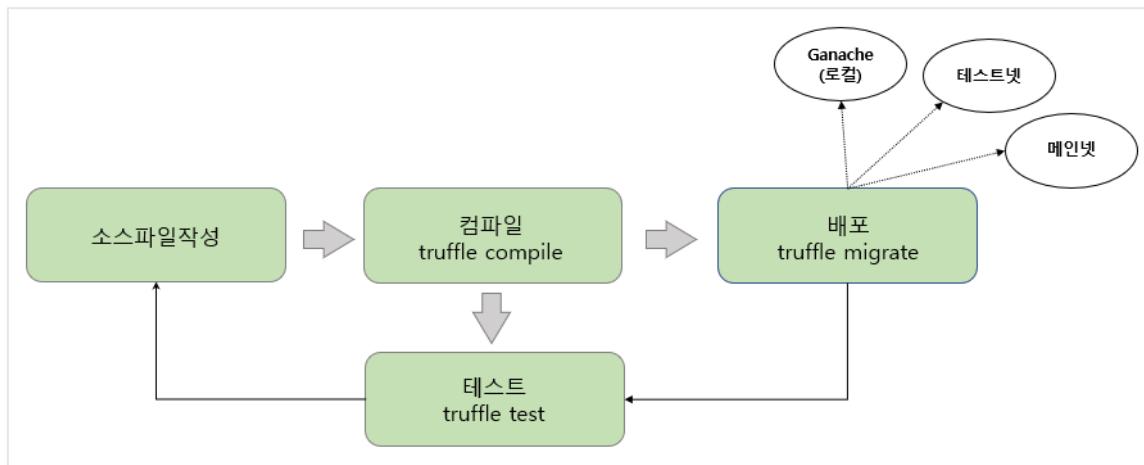
트러플과 하드햇 모두 자바스크립트 환경으로 기존의 개발자들이 쉽게 접근할 수 있습니다. 또 자바스크립트와 잘 결합될 수 있기 때문에 자연스럽게 프론트엔드와 백엔드 애플리케이션도 자바스크립트 기반의 기술 스택으로 이어지게 됩니다.

초기에는 트러플을 사용하는 개발자들이 압도적으로 많았지만 지금은 하드햇이 좋은 반응을 얻고 있어서 점차 확산되는 추세입니다.

5.3. 트러플(Truffle)

트러플은 가장 먼저 나온 이더리움 스마트 컨트랙트 개발도구입니다. 주로 스마트 컨트랙트의 컴파일, 테스트, 디버깅을 위한 명령행(CLI) 기반의 자동화 도구라고 할 수 있습니다.

트러플의 개발 프로세스를 간단하게 나타내면 아래 그림과 같습니다.



컨트랙트 소스파일은 개발자가 원하는 소스 편집기를 사용하여 작성하면 됩니다. 대부분의 소스 편집 도구들이(비주얼 스튜디오 코드나 인텔리제이 같은) 터미널을 지원하므로 소스를 작성한 후에 도구 내에서 트러플 명령어로 컴파일과 테스트, 배포를 수행할 수 있습니다.

컴파일된 솔리디티 스마트 컨트랙트는 바이트코드로 변환되어 이더리움에 전송됩니다. 트러플 명령어를 사용하면 간단히 배포할 수 있습니다. 그런데 개발 중인 컨트랙트를 이더리움 메인넷에 직접 배포하는 것은 속도와 비용 측면에서 매우 비효율적입니다. 왜냐하면 이더리움에는 가스라는 개념이 있기 때문에 컨트랙트를 배포하고 실행하기 위해서는 수수료를 지불해야 하기 때문입니다.

그래서 이더리움 실행환경을 로컬 환경에서 애뮬레이션해주는 가상 이더리움 프로그램이 있습니다. 가장 많이 사용되는 것이 가나슈(Ganache)입니다. 가나슈는 CLI 버전 ganache-cli 와 UI 버전 Ganache UI 가 있습니다. 가나슈는 트러플 개발도구 모음에 속해 있기 때문에 트러플 홈페이지에서 다운로드 받을 수 있습니다.

- 가나슈(UI) <https://github.com/trufflesuite/ganache-ui/releases>
- 가나슈(CLI) <https://github.com/trufflesuite/ganache/releases>

가나슈 CLI와 GUI 버전은 포트와 네트워크 번호가 다르므로 연결할 때 확인해야 합니다.

가나슈	포트	네트워크 번호(체인 번호)
CLI	8545	임의(1337)
UI	7545	5777(1337)

아래 그림은 가나슈 UI의 실행화면입니다. 가나슈는 개발 중에 사용할 수 있는 테스트 계정을 제공합니다. 당연한 말이지만 각 계정의 잔액 100 이더는 진짜 이더가 아닙니다.

The screenshot shows the Ganache UI interface. At the top, there are tabs for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS. Below the tabs, there are several status indicators: CURRENT BLOCK (0), GAS PRICE (20000000000), GAS LIMIT (6721975), HARDFORK (MUIRGLEACIER), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), and MINING STATUS (AUTOMINING). On the right side, there are buttons for WORKSPACE (ACTIVE), QUICKSTART, SAVE, SWITCH, and GEAR. A search bar at the top right allows searching for block numbers or tx hashes. The main area displays a table of accounts:

MNEMONIC	HD PATH			
spice document basic express evil sentence author frost only neutral material social	m/44'/60'/0'/0/account_index			
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 0	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 1	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 2	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 3	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 4	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 5	🔑
ADDRESS 0x... (multiple rows)	BALANCE 100.00 ETH	TX COUNT 0	INDEX 6	🔑

가나슈 UI는 “QUICKSTART” 모드와 “WORKSPACE” 모드가 있는데 전자는 가나슈를 재시작하면 이전에 배포된 컨트랙트와 트랜잭션들이 초기화되어 매번 새로 배포할 필요가 있습니다. 후자는 실행한 데이터들이 계속 유지됩니다.

트러플은 내부적으로 가나슈 CLI를 제공하지만 별도로 ganache-cli를 설치해도 됩니다. ganache-cli는 npm으로 설치합니다.

```
> npm install -g ganache-cli
> ganache-cli --version
Ganache CLI v6.12.2 (ganache-core: 2.13.2)
```

ganache-cli를 실행하면 역시 테스트 계정을 생성하면서 <http://127.0.0.1:8585>로 접속할 수 있습니다.

```
> ganache-cli
Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
=====
(0) 0x16047eACC98f4cb5c3f125aAF4233BA339bcF040 (100 ETH)
(1) 0x1a34e40FE07F0486F6bdd7B07baf1CC73842f49e (100 ETH)
(2) 0x0893f6A4073464AaA499e5440989F454717E3FFF (100 ETH)
(3) 0xe27666D809F9aBF812D172C52B6582587038edA4 (100 ETH)
(4) 0x60631e9701BE3b200d3E6ac1188897167cab5295 (100 ETH)
(5) 0xa5C427ACA770f148858A566d6Be5B4e3bb0Dcd1a (100 ETH)
(6) 0x51361AB558A5e4EC3d38Cb994fc588dc24F4A918 (100 ETH)
(7) 0x0BBb74fb56e4c2825ED3Dc0bd4e5De752A215469 (100 ETH)
(8) 0x953535bdb40fD4900bf92A1A2CE89aF36BbB49BC (100 ETH)
(9) 0x4d535493f3B6ba00ce2FffAAf311C77E29bb6879 (100 ETH)

Private Keys
=====
(0) 0xe31fff43c86e2d61558f3b671db5e832f97dad70db8d699adc87dd4f810372b8
(1) 0x350e15a33c9a4eda31c13c5bf4e9eda5d380f742b92ef6fcfd98ccac3debb235c
(2) 0xb5342306b8de45fc554bf50fe1149b621c796fdf513f6a4e78f3f279968a86af
(3) 0xd2799c482a284166c2712af3b4b7fb413ae013c04d9826f977fbbd41b5c14c39
(4) 0x58263d911b24e2a4079daa57ab90dc4d919737e5c696ab46ff38922d040bb913
(5) 0x92c592f9ea8266260573ea2b5d9ac155a36d6e4081811c77aad4a384af20bd8d
(6) 0xd6aa4663bac60766317a9a093205cf587af24584facffc4830e71818d83ba0c2
(7) 0xd3998202107f5e3a2bd18bbd94f0fb1b6999775728c82665c326730e78b78107
(8) 0xc50a4ba80a5abf85a695ba82e561b2241a7c962d645b0bd38a3d80713f42cb3
(9) 0x3571c75345c8b693d2a225b86df9bbba6b5650ac91752ed9eaeaf9d649e99f61

HD Wallet
=====
Mnemonic:      urge warm local kingdom broccoli excess strike chunk armor basic wide trend
Base HD Path: m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 127.0.0.1:8545
```

ganache-cli는 Ganache UI에 비해 다양한 실행 옵션을 제공합니다. 옵션을 보려면 다음과 같이 입력하면 됩니다.

```
> ganache-cli --help
```

가나슈는 어디까지나 로컬 환경에서 이더리움을 애뮬레이션하는 것이기 때문에 실제 메인넷과 유사한 환경에서 테스트를 할 필요가 있습니다. 이 때는 이더리움의 테스트넷을 이용합니다.

테스트넷에 컨트랙트를 배포하려면 이더리움의 공용 토큰인 이더가 필요하지만 테스트넷의 이더는 가치가 없으므로 무료로 받을 수 있습니다(인터넷에서 "ethereum faucet"으로 검색). 이더리움의 주요 테스트넷은 다음과 같습니다.

테스트넷	체인 번호
Ropsten	3
Rinkeby	4
Goerli	5
Kovan	42

5.3.1. 트러플 설치 (Window 10/11, Powershell 기준)

트러플은 자바스크립트 기반의 개발 도구입니다. 따라서 Node.js를 미리 설치해야 합니다. 이 문서가 작성되는 시점에 Node.js의 LTS 버전은 16.13.0 입니다. Node.js 는 아래 사이트에서 다운로드하기 바랍니다.

<https://nodejs.org/ko/>

Node.js가 설치된 후에 다음 명령어로 정상적으로 설치되었는지 확인합니다.

```
> node -v  
v16.13.0
```

앞으로 설치할 자바스크립트의 패키지(트러플도 자바스크립트 패키지입니다)를 위해 패키지 매니저인 npm을 설치합니다.

```
> npm install -g npm  
> npm -v  
8.1.3
```

트러플이 이미 설치되어 있는 경우 새로운 버전의 트러플을 설치하려면 이전 트러플을 삭제한 후에 새로운 버전을 설치하는 것이 좋습니다. 트러플은 다음과 같이 삭제합니다.

```
> npm uninstall -g truffle
```

이제 트러플을 설치할 차례입니다. 트러플은 다음과 같이 설치합니다. 설치 후에 truffle version으로 확인합니다.

```
> npm install -g truffle  
  
> truffle version  
Truffle v5.4.19 (core: 5.4.19)  
Solidity v0.5.16 (solc-javascript)  
Node v16.13.0  
Web3.js v1.5.3
```

트러플은 Web3.js라는 이더리움 자바스크립트 라이브러리를 사용합니다. 환경 설정에서 솔리디티 컴파일러 버전을 명시하지 않은 경우는 위에 표시된 0.5.16을 사용하여 컴파일하게 됩니다.

5.3.2. 기본 구조

트러플은 컨트랙트 컴파일, 테스트, 배포 관리를 위한 자동화 도구입니다. GUI 기반의 개발환경에 익숙한 사용자들에게는 다소 번거롭고 불편하게 느껴질 수 있지만 익숙해지면 또 그것만큼 편한 것 없을 것입니다.

트러플 프로젝트를 시작하려면 트러플에서 스캐폴드한 프로젝트 구조를 따르는 것이 좋습니다. 우선 프로젝트 디렉토리를 하나 생성한 후 생성된 디렉토리로 이동하여 프로젝트를 초기화합니다.

```
> md truffle-dapp  
> cd truffle-dapp  
> truffle init  
  
Starting init...  
=====>  
  
> Copying project files to C:\Users\song\bookapp\truffle-dapp  
  
Init successful, sweet!  
  
Try our scaffold commands to get started:  
$ truffle create contract YourContractName # scaffold a contract  
$ truffle create test YourTestName # scaffold a test
```

<http://trufflesuite.com/docs>

트러플의 init 명령어는 프로젝트 디렉토리 구조를 잡아주고 필요한 설정파일을 생성합니다. 생성된 프로젝트 구조는 아래와 같습니다.

Mode	LastWriteTime	Length	Name
-	-	-	-
d-----	2021-11-16 오전 11:24		contracts
d-----	2021-11-16 오전 11:24		migrations
d-----	2021-11-16 오전 11:24		test
-a----	2021-11-16 오전 11:10	4901	truffle-config.js

디렉토리 구조를 준수해야 트러플이 제공하는 자동화 프로세스를 그대로 사용할 수 있습니다. contracts 디렉토리에서 컨트랙트를 작성해야 하고, migrations 디렉토리에서는 배포 스크립트를 관리합니다. test 디렉토리 아래에는 단위 테스트 스크립트가 있어야 합니다(물론 설정을 통해 원하는 디렉토리로 변경할 수는 있습니다). truffle-config.js 는 환경 설정 파일입니다. 컴파일러 버전을 지정하거나 컴파일된 컨트랙트를 로컬이나 테스트넷 또는 메인넷에 배포할 때 연결정보를 작성합니다.

5.3.3. 기본 설정

그럼 간단한 컨트랙트를 작성하고 배포하면서 기본 사용법을 알아보도록 하겠습니다. 우선 각자 선호하는 소스 편집기에서 프로젝트 디렉토리를 선택합니다. 비주얼 스튜디오 코드 같은 도구들은 솔리디티 플러그인이 존재하므로 설치해서 사용하는 것도 좋습니다.

truffle-config.js을 열어서 기본 설정을 살펴보겠습니다. 대부분은 주석처리가 되어 있습니다. 우선 networks 부분을 보겠습니다.

```
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache-cli, geth or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //
  development: {
    host: "127.0.0.1",
    port: 7545,
    network_id: "5777",
  },
}
```

`networks` 설정은 컴파일된 컨트랙트를 어디에 배포할지 설정합니다. 가나슈 같은 로컬 가상 이더리움, 테스트넷 또는 진짜 메인넷 등을 설정할 수 있습니다. 나중에 배포 명령어를 실행할 때 설정된 네트워크 중 하나를 지정하여 배포하게 됩니다.

`development` 부분의 주석을 해제합니다. 가나슈 GUI를 사용할 것이므로 `port`와 `network_id` 를 각각 7545와 5777로 변경합니다.

네트워크의 이름은 임의로 정할 수 있지만 `development`라는 이름은 바꾸지 않는 것이 좋습니다. 트러플은 네트워크를 지정하지 않는 경우 `development`라는 이름의 네트워크에 배포합니다.

설정 파일의 하단에는 컴파일러 버전을 지정하는 곳이 있습니다.

```
compilers: {
  solc: {
    version: "0.8.10",
    // docker: true,
    // settings: {
    //   optimizer: {
    //     enabled: false,
    //     runs: 200
    //   },
    //   evmVersion: "byzantium"
    // }
  }
},
```

`version`에 지정되는 컴파일러 버전은 범위 지정이 가능합니다. 즉 ^0.8.0처럼 버전 범위 표기법 (semver)을 사용할 수 있습니다. 하지만 이 경우에는 컨트랙트 소스 파일의 `pragma`에서도 범위 지정이 되어야 합니다. 위와 같이 단일 버전을 지정하는 경우는 `pragma`에 지정된 버전과 일치하거나 범위 내에 있어야 합니다.

`optimizer`는 컴파일된 바이트 코드의 크기를 최적화하는 옵션입니다. 메인넷에 배포할 때 수수료를 절약하는데 도움이 될 수 있습니다.

환경 설정에서 `version`을 명시하는 경우는 트러플 설치 후에 표시되었던 디폴트 컴파일러 대신 여기에 지정된 컴파일러를 다운로드 받아서 컴파일하게 됩니다.

5.3.4. 컴파일

이제 컨트랙트를 작성해보겠습니다. SimpleStorage 컨트랙트는 상태변수 하나에 값을 쓰고 읽는 간단한 컨트랙트입니다. set이라는 함수에 전달되는 값이 조건에 부합하지 않으면 예외를 발생시킵니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

contract SimpleStorage {

    uint256 storedData;

    event Change(string message, uint newVal);

    constructor (uint s) {
        storedData = s;
    }

    function set(uint x) public {
        require(x < 5000, "Should be less than 5000");
        storedData = x;
        emit Change("set", x);
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

컨트랙트는 contracts 디렉토리 아래에 있어야 합니다. 다음 명령어로 컴파일을 수행합니다.

```
> truffle compile

Compiling your contracts...
=====
✓ Fetching solc version list from solc-bin. Attempt #1
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\SimpleStorage.sol
> Artifacts written to C:\Users\song\bookapp\truffle-dapp\build\contracts
> Compiled successfully using:
  - solc: 0.8.10+commit.fc410830.Emscripten clang
```

truffle-config.js에 설정한 0.8.10 컴파일러가 사용되었음을 알 수 있습니다. 컴파일 결과물(artifact)은 build 디렉토리에 SimpleStorage.json 파일로 저장됩니다(설정에서 생성 위치를 변경할 수 있습니다). 컴파일 결과물은 소스파일 별로 생기는 것이 아니라 컨트랙트 별로 생성됩니다. 한 소스 파일에 컨트랙트가 여러 개 작성되어 있다면 각각 JSON 파일이 만들어집니다.

컴파일 후에 생성되는 파일에는 나중에 애플리케이션에서 사용하는 ABI 정보와 배포용 바이트 코드,

배포 네트워크와 주소 등의 정보가 기록됩니다. 사람이 보기 위한 것이 아니므로 건드리지 않는 것이 좋습니다.

5.3.5. 배포(migrate)

트러플에서는 컨트랙트를 배포할 때 변경하지 않은 컨트랙트가 의도치 않게 재배포되어 주소가 바뀌는 것을 방지하기 위해 Migrations라는 컨트랙트를 사용합니다. init 명령어로 프로젝트를 만들면 contracts/Migrations.sol 과 migrations/1_initial_migration.js 라는 파일이 생성됩니다.

마지막에 실행된 배포 스크립트의 번호를 Migrations 컨트랙트에 기록하고 다시 배포 명령이 실행될 때는 그 다음 순번의 스크립트부터 실행하게 됩니다. 그래서 배포 스크립트를 작성할 때는 규칙에 따라 파일명에 순번을 붙여야 합니다.

물론 개발 중에는 가나슈나 테스트넷을 사용하므로 계속 새로운 컨트랙트를 배포하는 것이 부담스럽지 않을 것입니다. 또 반드시 Migrations 컨트랙트를 사용해야 하는 것은 아니라서 지워도 상관없습니다.

배포를 위해서는 migrations 디렉토리에 배포 스크립트를 작성합니다. 이미 트러플이 사용하는 배포 스크립트 1_initial_migration.js이 존재하므로 파일명은 2_로 시작해야 합니다. 아래와 같이 스크립트를 작성합니다. 파일명은 2_deploySimpleStorage.js으로 합니다.

```
const SimpleStorage = artifacts.require("SimpleStorage");

module.exports = function (deployer) {
  deployer.deploy(SimpleStorage, 100);
};
```

artifacts.require("SimpleStorage")에서 "SimpleStorage"는 파일명이 아니라 컨트랙트 이름을 써야 합니다. 배포는 deployer를 받는 콜백 함수에서 수행됩니다. 컴파일된 컨트랙트의 결과물을 가져오고 생성자의 초기 값을 넣어 deploy를 호출합니다. 배포 스크립트 안에서 네트워크를 지정하는 것이 아니기 때문에 하나의 스크립트를 여러 네트워크 배포에 사용할 수 있습니다.

배포 전에 가나슈를 실행시켜야 합니다. 배포 명령어는 migrate입니다.

```
> truffle migrate
Compiling your contracts...
=====
```

```

> Compiling ./contracts\Migrations.sol
> Compiling ./contracts\SimpleStorage.sol
> Artifacts written to C:\Users\song\bookapp\truffle-dapp\build\contracts
> Compiled successfully using:
- solc: 0.8.10+commit.fc410830.Emscripten.clang

Starting migrations...
=====
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)

1_initial_migration.js
=====

Replacing 'Migrations'
-----
> transaction hash: 0x2c61f2f2e47fd302d8be7a1bfee3660cd06ce51c85ec4bd2973eafce2e1cde30
> Blocks: 0          Seconds: 0
> contract address: 0xbb88F7A047b1f024f7EDbdcC88dEEB166850298f
> block number:      9
> block timestamp:   1637041341
> account:           0xDc64b681687a471245eFE89CfCf2AaDcBcdd9dE3
> balance:            99.9731668
> gas used:          248854 (0x3cc16)
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.00497708 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:        0.00497708 ETH

2_deploySimpleStorage.js
=====
development

Replacing 'SimpleStorage'
-----
> transaction hash: 0x9f2fa1872d6f5273ed468e2e585509b0a97a1a0c31f33f7e3752a83c70e436b7
> Blocks: 0          Seconds: 0
> contract address: 0xa6eD44D30135d23BF308a77A3Cfb2e4809a805c
> block number:      11
> block timestamp:   1637041342
> account:           0xDc64b681687a471245eFE89CfCf2AaDcBcdd9dE3
> balance:            99.96776608
> gas used:          227523 (0x378c3)
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.00455046 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:        0.00455046 ETH

```

```

Summary
=====
> Total deployments: 2
> Final cost: 0.00952754 ETH

```

배포되면서 소모된 가스와 수수료, 그리고 컨트랙트 주소가 표시됩니다. 가나슈는 디폴트로 채굴 없이 트랜잭션을 즉시 처리하므로(블록 생성 시간을 설정하는 것도 가능) 배포가 금방 끝나지만 메인넷이나 테스트넷에서는 시간이 소요된다는 것을 기억해야 합니다. 배포하는 계정은 가나슈의 첫 번째 테스트 계정입니다. 만약 두 번째 계정으로 배포하고 싶다면 다음과 같이 수정합니다.

```

const SimpleStorage = artifacts.require("SimpleStorage");

module.exports = function (deployer, network, accounts) {
  deployer.deploy(SimpleStorage, 100, {from: accounts[1]});
};

```

컨트랙트가 배포되면 가나슈의 트랜잭션 메뉴에서 컨트랙트 생성 트랜잭션(CONTRACT CREATION)을 확인할 수 있습니다. 맨 위에는 배포 스크립트의 순번을 조회하는 호출입니다.

CONTRACT CALL			
TX HASH	FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED
0x149eea38c29efa9361310ee900e4e872ac4a458b57cf0df362a71020c1261139	0xDc64b681687a471245eFE89CFCf2AaDcBcdd9dE3	0x227b5b885DDC68Fa7e74372C4A2cCeCdd83cB353E	27513
CONTRACT CREATION			
TX HASH	FROM ADDRESS	CREATED CONTRACT ADDRESS	VALUE
0xe0d6842345b62fbbad6cc41217d0367fa03b0f5f134d546ea32bd53e2b269a8d	0x77E31f687b873887275B3A99088462EE67784FAD	0xC1CF4b87b84a60B88e70056af70D7e41a50aDA4F	227523

migrate 명령은 동일한 배포 스크립트를 사용하여 --network 옵션에 지정된 네트워크에 배포합니다. 생략하는 경우 development로 지정된 네트워크에 배포됩니다. 현재 development는 가나슈로 되어 있습니다.

만약 테스트넷 Rinkeby를 truffle-config.js의 networks 항목에 "rinkeby"라는 이름으로 설정했다면 배포 명령어는 다음과 같이 되겠습니다.

```
> truffle migrate --network rinkeby
```

순서에 상관없이 모든 컨트랙트를 새로 배포하고 싶다면 --reset 옵션을 사용합니다.

```
> truffle migrate --reset
```

배포 스크립트를 실행할 때 특정 번호의 스크립트만 실행하려면 범위를 지정할 수 있습니다. 예를 들어 3_deploy.js와 4_deploy.js만을 실행시키려면 -f 와 --to 옵션을 사용합니다.

```
> truffle migrate -f 3 --to 4
```

각 네트워크에 배포된 컨트랙트의 주소는 다음과 같이 볼 수 있습니다. 배포 주소는 컴파일 결과인 SimpleStorage.json에 기록이 됩니다.

```
> truffle networks

Network: development (id: 5777)
Migrations: 0x227b5b885DDC68fAe74372C4A2cCeCdd83cB353E
SimpleStorage: 0xC1CF4b87b84a60B88e70056af70D7e41a50aDA4F
```

다수의 컨트랙트가 복잡하게 배포되는 경우는 트러플 홈페이지에 다양한 예시가 있으므로 도움말을 참고하시기 바랍니다.

<https://www.trufflesuite.com/docs/truffle/getting-started/running-migrations>

5.3.6. 단위 테스트

스마트 컨트랙트는 화면이 제공되지 않기 때문에 제대로 동작하는지를 확인하기 위해서는 테스트 스크립트를 작성해서 단위 테스트를 수행해야 합니다. 테스트 스크립트는 test 디렉토리 아래에 작성합니다.

테스트 스크립트는 솔리디티와 자바스크립트로 작성할 수 있는데 여기서는 자바스크립트를 사용하여 테스트 스크립트를 작성하기로 하겠습니다.

트러플 테스트 스크립트는 잘 알려진 자바스크립트 테스트 프레임워크인 모카(Mocha)와 차이(Chai)를 사용하는데, 차이점은 describe 대신 contract를 쓴다는 것입니다(describe도 가능).

```
const SimpleStorage = artifacts.require("SimpleStorage");

contract ("SimpleStorage Test", function () {
    it ("Test case 1", async () => {
        //Test code...
    });
})
```

하나의 테스트 스크립트는 여러 개의 테스트 케이스 it(...) 블록으로 구성됩니다. 테스트 스크립트 파일을 여러 개 작성하는 경우 각 파일에서 실행되는 결과는 서로 영향을 주지 않습니다. 또 스크립트에서 변경된 상태 변수들은 실제 컨트랙트에는 반영되지 않는 “클린룸” 환경에서 수행됩니다.

test 디렉토리 아래에 다음과 같은 test.js 파일을 작성합니다.

```
const SimpleStorage = artifacts.require("SimpleStorage");

contract ("SimpleStorage Test", function () {
    before(async () => {
        this.instance = await SimpleStorage.deployed();
    });

    it ("Should change the value", async () => {
        await this.instance.set(500);
        const val = await this.instance.get.call();
        assert.equal(500, val, "The result is incorrect!");
    });

    it ("Should emit the event", async () => {
        const expectedEvent = ["Change", "set", "2000"];
    });
})
```

```

    const result = await this.instance.set(2000);
    const logs = result.receipt.logs[0];
    const receivedEvent = [logs.event, logs.args.message, logs.args.newVal.toString()];

    assert.isOk(JSON.stringify(expectedEvent) === JSON.stringify(receivedEvent));

});

it ("Should be failed if the value greater than 5K", async () => {
    let err = null;
    try {
        await this.instance.set(10000);

    } catch ( error ) {
        err = error;

        if (error.message !== null && error.message.length > 0) {
            var reason = error.message.match(/Reason given: (.*)\./);
        }
    }

    assert.isOk(err instanceof Error, "The value exceeds the limit");
    assert.equal(reason[1], "Should be less than 5000");
});

})

```

단위 테스트는 프로그램을 실행했을 때 올바른 결과가 나오는지를 확인하는 것입니다. 먼저 컨트랙트의 인스턴스를 생성합니다. 트러플의 테스트 스크립트에서는 `@truffle/contract`라는 라이브러리 (<https://github.com/trufflesuite/truffle/tree/master/packages/contract#readme>)를 그대로 사용할 수 있습니다. `deployed()`는 배포 컨트랙트의 인스턴스를 생성하는 함수입니다.

`test.js`에는 `it(...)`로 묶인 3개의 테스트 케이스가 있습니다. 첫 번째 케이스는 `set`이라는 함수를 테스트하는 것입니다. `set`을 호출하면서 전달한 값이 `storedData`를 제대로 변경하는지 확인합니다.

자바스크립트에서 컨트랙트의 함수를 호출할 때는 항상 `async-await`를 사용합니다. `set(500)`을 호출한 후에 다시 `get.call()` 메소드로 조회를 하면 당연히 500을 리턴 해야 합니다. 이것을 `assert`라는 단위 테스트 함수로 확인합니다.

두 번째는 이벤트 테스트입니다. `set` 호출 후에 리턴 받은 트랜잭션 `Receipt` 정보로부터 이벤트 로그를 추출하여 이벤트에 전달된 값을 검사하면 되겠습니다. `Change`라는 이벤트는 두 개의 값을 로그에 기록합니다. 이것을 배열에 넣어서 비교할 수 있습니다.

세 번째는 `revert` 발생을 테스트합니다. 이 경우는 `revert`가 되는 것이 맞는 것입니다. 일부러 `require(x<5000)`의 조건을 `false`로 하는 큰 값을 입력하고 `try-catch` 예외처리에서 `revert reason`을 추출하고 그 값이 맞는지 확인합니다.

테스트 케이스 확인 로직은 작성자의 재량에 달려있습니다. 고정된 형식이 있는 것이 아니라 다양한 방법으로 테스트 스크립트를 작성할 수 있습니다. 단위 테스트 명령어는 test입니다.

```
> truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\SimpleStorage.sol
> Artifacts written to C:\Users\song\AppData\Local\Temp\test--3080-e6nJTN0vMXHy
> Compiled successfully using:
- solc: 0.8.10+commit.fc410830.Emscripten clang

SimpleStorage Test
✓ Should change the value (704ms)
✓ Should emit the event (584ms)
✓ Should be failed if the value greater than 5K (1855ms)

3 passing (3s)
```

5.3.7. 디버깅

트러플은 디버깅 기능을 제공합니다. 명령행 기반이라서 다소 불편한 느낌은 들지만 다양한 방법으로 디버깅이 가능합니다. 디버깅은 보통 트랜잭션이 실패한 경우 원인을 찾을 때 사용합니다. 예를 들어 다음과 같은 경우를 가정해보겠습니다.

```
truffle(development)> await ss.set(10000)

Uncaught Error: Returned error: VM Exception while processing transaction: revert
...
data: {
  '0xda15bca9dc896463e0f1bb62c62cd9743b1f810dd7156f21d6dc9bb46d34c094': { error: 'revert',
  program_counter: 131, return: '0x' },
  stack: 'RuntimeError: VM Exception while processing transaction: revert\n' +
...
...
```

위의 상황은 set을 호출하면서 일부러 5000 보다 큰 값을 전달했을 때 revert가 발생하는 상황을 재연한 것입니다. 컨트랙트에서 revert reason을 지원하기 때문에 어떤 이유인지 알 수 없습니다.

트러플은 트랜잭션 해시를 사용해서 디버깅을 수행합니다. 여기서 트랜잭션 해시는 오류가 나는 화면으로부터 확보합니다.

```
'0xda15bca9dc896463e0f1bb62c62cd9743b1f810dd7156f21d6dc9bb46d34c094'
```

디버깅 명령어는 debug <트랜잭션 해시>입니다.

```
> truffle debug 0xda15bca9dc896463e0f1bb62c62cd9743b1f810dd7156f21d6dc9bb46d34c094
Starting Truffle Debugger...
✓ Gathering information about your project and the transaction...

Addresses affected:
0x3493cCd55DD059d2ce637EB9077068dda877130a - SimpleStorage

Commands:
(enter) last command entered (step next)
(o) step over, (i) step into, (u) step out, (n) step next
(c) continue until breakpoint, (Y) reset & continue to previous error
(y) (if at end) reset & continue to final error
(;) step instruction (include number to step multiple)
(g) turn on generated sources, (G) turn off generated sources except via `;` 
(p) print instruction & state (`p [mem|cal|sto]*`; see docs for more)
(l) print additional source context, (s) print stacktrace, (h) print this help
(q) quit, (r) reset, (t) load new transaction, (T) unload transaction
(b) add breakpoint (`b [<source-file>:]<line-number>`; see docs for more)
(B) remove breakpoint (similar to adding, or `B all` to remove all)
(+) add watch expression `+<expr>`, (-) remove watch expression `-<expr>`
(?) list existing watch expressions and breakpoints
(v) print variables and values (`v [bui|glo|con|loc]*`; see docs for more)
(:) evaluate expression - see `v`
```

SimpleStorage.sol:

```
5:
6: contract SimpleStorage {
    ^^^^^^^^^^^^^^^^^^^^^^

debug(development:0xda15bca9...)>
```

디버깅이 시작되면 디버그 콘솔로 전환됩니다. Commands에 나온 단축키를 눌러서 단계별로 코드 실행을 따라가거나 중간 값을 확인할 수 있습니다. 엔터를 치면 (n) step next 와 동일합니다. 디버그 콘솔에서 n을 입력합니다.

```
debug(development:0xda15bca9...)> n
SimpleStorage.sol:

14:     }
15:
16:     function set(uint x) public {
    ^^^^^^^^^^^^^^^^^^
```

set 함수를 호출한 트랜잭션이므로 set 함수로 이동했습니다. 다시 n을 입력합니다.

```
debug(development:0xda15bca9...) > n
SimpleStorage.sol:
16:     function set(uint x) public {
17:         require(x < 5000);
18:         ^
        ^
```

다음 실행 코드에서 require를 만났습니다. 그렇다면 여기서 현재 x의 값을 보고 싶습니다. 이때는 v를 누릅니다.

맨아래에 있는 로컬 변수 `x` 값을 볼 수 있습니다. 10000이라는 값이 전달되었음을 알 수 있습니다. 너무 많은 정보가 나온 경우는 `v -buf`를 입력하면 좀더 간결하게 볼 수 있습니다.

```
debug(development:0xda15bca9...)> v -bu  
  
Contract variables:  
  storedData: 0
```

```
Local variables:  
x: 10000
```

좀 더 진행하기 위해 계속 n을 눌러보겠습니다. 드디어 의심스러운 require의 조건에 도착했습니다.

```
debug(development:0xda15bca9...)> n  
  
SimpleStorage.sol:  
  
16:     function set(uint x) public {  
17:         require(x < 5000);  
18:             ^^^^^^^^^
```

조건에 있는 표현식의 값을 다음과 같이 확인해볼 수 있습니다. require의 조건식이 false가 된다는 것을 알 수 있으므로 revert의 원인을 찾게 됩니다.

```
debug(development:0xda15bca9...)> :x < 5000  
false
```

트랜잭션 해시를 사용하는 디버깅 외에도 테스트 스크립트에서 debug() 함수를 써서 중단점(break point)을 설정할 수 있는 방법도 제공합니다. 예를 들어 set 함수를 실행할 때 debug 함수의 인자로 함수 호출을 넣으면 스크립트가 실행될 때 테스트가 중지되면서 디버그 콘솔로 전환됩니다.

```
it ("Should change the value", async () => {  
    await debug(this.instance.set(500));  
    const val = await this.instance.get();  
    assert.equal(500, val, "The result is incorrect!");  
});
```

5.4. 하드햇(Hardhat)

노믹 랩스(Nomic Labs.)의 하드햇은 이전에 “비들러 Buidler”라는 이름으로 알려진 도구였는데, 2020년말 즈음에 리브랜딩을 하고 지금은 자타공인 인기있는 이더리움 Dapp 개발도구로 자리매김하고 있습니다. 다수의 유명한 개발팀들이 하드햇을 사용하고 있습니다.

하드햇은 트러플에 비해 경량의 개발도구입니다. 여러 기능들이 플러그인 형태로 구성되었기 때문에 선택적으로 사용가능하며 유연하고 확장성이 있습니다. 그래서 숙련된(?) 개발자들이 선호하는 편입니다. 물론 하드햇을 쓴다고 해서 숙련된 개발자가 되는 것은 아닙니다.

트러플이 Web3.js라는 이더리움 라이브러리를 기본으로 제공한다면 하드햇은 Ethers.js라는 라이브러리를 사용합니다. 또 “와플(Waffle)”이라는 단위 테스트 프레임워크를 함께 쓸 수 있습니다.



5.4.1. 하드햇 설치 (Windows 10, Powershell 기준)

트러플에서는 Node.js의 패키지 매니저인 npm을 사용했는데 이제부터는 yarn이라는 패키지 매니저를 사용하도록 하겠습니다. yarn은 다음과 같이 설치합니다.

```
> npm install -g yarn  
> yarn --version  
1.22.5
```

하드햇 역시 자바스크립트 기반의 개발 도구이므로 Node.js가 미리 설치되어 있어야 합니다. 트러플은 전역적으로 설치하여 사용하지만 하드햇은 프로젝트별로 설치하기 때문에 다른 곳으로 이식하기 쉽고 또 npx를 통해 실행하므로 최근 버전을 사용할 수 있다는 장점이 있습니다.

우선 프로젝트 디렉토리를 먼저 생성하고 yarn init으로 초기화 합니다.

```
> md hardhat-dapp  
> cd hardhat-dapp  
> yarn init -y
```

하드햇은 프로젝트 별로 설치하므로 프로젝트 디렉토리 안에서 개발 모드 --dev로 설치합니다.

```
hardhat-dapp> yarn add hardhat --dev
hardhat-dapp> dir

Mode           LastWriteTime       Length  Name
----           -----          -----  --
d-----      2021-11-17  오전 5:29          node_modules
-a----      2021-11-17  오전 5:29        154  package.json
-a----      2021-11-17  오전 5:29    105881  yarn.lock
```

이제 하드햇을 npx hardhat으로 실행하겠습니다.

```
hardhat-dapp> npx hardhat
888   888           888 888           888
888   888           888 888           888
888   888           888 888           888
888888888888 88888b. 888d888 .d888888 88888b. 8888b. 8888888
888   888   "88b 888P" d88" 888 888 "88b   "88b 888
888   888 .d888888 888   888 888 888 .d888888 888
888   888 888 888 888   Y88b 888 888 888 888 888 Y88b.
888   888 "Y888888 888   "Y888888 888 888 "Y888888 "Y888

Welcome to Hardhat v2.6.8

? What do you want to do? ...
> Create a basic sample project
  Create an advanced sample project
  Create an advanced sample project that uses TypeScript
  Create an empty hardhat.config.js
  Quit
```

화살표 키를 누르면 5가지 메뉴 중 하나를 선택할 수 있습니다. Quit는 종료입니다. 하드햇 역시 트러플처럼 권장 프로젝트 구조가 있습니다. 여기서는 첫번째 Create a basic sample project를 선택합니다. 이제 디렉토리 구조를 확인하면 아래와 같습니다.

```
✓ What do you want to do? · Create a basic sample project
✓ Hardhat project root: · C:\Users\song\bookapp\hardhat-dapp
✓ Do you want to add a .gitignore? (Y/n) · y

You need to install these dependencies to run the sample project:
yarn add --dev "hardhat@^2.6.8" "@nomiclabs/hardhat-waffle@^2.0.0" "ethereum-waffle@^3.0.0"
"chai@^4.2.0" "@nomiclabs/hardhat-ethers@^2.0.0" "ethers@^5.0.0"

Project created
See the README.txt file for some example tasks you can run.
```

```
PS C:\Users\song\bookapp\hardhat-dapp> dir
```

Mode	LastWriteTime	Length	Name
d-----	2021-11-17	오전 6:08	contracts
d-----	2021-11-17	오전 5:29	node_modules
d-----	2021-11-17	오전 6:08	scripts
d-----	2021-11-17	오전 6:08	test
-a----	2021-11-17	오전 6:08	83 .gitignore
-a----	2021-11-06	오후 7:26	572 hardhat.config.js
-a----	2021-11-17	오전 5:29	154 package.json
-a----	2021-11-06	오후 7:26	467 README.md
-a----	2021-11-17	오전 5:29	105881 yarn.lock

생성된 디렉토리들은 트러플과 유사합니다. 트러플의 migrations 디렉토리 대신 scripts 디렉토리가 있습니다. hardhat.config.js는 환경설정 파일입니다.

다음에는 필요한 패키지를 설치할 차례입니다. 하드햇은 경량의 개발환경을 유지하기 위해 디폴트로 설치되는 패키지(플러그인)를 최소화합니다. 그래서 처음 시작할 때는 여러 가지 패키지를 개발자가 직접 선택해서 설치해야 합니다.

Create a basic sample project을 선택하면 추가적으로 설치해야 하는 라이브러리들이 있습니다. 안내 메시지대로 추가 패키지를 설치합니다. 다소 시간이 소요될 수 있습니다.

```
yarn add --dev "@nomiclabs/hardhat-waffle@^2.0.0" "ethereum-waffle@^3.0.0" "chai@^4.2.0"  
"@nomiclabs/hardhat-ethers@^2.0.0" "ethers@^5.0.0"
```

5.4.2. 기본 설정

설정 파일 hardhat.config.js를 열어보면 무척 단순해 보입니다. 하지만 하나 둘 설정을 추가하기 시작하면 복잡해질 것입니다.

```
require("@nomiclabs/hardhat-waffle");

// This is a sample Hardhat task. To learn how to create your own go to
// https://hardhat.org/guides/create-task.html
task("accounts", "Prints the list of accounts", async (taskArgs, hre) => {
  const accounts = await hre.ethers.getSigners();

  for (const account of accounts) {
    console.log(account.address);
  }
});

// You need to export an object to set up your config
```

```
// Go to https://hardhat.org/config/ to learn more

/**
 * @type import('hardhat/config').HardhatUserConfig
 */
module.exports = {
  solidity: "0.8.4",
};
```

우선 hardhat-waffle이라는 컨트랙트용 단위 테스트 플러그인이 기본적으로 참조되어 있습니다. 앞서 추가적으로 설치한 패키지 중 하나입니다. 원래 와플은 독립적인 테스트 라이브러리입니다. 아래 홈페이지를 방문하면 더 많은 정보를 얻을 수 있습니다. 하드햇은 이렇게 외부의 독립적인 라이브러리를 플러그인으로 만들어서 개발 도구에 결합할 수 있도록 하고 있습니다.

<https://getwaffle.io/>

사이트의 플러그인 페이지를 보면 상당히 많은 플러그인들이 제공되고 있음을 알 수 있습니다. 공식 플러그인 뿐만 아니라 개발자 커뮤니티에서 만든 플러그인들도 많습니다.

<https://hardhat.org/plugins/>

그 다음은 태스크(task)입니다. 태스크는 반복되는 작업을 자동화할 수 있도록 작성하는 스크립트입니다. 예를 들어 컴파일이나 배포는 반복되는 작업이므로 태스크라고 할 수 있겠습니다. 물론 컴파일과 배포와 같이 필수적인 태스크들을 디폴트로 제공하고 있으므로 직접 만들 필요는 없습니다.

위에 예제 태스크는 accounts입니다. 이것은 하드햇에 내장된 가상 이더리움의 테스트 계정을 출력합니다. 태스크는 npx hardhat <태스크명>으로 실행하면 됩니다.

```
> npx hardhat accounts
0xf39Fd6e51aad88F6F4ce6aB8827279cffB92266
0x70997970C51812dc3A010C7d01b50e0d17dc79C8
0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC
0x90F79bf6EB2c4f870365E785982E1f101E93b906
0x15d34AAF54267DB7D7c367839AAF71A00a2C6A65
0x9965507D1a55bccC2695C58ba16FB37d819B0A4dc
...
```

메타마스크 지갑 계정으로 가나슈의 테스트 이더를 보내는 태스크가 필요하면 다음과 같이 작성할 수 있겠습니다.

```
task("send", "Send ETH to MetaMask account")
  .addParam("recipient", "MetaMask account")
  .setAction(async (args) => {
    const accounts = await ethers.getSigners();
```

```
const result = await accounts[0].sendTransaction(  
    {to: args.recipient, value: ethers.utils.parseEther("0.5")}  
);  
console.log(`TX HASH=${result.hash}`);  
});
```

태스크 실행은 아래와 같이 하면 됩니다.

```
npx hardhat send --recipient "0xb2b30a307214EbF4104cE4300d57eD0e6c9bd265" --network ganache
```

하드햇의 설정들은 `module.exports = { ... }` 안에 작성합니다. 솔리디티 컴파일러 버전은 `solidity`에서 지정할 수 있습니다. 단일 버전 또는 `compilers` 항목에 여러 버전을 설정할 수 있습니다. 컴파일 명령어는 `contracts` 디렉토리 아래에 있는 소스파일들을 컴파일 합니다.

```
solidity : {  
    compilers: [  
        {version: "0.5.2", settings: {optimizer: {enabled: true}}},  
        {version: "0.8.4", settings: {optimizer: {enabled: true}}}  
    ]  
}
```

트러플과 동일하게 네트워크 지정은 `networks`입니다. 정해진 네트워크 이름은 없지만 트러플과 동일하게 로컬 가나슈를 `development`로 하겠습니다. 가나슈의 체인 번호는 1337입니다(일반적으로 네트워크와 체인 번호는 동일하지만 현재 가나슈는 다르게 되어 있어 조금 혼란스럽습니다).

```
networks: {  
    development: {  
        url: "http://localhost:7545",  
        chainId: 1337  
    },  
    rinkeby: {  
        url: "https://eth-rinkeby.alchemyapi.io/v2/n6XQ...S3ik",  
        accounts: [...]  
    }  
}
```

디폴트 네트워크를 지정하고 싶을 때는 설정 파일에 다음과 같이 명시하면 됩니다.

```
defaultNetwork: "rinkeby",
```

이제 간단한 컨트랙트를 작성해서 컴파일과 배포를 해보도록 하겠습니다.

5.4.3 컴파일과 배포

앞서 작성한 SimpleStorage.sol을 동일하게 사용합니다. contracts 디렉토리 아래에 작성합니다. 컴파일은 compile 태스크를 실행합니다.

```
> npx hardhat compile  
Compiling 1 file with 0.8.4  
Compilation finished successfully
```

컴파일 결과는 artifacts/contracts/라는 디렉토리 아래에 소스파일명과 동일한 이름으로 디렉토리가 생기고 그 안에 컨트랙트 별로 JSON 파일이 생성됩니다.

JSON 파일은 트러플과 비교하면 단순합니다. ABI와 바이트 코드로 구성되어 있고 배포 주소는 저장되지 않습니다. 마찬가지로 컴파일 결과 생성 위치는 설정에서 변경할 수 있습니다.

```
{  
  "_format": "hh-sol-artifact-1",  
  "contractName": "SimpleStorage",  
  "sourceName": "contracts/SimpleStorage.sol",  
  "abi": [  
    {  
      "inputs": [  
        {  
          "internalType": "uint256",  
          "name": "s",  
          "type": "uint256"  
        }  
      ],  
      "stateMutability": "nonpayable",  
      "type": "constructor"  
    },  
    {  
      "anonymous": false,  
      "inputs": [  
        {  
          "indexed": false,  
          "internalType": "string",  
          "name": "message",  
          "type": "string"  
        },  
        {  
          "indexed": false,  
          "internalType": "uint256",  
          "name": "newVal",  
          "type": "uint256"  
        }  
      ],  
      "name": "Change",  
      "type": "function"  
    }  
  ]  
}
```

```

    "type": "event"
  },
{
  "inputs": [],
  "name": "get",
  "outputs": [
    {
      "internalType": "uint256",
      "name": "",
      "type": "uint256"
    }
  ],
  "stateMutability": "view",
  "type": "function"
},
{
  "inputs": [
    {
      "internalType": "uint256",
      "name": "x",
      "type": "uint256"
    }
  ],
  "name": "set",
  "outputs": [],
  "stateMutability": "nonpayable",
  "type": "function"
}
],
"bytecode": "0x6080604052348015610010...c2e2f55551443364736f6c63430008040033",
"deployedBytecode": "0x6080604052348015610010...c2e2f55551443364736f6c63430008040033",
"linkReferences": {},
"deployedLinkReferences": {}
}

```

이제 로컬 가나슈에 배포를 해보겠습니다(미리 가나슈를 실행해야 합니다). 배포 스크립트는 scripts 디렉토리 아래에 deploySimpleStorage.js로 작성합니다. 하드햇 스크립트 내에서는 기본적으로 Ethers.js를 쓸 수 있습니다. 다음과 같이 배포 스크립트를 작성합니다.

```

const hre = require("hardhat");

async function main() {
  const SimpleStorage = await hre.ethers.getContractFactory("SimpleStorage");
  const ss = await SimpleStorage.deploy(500);

  await ss.deployed();

  console.log("SimpleStorage deployed to:", ss.address);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
});

```

이제 배포를 실행합니다. 배포는 그냥 스크립트를 실행하는 것이기 때문에 실행 태스크인 run을 수행하면 됩니다.

```
> npx hardhat run ./scripts/deploySimpleStorage.js --network development
SimpleStorage deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

배포할 때 옵션 --network을 사용해서 네트워크를 development(가나슈)로 지정합니다. 생략하는 경우 하드햇의 내부 네트워크 "hardhat"에 배포됩니다. 설정 파일에 defaultNetwork 항목이 있으면 그 네트워크에 배포됩니다.

배포 계정은 항상 가나슈의 첫번째 계정이 되는데 이를 변경하려면 connect(signer)를 지정합니다.

```
const hre = require("hardhat");

async function main() {
    const SimpleStorage = await hre.ethers.getContractFactory("SimpleStorage");
    const signers = await hre.ethers.getSigners();
    const ss = await SimpleStorage.connect(signers[2]).deploy(500);

    await ss.deployed();

    console.log("SimpleStorage deployed to:", ss.address);
}

main()
    .then(() => process.exit(0))
    .catch((error) => {
        console.error(error);
        process.exit(1);
});
```

5.4.4. 단위 테스트

하드햇에서는 와플이라는 스마트 컨트랙트 전용 테스트 라이브러리를 플러그인 형태로 결합하여 사용할 수 있습니다. 이미 @nomiclabs/hardhat-waffle을 설치했습니다. 이제 테스트 스크립트 test.js 를 작성해보겠습니다.

```
const hre = require("hardhat");
const {expect, assert} = require("chai");

const addr = "0x53541aabFCc02dc6E8269ecA599214f7853ac8Ab";

describe ("SimpleStorage Test", function () {
```

```

before(async () => {
  const SimpleStorage = await hre.ethers.getContractFactory("SimpleStorage");
  this.instance = await SimpleStorage.attach(addr);
});

it ("Should have the initial value", async () => {
  const v = await this.instance.get();
  assert.equal(v.toString(), 500);
});

it("Should change the value", async () => {
  await this.instance.set(2000);
  const v = await this.instance.get();
  assert.equal(v, 2000);
});

it("Should emit the event by set", async () => {
  const v = 500;
  await expect(this.instance.set(v))
    .to.emit(this.instance, "Change")
    .withArgs("set", v);
});

it("Should revert", async () => {
  await expect(this.instance.set(10000))
    .to.be.revertedWith("Should be less than 5000");
});
}

)

```

와플을 사용했기 때문에 이벤트와 revert 테스트를 좀더 쉽게 할 수 있습니다. `expect().to.emit`을 사용하여 이벤트에서 발생한 값들을 확인할 수 있고 `expect().to.be.revertWith`로 revert reason을 비교할 수 있습니다.

테스트 스크립트 실행은 `test`라는 태스크로 수행하고 `--network` 옵션을 주어야 합니다.

```

> npx hardhat test ./test/test.js --network development

SimpleStorage Test
  ✓ Should have the initial value (149ms)
  ✓ Should change the value (513ms)
  ✓ Should emit the event by set (515ms)
  ✓ Should revert (121ms)

4 passing (2s)

```

하드햇은 내부(in-process) 가상 이더리움을 실행할 수 있기 때문에 외부 가나슈에 컨트랙트를 배포할 필요가 없이 컨트랙트 테스트가 즉시 가능합니다. 그래서 와플에서 제공하는 배포 기능을 이용해

서 다음과 같이 테스트 스크립트를 작성할 수도 있습니다.

```
const hre = require("hardhat");
const { deployContract, provider } = hre.waffle;
const {expect, assert} = require("chai");

const SimpleStorage = require("../artifacts/contracts/SimpleStorage.sol/SimpleStorage.json");

describe ("SimpleStorage Test", function () {

    before(async () => {
        const signers = provider.getSigner();
        this.instance = await deployContract(signers, SimpleStorage, [500]);
    });

    it ("Should have the initial value", async () => {
        const v = await this.instance.get();
        assert.equal(v.toString(), 500);
    });

    it("Should change the value", async () => {
        await this.instance.set(2000);
        const v = await this.instance.get();
        assert.equal(v, 2000);
    });

    it("Should emit the event by set", async () => {
        const v = 500;
        await expect(this.instance.set(v))
            .to.emit(this.instance, "Change")
            .withArgs("set", v);
    });

    it("Should revert", async () => {
        await expect(this.instance.set(10000))
            .to.be.revertedWith("Should be less than 5000");
    });
})
})
```

테스트 스크립트에서 직접 내부 네트워크에 배포 후에 각 테스트 케이스를 실행하고 종료합니다. 내부 네트워크에서 실행하는 것이므로 --network 옵션을 주지 않습니다.

```
> npx hardhat test ./test/test2.js

SimpleStorage Test
  ✓ Should have the initial value
  ✓ Should change the value (81ms)
  ✓ Should emit the event by set (87ms)
  ✓ Should revert

4 passing (1s)
```

하드햇은 컨트랙트에서 console.log를 사용할 수 있습니다. 이 기능이 개발자들에게 매우 좋은 반응을 얻었습니다. 컨트랙트에서 사용하려면 내부 네트워크를 사용해야 하고 console.sol 컨트랙트를 import 해야 합니다.

```
// SPDX-License-Identifier:MIT
pragma solidity ^0.8.0;

import "hardhat/console.sol";

contract SimpleStorage {
    uint256 storedData;

    event Change(string message, uint newVal);

    constructor (uint s) {
        storedData = s;
    }

    function set(uint x) public {
        console.log("passed argument = %d", x);
        require(x < 5000, "Should be less than 5000");
        storedData = x;
        emit Change("set", x);
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

자바스크립트의 console.log 처럼 원하는 곳에 삽입하여 값을 확인할 수 있습니다. 예를 들어 위와 같이 set 함수에 전달된 x의 값을 보고 싶다면 console.log(x)를 사용합니다(포맷을 줄 수도 있습니다). 내부 네트워크에서 테스트를 수행하면 다음과 같은 결과가 출력됩니다.

```
> npx hardhat test ./test/test2.js
Compiling 1 file with 0.8.4
Compilation finished successfully

SimpleStorage Test
  ✓ Should have the initial value
  passed argument = 2000
  ✓ Should change the value (82ms)
  passed argument = 500
  ✓ Should emit the event by set (62ms)
  passed argument = 10000
  ✓ Should revert

  4 passing (1s)
```

console 컨트랙트는 순전히 테스트 용도이므로 나중에 정식 배포할 때는 import에서 제거하고 console.log도 삭제하는 것이 좋습니다.

5.4.5. 플러그인

- `@nomiclabs/hardhat-ganache`

하드햇에서 제공하는 공식 플러그인 중에는 가나슈 플러그인이 있습니다. 이 플러그인을 사용하면 수동으로 가나슈를 실행하거나 종료할 필요가 없습니다. 하드햇의 내부 네트워크처럼 배포와 테스트를 수행한 후에 자동으로 종료합니다.

```
> yarn add @nomiclabs/hardhat-ganache --dev
```

플러그인을 활성화시키려면 `hardhat.config.js`에 추가해 주어야 합니다. 상단에 다음을 추가합니다.

```
require("@nomiclabs/hardhat-ganache");
```

이 플러그인을 쓰는 경우 네트워크 이름은 `ganache`로 정해져 있습니다.

```
> npx hardhat test ./test/test2.js --network ganache
```

- `hardhat-deploy`

하드햇은 내장된 배포 태스크가 없다고 봐도 무방합니다. 트러플처럼 명시적으로 `migrate`가 있는 것 이 아니고 그냥 `run` 태스크로 스크립트를 실행하는 것입니다. 그래서 배포 기능이 강화된 별도의 `hardhat-deploy` 플러그인이 존재합니다.

```
> yarn add hardhat-deploy --dev
```

설치 후에 플러그인을 hardhat.config.js에 추가합니다.

```
require("hardhat-deploy");
```

그리고 namedAccounts를 추가합니다. 이 설정은 다수의 네트워크에 배포할 때 사용할 계정을 지정할 수 있습니다. 여기서는 배포 계정 deployer 항목을 지정합니다. development는 현재 가나슈이므로 0은 가나슈의 첫번째 계정을 의미합니다. 계정 주소를 직접 써도 상관없습니다.

```
namedAccounts: {  
    deployer: {  
        development: 0  
    }  
},
```

배포 스크립트는 원래 scripts 안에 두었지만 hardhat-deploy에서는 deploy라는 디렉토리에서 배포 스크립트를 찾습니다(디폴트 설정입니다). deploy 디렉토리 아래에 deploySimpleStorage.js 를 작성합니다.

```
module.exports = async ({deployments, getNamedAccounts}) => {  
  
    const {deploy} = deployments;  
    const {deployer} = await getNamedAccounts();  
  
    const result = await deploy("SimpleStorage", {  
        from: deployer,  
        args: [500],  
    });  
    console.log(`Contract address = ${result.address}`);  
};
```

배포 스크립트에서 전달되는 파라미터는 deployments와 getNamedAccounts입니다. deploy() 함수의 첫 번째 파라미터는 컨트랙트의 이름이고 두 번째는 배포 계정과 초기값입니다.

hardhat-deploy의 배포 태스크는 deploy입니다.

```
> npx hardhat deploy --network development  
  
Compiling 1 file with 0.8.4  
Compilation finished successfully  
Contract address = 0xf1b5c54d3F13bf58734acF6AE7E32FFb24fe616F
```

배포가 끝나면 deployments라는 디렉토리가 생기고 배포 결과물이 네트워크 이름별로 저장됩니다. JSON 파일을 열어보면 배포 주소와 ABI, 바이트 코드, Receipt, 메타정보 등이 기록되어 있습니다.

```
> cd deployments\development

Mode           LastWriteTime      Length  Name
----           -----          -----  --
d-----       2021-11-19  오후 3:49          solcInputs
-a----       2021-11-19  오후 3:49           4   .chainId
-a----       2021-11-19  오후 3:49        80626 SimpleStorage.json
```

배포 스크립트가 여러 개 있는 경우에는 태그를 지정할 수 있습니다. 이렇게 하면 배포 스크립트 실행할 때 파일경로를 쓸 필요가 없어집니다. --tags 옵션을 사용하여 특정 스크립트만 실행할 수 있습니다.

```
module.exports = async ({deployments, getNamedAccounts}) => {

  const {deploy} = deployments;
  const {deployer} = await getNamedAccounts();

  const result = await deploy("SimpleStorage", {
    from: deployer,
    args: [500],
  });
  console.log(`Contract address = ${result.address}`);
};

module.exports.tags = ['simple'];
```

다음과 같이 실행합니다.

```
> npx hardhat deploy --tags simple --network development
```

컨트랙트 배포 후 트랜잭션 해시, 사용된 가스, 배포 주소 등의 정보를 보려면 log를 활성화시킬 수 있습니다. 기본값은 false이며 출력되지 않기 때문에 true로 변경합니다.

```
module.exports = async ({deployments, getNamedAccounts}) => {

  const {deploy} = deployments;
  const {deployer} = await getNamedAccounts();

  const result = await deploy("SimpleStorage", {
    from: deployer,
    args: [500],
```

```

        log: true
    });
};

module.exports.tags = ['simple'];

```

나중에 애플리케이션에서 필요한 컨트랙트의 주소와 ABI만을 따로 추출하여 저장할 수도 있습니다. 여러 컨트랙트의 ABI 정보를 하나의 파일로 만들 수도 있습니다. --export 옵션을 사용합니다.

```
> npx hardhat deploy --export SimpleStorage.abi.json --network development
```

생성된 SimpleStorage.abi.json은 다음과 같습니다.

```
{
  "name": "development",
  "chainId": "1337",
  "contracts": {
    "SimpleStorage": {
      "address": "0xC76c887a8899a02247B3fe77f06C8ff2E06Ce37b",
      "abi": [
        {
          "inputs": [
            {
              "internalType": "uint256",
              "name": "s",
              "type": "uint256"
            }
          ],
          "stateMutability": "nonpayable",
          "type": "constructor"
        },
        {
          "anonymous": false,
          "inputs": [
            {
              "indexed": false,
              "internalType": "string",
              "name": "message",
              "type": "string"
            },
            {
              "indexed": false,
              "internalType": "uint256",
              "name": "newVal",
              "type": "uint256"
            }
          ],
          "name": "Change",
          "type": "event"
        },
        {
          "inputs": []
        }
      ]
    }
  }
}
```

```

    "name": "get",
    "outputs": [
      {
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  },
  {
    "inputs": [
      {
        "internalType": "uint256",
        "name": "x",
        "type": "uint256"
      }
    ],
    "name": "set",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
}
}
}

```

- 이더스캔 컨트랙트 검증

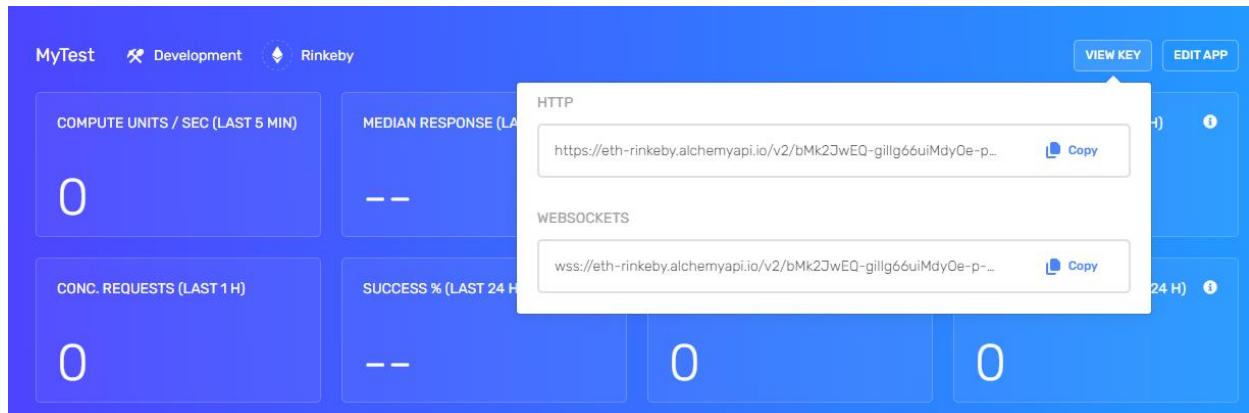
스마트 컨트랙트는 코드 검증을 위해서 공개된 사이트에 게시를 하는 것이 관례입니다. 단순히 컨트랙트를 작성한 사람을 믿기 보다는 코드를 신뢰하기 때문입니다.

이더스캔(<https://etherscan.io/>)은 대표적인 이더리움 블록 탐색기로, 블록과 트랜잭션을 조회할 수 있을 뿐만 아니라, 배포된 바이트 코드가 실제 어떤 소스코드인지 확인할 수 있는 정보도 제공합니다.

이를 위해서는 배포자가 소스코드와 바이트코드 등의 메타정보를 업로드해야 하는데 hardhat-deploy 플러그인은 이더스캔에서 컨트랙트 검증을 쉽게 할 수 있는 기능을 제공합니다.

테스트넷 Rinkeby에 컨트랙트를 배포하면서 이 기능을 사용해 보겠습니다. 우선 Rinkeby를 사용하려면 이더가 있어야 하므로 인터넷에서 “rinkeby faucet”를 검색하여 이더를 받습니다(또는 메타마스크에 링크된 faucet을 이용).

Rinkeby 네트워크에 연결하기 위해서는 이더리움 API 서비스를 이용해야 합니다. 여기서는 알케미 (<https://alchemyapi.io/>)를 이용합니다. 사이트에 가입 후 Rinkeby에 연결할 수 있는 API 키를 발급 받습니다.



hardhat.config.js 파일을 열어서 networks 항목에 rinkeby를 추가합니다.

```
const keys = ["cae19a0108...b74cfa"];

networks: {

  ...
  rinkeby: {
    url: "https://eth-rinkeby.alchemyapi.io/v2/{Your API key}",
    accounts: keys
  }
},
```

여기서 keys는 메타마스크 같은 지갑 소프트웨어에서 생성한 전자서명 개인키입니다. 가나슈는 가상 계정이기 때문에 전자서명이 필요하지 않았지만 테스트넷에서는 배포 트랜잭션을 전송할 때 전자서명을 해야 합니다.

그리고 namedAccounts 항목에 배포 계정을 추가합니다. 배포 계정은 개인키와 쌍을 이루는 지갑주소가 되겠습니다.

```
namedAccounts: {
  deployer: {
    development: 0,
    rinkeby: "0xAFc4...c08a"
  }
}
```

이더스캔은 개발자들이 컨트랙트 검증을 쉽게 할 수 있도록 검증 API를 제공합니다. hardhat-deploy 는 이것을 이용합니다. 이더스캔 사이트에서 API 키를 발급받습니다.

My API Keys		 Add
1 used (Out of 3 max quota)		
Action	Api-Key Token	
 Edit  Stat	1W4SQ 	JXQ
AppName: defi-dapp		
* API keys created on Etherscan.io can be used for the Kovan, Rinkeby, Ropsten, and Goerli Testnets.		

설정 파일에는 다음과 같이 etherscan 항목을 추가합니다.

```
etherscan: {
    apiKey: "1W4SQ...JXQ"
}
```

이제 준비가 끝났습니다. 먼저 Rinkeby 테스트넷에 컨트랙트를 배포합니다. --network rinkeby로 옵션을 지정합니다.

```
> npx hardhat deploy --network rinkeby
Nothing to compile
Contract address = 0x10798B4B3a7FA7D0252B3E8b3371f34AC9e2765a
```

배포된 컨트랙트를 블록 탐색기에서 조회해 볼 수 있습니다. Rinkeby 테스트넷의 블록 탐색기는 다음 링크를 방문합니다.

<https://rinkeby.etherscan.io/>

The screenshot shows the Etherscan contract overview for the address 0x10798B4B3a7FA7D0252B3E8b3371f34AC9e2765a. The 'Contract Overview' section displays a balance of 0 Ether. The 'Transactions' section shows one transaction from 6 mins ago, with details: Txn Hash: 0x82e8d743e7a31f5ecd..., Method: 0x60806040, Block: 9668571, From: 0xaafc4f9f3ba806d. The 'Contract' and 'Events' tabs are also visible.

이 컨트랙트는 정상적으로 동작하지만 소스코드와 바이트코드가 일치하는지 여부가 검증이 되지 않은 상태입니다.

hardhat-deploy 플러그인이 제공하는 etherscan-verify라는 태스크를 수행합니다.

```
> npx hardhat etherscan-verify --network rinkeby
verifying SimpleStorage (0x10798B4B3a7FA7D0252B3E8b3371f34AC9e2765a) ...
waiting for result...
=> contract SimpleStorage is now verified
```

이제 이더스캔에서 다시 컨트랙트를 조회해보겠습니다. 컨트랙트 탭에 체크 표시되어 있고 소스코드를 볼 수 있게 됩니다.

Transactions **Contract** ✓ Events

Code Read Contract Write Contract

✓ **Contract Source Code Verified** (Exact Match)

Contract Name: **SimpleStorage**

Compiler Version: **v0.8.4+commit.c7e474f2**

Contract Source Code (Solidity Standard Json-Input format)

File 1 of 2 : SimpleStorage.sol

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.0;
3
4 import "hardhat/console.sol";
5
6 contract SimpleStorage {
```

- hardhat-abi-exporter

이름에서 알 수 있듯이 애플리케이션에서 필요한 컨트랙트의 ABI만을 따로 추출하는 플러그인입니다. 앞서 hardhat-deploy에도 유사한 기능이 제공되지만 이 플러그인은 ABI만을 생성합니다.

```
> yarn add hardhat-abi-exporter --dev
```

설치 후에 플러그인을 설정 파일 hardhat.config.js에 추가합니다.

```
require('hardhat-abi-exporter');
```

그리고 설정 파일에 abiExporter라는 키를 추가합니다. 예를 들어 다음과 같이 설정할 수 있습니다.

```
abiExporter: {  
  path: "./app/src",  
  runOnCompile: true,  
  clear: true,  
  pretty: false,  
}
```

path는 ABI 파일이 저장되는 곳입니다. runOnCompile을 true로 설정하면 컴파일 즉시 exporter-abi 태스크를 자동으로 실행할 수 있습니다. 다시 말해서 컴파일이 되고 artifacts가 생긴 다음에 즉시 path 위치에 ABI 파일을 생성하게 합니다. clear를 true로 하면 이전에 있던 것들 것 삭제한 후에 다시 만듭니다.

예를 들어 다음과 같이 기존의 컴파일 태스크를 실행하면 ./app/src에 ABI 파일이 생성됩니다.

```
npx hardhat compile
```

runOnCompile은 디폴트로 false이므로 생략하면 컴파일 후에 추가적으로 export-abi 태스크를 실행해야 합니다. 그런데 export-abi 태스크는 항상 컴파일을 먼저 수행하기 때문에 export-abi 태스크만 실행해도 되겠습니다.

```
npx hardhat export-abi
```

ABI 파일은 지정된 path에 다음과 같이 생성됩니다.

```
└──src  
  └──contracts  
    └──SimpleStorage.sol  
      └──SimpleStorage.json
```

pretty 옵션은 ABI를 Ethers.js의 "Human-Readable"로 변환하여 생성합니다. SimpleStorage 컨트랙트의 ABI는 보통 아래와 같이 만들어집니다.

```
[  
 {  
   "inputs": [
```

```
{
  "internalType": "uint256",
  "name": "s",
  "type": "uint256"
},
],
"stateMutability": "nonpayable",
"type": "constructor"
},
{
  "anonymous": false,
  "inputs": [
    {
      "indexed": false,
      "internalType": "string",
      "name": "message",
      "type": "string"
    },
    {
      "indexed": false,
      "internalType": "uint256",
      "name": "newVal",
      "type": "uint256"
    }
  ],
  "name": "Change",
  "type": "event"
},
{
  "inputs": [],
  "name": "get",
  "outputs": [
    {
      "internalType": "uint256",
      "name": "",
      "type": "uint256"
    }
  ],
  "stateMutability": "view",
  "type": "function"
},
{
  "inputs": [
    {
      "internalType": "uint256",
      "name": "x",
      "type": "uint256"
    }
  ],
  "name": "set",
  "outputs": [],
  "stateMutability": "nonpayable",
  "type": "function"
}
]
```

pretty를 true로 설정하면 다음과 같이 사람이 알아보기 쉬운 형태로 변경됩니다.

```
[  
  "constructor(uint256)",  
  "event Change(string,uint256)",  
  "function get() view returns (uint256)",  
  "function set(uint256)"  
]
```

"Human-Readable" ABI는 Ethers.js를 사용하여 컨트랙트 인스턴스를 생성할 때 기존 ABI와 동일하게 사용할 수 있습니다.

이외에도 하드햇은 다양한 플러그인이 존재합니다. 하드햇 사이트에서 목적에 맞는 플러그인을 찾아서 효율적으로 활용하기 바랍니다.

<https://hardhat.org/plugins/>

5.5. 정리

이번 장에서는 스마트 컨트랙트 개발에 가장 많이 활용되고 있는 트러플과 하드햇에 대해 알아보았습니다. 다음 6장에서 컨트랙트를 작성할 때는 하드햇을 이용할 것입니다.

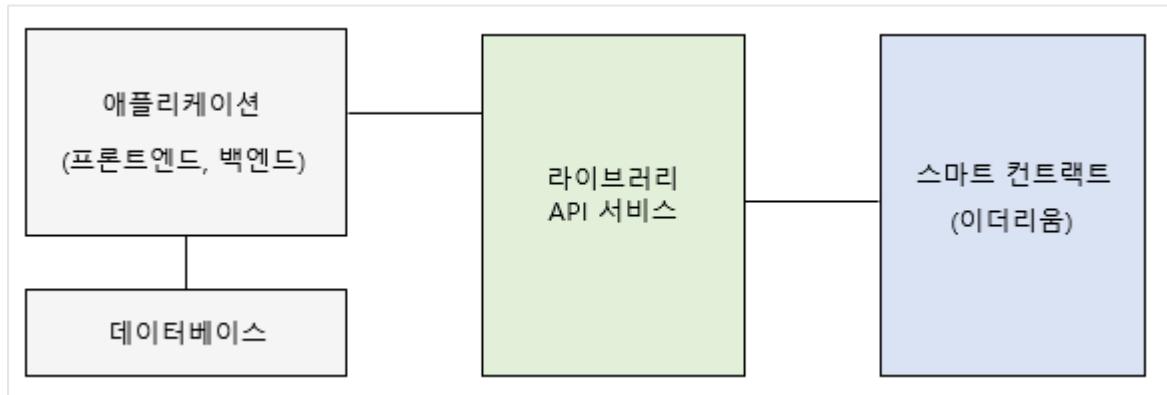
Simple NFT 프로젝트



6장에서는 솔리디티 스마트 컨트랙트와 자바스크립트 라이브러리인 리액트를 사용하여 간단한 Dapp을 만들어 보겠습니다. 스마트 컨트랙트와 애플리케이션이 어떻게 결합될 수 있는지 알게 될 것입니다.

6.1. 스마트 컨트랙트와 애플리케이션

5장에서 이야기한 것처럼 스마트 컨트랙트만으로 모든 것을 할 수는 없습니다. 컨트랙트가 중요한 데이터를 저장할 수는 있겠지만 사람들이 사용할 수 있는 애플리케이션이 되려면 프론트엔드 애플리케이션이 있어야 하고 백엔드 애플리케이션, 데이터베이스도 함께 활용해야 합니다.



이번 장에서는 스마트 컨트랙트와 연결되는 간단한 프론트엔드 애플리케이션을 구현하면서 다음 두 가지 주제를 다룰 것입니다.

- 오픈제펠린 라이브러리를 사용하여 ERC-721 표준을 구현한 컨트랙트

- 리액트 라이브러리를 활용한 프론트엔드 애플리케이션

사실 프론트엔드 애플리케이션 개발은 블록체인 그 자체와는 크게 관련이 없지만 스마트 컨트랙트와 애플리케이션이 어떻게 데이터를 주고받는지 이해하는 것은 블록체인 기반의 서비스를 개발할 때 중요한 요소입니다.

6.2. 시나리오

앨리스는 인터넷에서 NFT가 유행이라는 소식을 접하고 자신도 이더리움 상에서 NFT를 발행해서 판매보기로 합니다. 토큰 이름은 "Alice Loves Sea(ALS)"라고 정했습니다. 대략 500개 정도의 ALS를 발행해서 NFT 마켓플레이스인 OpenSea(<https://opensea.io/>)에 게시할 생각입니다.

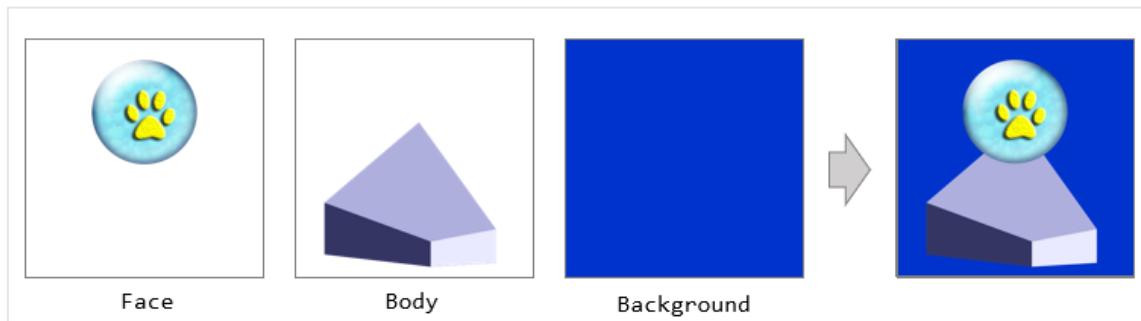
물론 OpenSea에서 제공하는 NFT 발행 기능을 사용해도 되지만 직접 스마트 컨트랙트를 작성하고, 애플리케이션도 만들 계획입니다.

- NFT 이미지

OpenSea를 둘러보니 대부분의 NFT는 어떤 이미지로 표현되는 것을 알게 되었습니다. 그렇다면 NFT 500개를 만들기 위해서는 500개의 이미지를 만들어야 하는데?

고민 끝에 세 가지 각각 다른 이미지를 만들고 그것을 프로그램에서 조합해서 최종 이미지 500개를 생성하기로 합니다. 앤리스는 이미지 요소를 Face, Body, Background로 구성하기로 합니다. 아래 그림은 앤리스가 이미지 편집도구를 사용하여 만든 세 개의 프로토타입 이미지입니다.

앤리스는 미리 각각의 요소들의 위치를 고정한 이미지를 만들었습니다. 프로그램에서 이미지 위치를 정해볼까 생각했지만 그냥 처음부터 위치를 "하드코딩"하기로 합니다. 프로그램에서는 이것들을 합치기만 하면 됩니다.



이제 남은 작업은 다양한 특징(trait)을 가진 Face, Body, Background를 만드는 일입니다. 서로 유일하게 구별되는 500개를 만들어야 하므로 경우의 수가 충분히 나올 수 있도록 해야 합니다.

일단 Face를 주요 속성으로 잡고 100개 정도의 이미지를 만들기로 합니다. 그리고 Body는 5개, Background는 2개만 제작하기로 합니다. 그렇다면 가능한 조합은?

$$100 \times 5 \times 2 = 1000$$

500개가 목표이므로 충분히 서로 다른 이미지를 만들어낼 수 있을 것 같습니다. 랜덤하게 이미지를 생성할 것이므로 확률적으로 동일한 조합이 나올 수 있습니다. 하지만 Face 속성의 경우 연속 3회 같은 이미지가 선택될 확률은 백만 분의 일 정도입니다.

- 스마트 컨트랙트

단일한 유형의 NFT를 발행할 것이므로 ERC-721 표준이 맞을 것 같습니다. 앤리스는 ERC-721 표준을 읽어봤지만 막상 실제 구현을 하려고 보니 막막해졌습니다. 다행히 개발자 커뮤니티에 물어보니 이더리움의 스마트 컨트랙트는 보통 솔리디티로 작성하고 “오픈제펠린”이라는 라이브러리를 쓰면 수월하게 구현할 수 있다는 것을 알게 됩니다.

- 자바스크립트 애플리케이션

이제 앤리스는 애플리케이션을 어떻게 만들지 고민합니다. 웹으로 접근해야 하므로 웹브라우저에서 실행되는 웹 애플리케이션으로 만들어야 합니다. 그렇다면 자바스크립트를 선택할 수밖에 없습니다. 자바스크립트 라이브러리 중에서 가장 인기있는 리액트를 사용하기로 합니다.

- 개발도구

스마트 컨트랙트 개발도구를 검색해보니 “트러플”이나 “하드햇”이라는 도구를 쓴다고 합니다. 앤리스는 초콜릿보다 안전모를 쓰기로 합니다.

개발 순서를 간단히 요약하면 아래와 같습니다.

- ① 각 속성들의 이미지를 준비합니다.

- ② 이미지들을 조합하여 하나의 이미지를 생성하는 스크립트를 작성합니다.
- ③ 생성된 이미지를 IPFS에 업로드하고 메타정보를 생성합니다. 메타정보는 JSON 형식으로 IPFS에 업로드합니다.
- ④ ERC-721 표준을 준수하는 스마트 컨트랙트를 작성합니다.
- ⑤ 리액트를 사용하여 NFT를 발행할 수 있는 애플리케이션을 만듭니다. 여기서 “발행”이라는 것은 미리 생성된 메타정보가 나타내는 NFT의 소유권 정보를 스마트 컨트랙트에 기록하는 것을 의미합니다.
- ⑥ 발행된 토큰을 OpenSea에서 조회하고 판매합니다. OpenSea API를 이용하여 판매상태 등을 애플리케이션에서 조회할 수 있도록 합니다.

6.3. 이미지 준비

6.3.1. 이미지 조합

자바스크립트를 쓰기로 했으므로 먼저 Node.js 프로젝트를 생성합니다. nft-gen 디렉토리를 만들고 yarn int -y 로 초기화합니다.

```
nft-gen> yarn init -y
```

images라는 디렉토리를 만들고 다시 그 아래에 각각 Face, Body, Background 디렉토리를 생성합니다. _Final은 완성 이미지가 들어가는 곳입니다. 이미지 편집도구를 사용하여 각 속성의 이미지를 만듭니다. 아래 그림은 Face 속성의 이미지들입니다.



이미지 파일명은 단순하게 1.png, 2.png, ... 형식의 순번을 주기로 하겠습니다. Face 속성은 1에서 100 까지, Body는 1에서 5까지, Background는 1에서 2까지 파일이 존재하게 됩니다. 이미지의 크기는 500x500 픽셀의 PNG 파일입니다.

Mode	LastWriteTime	Length	Name
d-----	2021-11-18	오전 6:08	Background
d-----	2021-11-18	오전 6:08	Body
d-----	2021-11-18	오전 6:08	Face
d-----	2021-11-19	오후 12:43	_Final

6.3.2. NFT 속성

ALS 토큰은 세가지 속성을 가지고 있어서 이것을 배열로 표현하기로 합니다. 속성의 값은 숫자로 나타내기로 합니다. 그리고 속성의 이미지는 id와 name을 갖도록 합니다. 예를 들어 traits.js에 다음과 같이 작성할 수 있겠습니다.

```
const face = [
    {id: 1, name: 'one'}
    ,{id: 2, name: 'two'}
    ,{id: 3, name: 'three'}
    ,{id: 4, name: 'four'}
    ,{id: 5, name: 'five'}
    ,{id: 6, name: 'six'}
    ,{id: 7, name: 'luckyone'}
    ...
    ,{id: 100, name: 'onezerozero'}]

const body = [
    {id: 1, name: 'greybody'}
    ,{id: 2, name: 'greenbody'}
    ,{id: 3, name: 'pinkbody'}
    ,{id: 4, name: 'yellowbody'}
    ,{id: 5, name: 'orangebody'}
];

const background = [
    {id: 1, name: 'blue'}
    ,{id: 2, name: 'green'}
];
```

id는 물리적인 이미지 파일명과 일치하게 합니다. 즉 Face 속성 {id: 1, name: 'one'}은 Face 디렉토리 안에 있는 1.PNG 파일에 대응합니다.

name 속성은 이미지 컨셉에 맞는 이름을 붙입니다. 유일하게 구별할 수 있는 이름으로 하는 것이 좋겠습니다(여기서는 속성을 구별하기 위한 것으로 특별한 의미를 부여하지 않았습니다). 속성을 표

시할 때는 name 대신 id를 쓰기로 합니다. 그러니까 Face 배열의 54번 속성, Body 배열의 3번 속성 그리고 Background 속성의 1번 속성으로 이루어진 ALS 토큰의 이미지는 다음과 같이 표현할 수 있습니다.

```
ALS = [Face, Body, Background] = [54, 3, 1]
```

그래서 ALS 토큰의 집합은 500개의 원소로 이루어진 2차원 배열이 됩니다.

```
let ALS = new Array(500);
ALS = [[54, 3, 1], [3, 5, 2], [67, 1, 1] ...[10, 3, 2]];
```

이제 속성들의 조합을 랜덤하게 만들어서 배열을 만드는 함수를 작성합니다. 난수를 하나 발생시킨 다음 traits.js에서 정의한 각 속성의 배열 인덱스로 사용합니다. 예를 들어 Face 속성 100개 중에 하나를 선택하는 것은 아래와 같이 작성될 수 있습니다.

```
const NUM_OF_FACES = 100;

const fnRng = (limit) => {
    return Math.floor(Math.random() * limit);
}

...
let nftTobe = [];
nftTobe.push(face[fnRng(NUM_OF_FACES)].id);
```

그런데 이렇게 만들어진 조합이 이미 존재한다면 ALS 토큰 집합에 넣지 말아야 합니다. 그래서 선택된 조합 nftTobe가 이제까지 저장된 배열 NFTs에 존재하면 건너뛰기로 합니다.

```
for (let i=0; i<NFTs.length; i++) {
    if (JSON.stringify(NFTs[i]) === JSON.stringify(nftTobe)) {
        return null;
    }
}
```

최종적으로 완성된 함수는 다음과 같습니다. 이 함수는 난수를 발생시켜서 각 속성을 선택하고 중복되지 않는 조합 하나를 리턴하게 될 것입니다.

```
const { face, body, background } = require("./traits.js");
```

```

const NUM_OF_FACES = 100;
const NUM_OF_BODY = 5;
const NUM_OF_BACKGROUND = 2;

const fnGenerateWithoutRedundancy = () => {
  let nftTobe = [];

  nftTobe.push(face[fnRng(NUM_OF_FACES)].id);
  nftTobe.push(body[fnRng(NUM_OF_BODY)].id);
  nftTobe.push(background[fnRng(NUM_OF_BACKGROUND)].id);

  if (NFTs.length > 0) {
    for (let i=0; i<NFTs.length; i++) {
      if (JSON.stringify(NFTs[i]) === JSON.stringify(nftTobe)) {
        return null;
      }
    }
  }
  return nftTobe;
}

```

그런데 Face의 속성 중에 하나는 발행량을 적게 하여 희귀성을 부여하고 싶습니다. 예를 들어 Face 77번 이미지를 가진 토큰을 2개 이하로 만들고 싶다면? 물론 데이터를 직접 수정해도 되지만 토큰 생성 함수를 실행할 때 적용하려고 합니다.

특정 Face 속성 RARE_TRAIT를 가진 토큰이 이미 정해진 수량 MAX_NUM_OF_RARITY에 도달했다면 다른 속성을 선택하도록 하는 함수를 작성합니다. 연속해서 동일한 속성이 선택될 확률은 그렇게 크지 않으므로 무한 재귀(recursive) 호출은 일어나지는 않을 것입니다.

```

const RARE_TRAIT = 77;
const MAX_NUM_OF_RARITY = 2;

const fnCheckRareTrait = (t) => {
  if (NFTs.length > 0 && t === RARE_TRAIT) {
    totalCountOfRareTrait++;
    if (totalCountOfRareTrait > MAX_NUM_OF_RARITY) {
      totalCountOfRareTrait--;
      return fnCheckRareTrait(face[fnRng(NUM_OF_FACES)].id);
    }
    return t;
  } else {
    return t;
  }
}

```

이 함수를 토큰 생성 함수 fnGenerateWithoutRedundancy에 적용하면 되겠습니다. 편의상 Face 속성에만 적용하기로 합니다.

```
const fnGenerateWithoutRedundancy = () => {
  let nftTobe = [];

  nftTobe.push(fnCheckRareTrait(face[fnRng(NUM_OF_FACES)].id));
  nftTobe.push(body[fnRng(NUM_OF_BODY)].id);
  nftTobe.push(background[fnRng(NUM_OF_BACKGROUND)].id);

  if (NFTs.length > 0) {
    for (let i=0; i<NFTs.length; i++) {
      if (JSON.stringify(NFTs[i]) === JSON.stringify(nftTobe)) {
        return null;
      }
    }
  }
  return nftTobe;
}
```

물론 랜덤으로 조합되기 때문에 임의의 속성이 매우 드물게 생성될 수 있습니다. 그러나 임의로 특정 속성의 개수를 제한하는 것이 프로그램적으로 가능합니다.

목표 수량 500개를 생성하려면 반복문을 이용하여 실행하면 됩니다.

```
const TARGET_NUM_OF_NFT = 500;

while (NFTs.length < TARGET_NUM_OF_NFT) {
  const n = fnGenerateWithoutRedundancy();
  if (n !== null) {
    NFTs.push(n);
  }
}
```

전체 소스 파일은 다음과 같습니다.

```
const { face, body, background } = require("./traits.js");

const NUM_OF_FACES = 100;
const NUM_OF_BODY = 5;
const NUM_OF_BACKGROUND = 2;

const TARGET_NUM_OF_NFT = 500;
const RARE_TRAIT = 77;
const MAX_NUM_OF_RARITY = 2;

let NFTs = [];
let totalCountOfRareTrait = 0;
```

```

const fnRng = (limit) => {
    return Math.floor(Math.random() * limit);
}

const fnGenerateWithoutRedundancy = () => {
    let nftTobe = [];

    nftTobe.push(fnCheckRareTrait(face[fnRng(NUM_OF_FACES)].id));
    nftTobe.push(body[fnRng(NUM_OF_BODY)].id);
    nftTobe.push(background[fnRng(NUM_OF_BACKGROUND)].id);

    if (NFTs.length > 0) {
        for (let i=0; i<NFTs.length; i++) {
            if (JSON.stringify(NFTs[i]) === JSON.stringify(nftTobe)) {
                return null;
            }
        }
    }
    return nftTobe;
}

const fnCheckRareTrait = (t) => {
    if (NFTs.length > 0 && t === RARE_TRAIT) {
        totalCountOfRareTrait++;
        if (totalCountOfRareTrait > MAX_NUM_OF_RARITY) {
            totalCountOfRareTrait--;
            return fnCheckRareTrait(face[fnRng(NUM_OF_FACES)].id);
        }
        return t;
    } else {
        return t;
    }
}

while (NFTs.length < TARGET_NUM_OF_NFT) {
    const n = fnGenerateWithoutRedundancy();
    if (n !== null) {
        NFTs.push(n);
    }
}

console.log(`TOTAL_NUM_OF_NFT = ${NFTs.length}`);
console.log(`TOTAL_NUM_OF_RARITY = ${totalCountOfRareTrait}`);

```

프로그램을 실행한 후 결과를 파일로 저장하고 토큰이 어떻게 만들어졌는지 보겠습니다. 대략 아래와 같은 데이터가 생성됩니다(500건 중 일부).

FACE	BODY	BACKGROUND
59	3	1
56	3	1
71	3	1
73	3	2
60	2	2

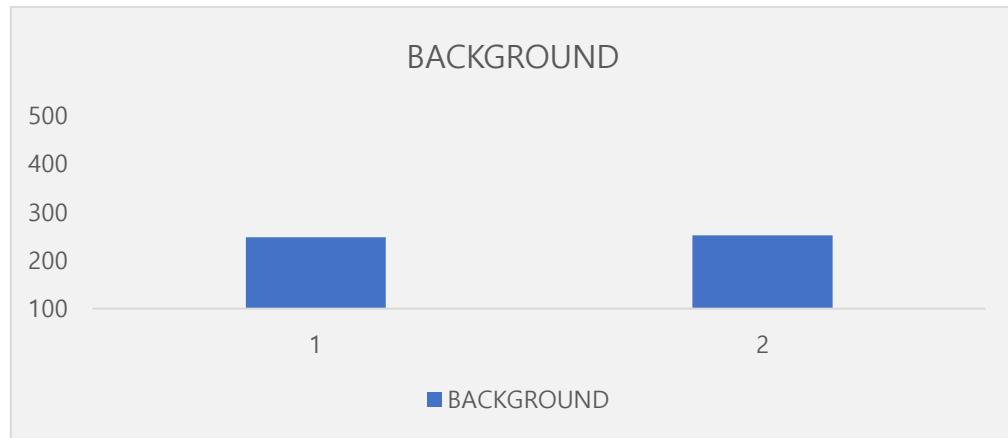
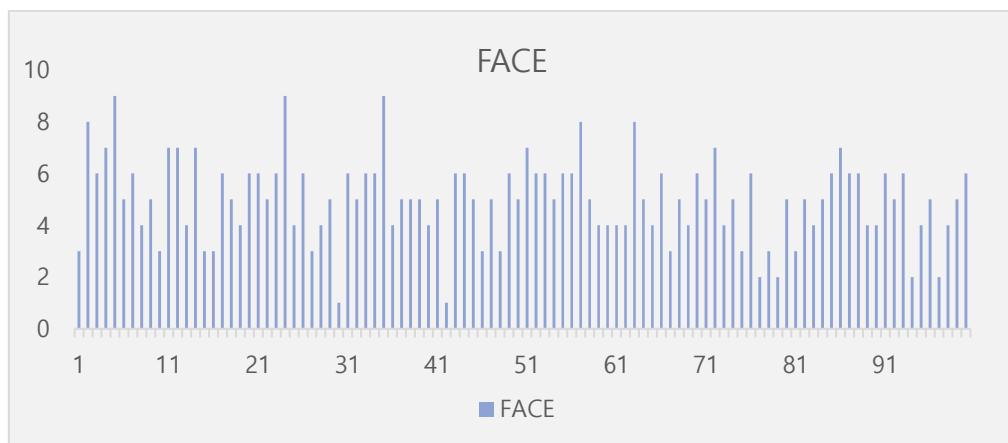
11	5	1
23	2	2
18	2	2
55	5	2
66	1	2
20	3	2
29	1	2
44	1	2
21	5	2
40	4	2
49	4	1
생략		

엑셀 COUNTIF 함수를 이용하여 속성 별로 카운트하면 다음과 같습니다(Face의 경우)

속성	개수	비율
1	3	0.6%
2	8	1.6%
3	6	1.2%
4	7	1.4%
5	9	1.8%
6	5	1.0%
7	6	1.2%
8	4	0.8%
9	5	1.0%
10	3	0.6%
11	7	1.4%
12	7	1.4%
13	4	0.8%
14	7	1.4%
15	3	0.6%
16	3	0.6%
17	6	1.2%
18	5	1.0%
19	4	0.8%

20	6	1.2%
21	6	1.2%
생략		
70	6	1.2%
71	5	1.0%
72	7	1.4%
73	4	0.8%
74	5	1.0%
75	3	0.6%
76	6	1.2%
77	2	0.4%
78	3	0.6%
79	2	0.4%
80	5	1.0%
81	3	0.6%
82	5	1.0%
83	4	0.8%
84	5	1.0%
85	6	1.2%
86	7	1.4%
87	6	1.2%
88	6	1.2%
89	4	0.8%
90	4	0.8%
91	6	1.2%
92	5	1.0%
93	6	1.2%
94	2	0.4%
95	4	0.8%
96	5	1.0%
97	2	0.4%
98	4	0.8%
99	5	1.0%
100	6	1.2%
합계	500	100.0%

77번 속성을 가진 토큰은 원하는 대로 2개만 생성되었고 비율은 0.4% 입니다. 랜덤으로 만들어지는 것이므로 다른 속성들 중에도 얼마든지 더 낮은 비율이 있을 수 있습니다. 각 속성별 분포는 다음과 같습니다. 속성의 개수가 많은 Face는 다소 산만하지만 속성의 개수가 적은 Body와 Background는 비교적 균등한 분포를 보여주고 있습니다.



6.3.3. 이미지 생성

세 개의 이미지를 레이어로 합치기 위해서는 node-canvas라는 라이브러리가 필요합니다. 다음과 같이 설치합니다.

```
yarn add canvas --dev
```

이미 데이터는 만들어졌으므로 배열에 저장된 데이터를 하나씩 읽어서 각 디렉토리에 있는 속성 파일들을 합치면 되겠습니다. 파일명은 속성 id와 일치하도록 했으므로 잘못된 파일이 선택되지는 않을 것입니다. 우선 canvas를 초기화 합니다. 크기는 500x500으로 정합니다.

```
const { createCanvas, loadImage } = require("canvas");

const canvas = createCanvas(500, 500);
const ctx = canvas.getContext('2d');
```

canvas에서 제공하는 loadImage() 함수를 사용하여 이미지를 가져옵니다. 전달인자는 경로를 포함하는 파일명입니다.

```
const { createCanvas, loadImage } = require("canvas");

const FILE_PATH = "./images";

const face = await loadImage(`.${FILE_PATH}/Face/${t[0]}.png`);
const body = await loadImage(`.${FILE_PATH}/Body/${t[1]}.png`);
const background = await loadImage(`.${FILE_PATH}/Background/${t[2]}.png`);
```



여기서 t는 앞서 생성한 ALS 토큰 배열에 담긴 원소입니다. 즉 t[0]은 Face 속성을, t[1]은 Body 속성을, t[2]는 Background 속성을 나타내게 됩니다. 속성 번호와 파일명은 일치하기 때문에 그대로 사용해도 됩니다.

canvas로 이미지를 합치려면 순서가 중요합니다. 맨 아래에 있어야 하는 이미지를 먼저 그리도록 합니다.

결과적으로 Background, Body, Face 순서가 될 것입니다. 각 요소의 위치는 이미지를 만들 때부터 고정되어 있으므로 모두 동일한 위치 (0,0)에 그리면 되겠습니다.

```
await ctx.drawImage(background, 0, 0, 500, 500);
await ctx.drawImage(body, 0, 0, 500, 500);
await ctx.drawImage(face, 0, 0, 500, 500);
```

이렇게 합친 이미지를 다시 하나의 파일로 저장하면 원하는 결과가 완성될 것입니다. 파일 저장 함수 `saveImage()`는 다음과 같이 구현합니다. 완성된 최종 파일명의 형식은 N001.png, N002.png, ... N500.png로 하겠습니다.

```
const saveImage = (canvas, index) => {
  const filename = `N${index.toString().padStart(3,0)}`;
  fs.writeFileSync(`.${FILE_PATH}/_Final/${filename}.png`, canvas.toBuffer("image/png"));
  //console.log(filename);
};
```

전체 소스파일 `create.js`는 다음과 같습니다.

```
const fs = require("fs");
const { createCanvas, loadImage } = require("canvas");

const canvas = createCanvas(500, 500);
const ctx = canvas.getContext('2d');

const FILE_PATH = "./images";

const saveImage = (canvas, index) => {
  const filename = `N${index.toString().padStart(3,0)}`;
  fs.writeFileSync(`.${FILE_PATH}/_Final/${filename}.png`, canvas.toBuffer("image/png"));
  //console.log(filename);
};

const create = async (t, i) => {

  const face = await loadImage(`.${FILE_PATH}/Face/${t[0]}.png`);
  const body = await loadImage(`.${FILE_PATH}/Body/${t[1]}.png`);
  const background = await loadImage(`.${FILE_PATH}/Background/${t[2]}.png`);

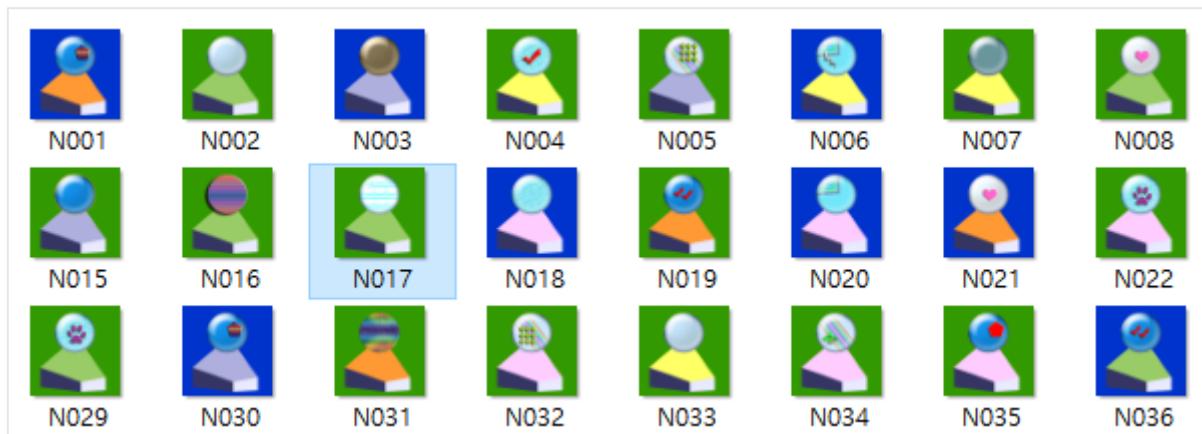
  await ctx.drawImage(background, 0, 0, 500, 500);
  await ctx.drawImage(body, 0, 0, 500, 500);
  await ctx.drawImage(face, 0, 0, 500, 500);

  saveImage(canvas, i+1);
};
```

`create()` 함수는 세 개의 이미지를 조합해서 완성된 NFT 이미지를 만드는 것입니다. 500개를 만들려면 NFTs 배열에 저장된 조합을 차례로 불러와서 `create`를 반복문에서 실행합니다.

```
(async () => {
  console.log("Creating...");
  for (let i=0; i<NFTs.length; i++) {
    await create(NFTs[i], i);
  }
})();
```

결과는 아래와 같습니다. N001부터 N500까지 이미지가 생성되었습니다. 몇 개를 샘플링하여 실제 조합과 만들어진 이미지가 맞는지 확인해보면 되겠습니다.



6.3.4. 메타정보 업로드

NFT는 메타정보가 있어야 합니다. 이미지가 모두 생성된 후 메타정보를 만들기로 하겠습니다. 일반적으로 메타정보는 JSON 형식인데, OpenSea의 형식을 따르겠습니다. 예를 들어 ALS 10번의 메타정보는 아래와 같은 형식이 될 것입니다.

```
{
  "description": "ALS::Alice Loves Sea NFT",
  "name": "ALS-10",
  "type": "Collectable",
  "image": "https://",
  "attributes": [
    {
      "trait_type": "Face",
      "value": "onezerozero"
    },
    {
      "trait_type": "Body",
      "value": "yellowbody"
    }
  ]
}
```

```
{  
    "trait_type": "Background",  
    "value": "blue"  
}  
]
```

메타정보 중 image 항목은 NFT를 대표하는 이미지가 위치한 곳으로 보통 웹에서 접근 가능한 URL 이거나 분산 파일 시스템 IPFS의 컨텐츠 번호(CID)가 될 것입니다. 이미지 업로드 후에 메타정보의 image 위치를 업데이트 할 수 있습니다.

여기서는 NFT 스토리지(<https://nft.storage/>)라는 무료 IPFS 서비스를 이용하도록 하겠습니다. NFT 스토리지를 이용하려면 우선 API 키를 받아야 합니다. 회원 가입을 하고 API 키를 발급받습니다.

API Keys

Name	Key
ALS-NFT	eyJhbGciOiJIUzI1NilsInR5cCI6IkpX

NFT 스토리지에 파일을 전송하려면 우선 nft.storage라는 라이브러리를 설치해야 합니다.

```
yarn add nft.storage --dev
```

사용법은 간단합니다. 발급받은 API 키를 가지고 NFT 스토리지 객체를 하나 생성한다음 그것을 이용하여 NFT 스토리지에 파일을 업로드하는 것입니다. 여기서는 client라는 이름으로 객체를 하나 만들었습니다.

메타정보를 업로드하는 함수 uploadMetaData()를 작성합니다. 메타정보에는 속성 정보도 필요하기 때문에 getAttributes()라는 함수를 만들어서 각 속성의 이름을 가져옵니다.

```
const {NFTStorage, File } = require("nft.storage");  
const apiKey = "eyJhbGci...iRApXYeCDsAAc"  
const client = new NFTStorage({ token: apiKey });  
  
const getAttributes = (v, k) => {
```

```

let attributes = {};
let trait_type = "";
let value = "";

switch (k) {
  case 0:
    trait_type = "Face";
    value = face[v-1].name;
    break;
  case 1:
    trait_type = "Body";
    value = body[v-1].name;
    break;
  case 2:
    trait_type = "Background";
    value = background[v-1].name;
    break;
  default:
    trait_type = "";
    value = "";
}

attributes.trait_type = trait_type;
attributes.value = value;

return attributes;
}

const uploadMetaData = async (t, i) => {

let metadata = {
  description: "ALS::Alice Loves Sea NFT",
  name: `ALS-${i}`,
  type: "Collectable",
  image: "https://",
  attributes: [],
};

for (let k=0; k<3; k++) {
  metadata.attributes.push(getAttributes(t[k], k));
}

const filename = `N${i.toString().padStart(3,0)}`;
metadata.image = new File(
  [await fs.promises.readFile(`${FILE_PATH}/_Final/${filename}.png`)],
  `${filename}.png`, {type: 'image/png'});
}

const result = await client.store(metadata);
console.log(result.url);
saveMetadataUri(` ${i}=${result.url}`);
}

```

uploadMetaData() 함수는 이미지가 모두 생성된 후에 호출합니다. 왜냐하면 이미지가 먼저 올라간 이후에 메타정보에 이미지 URI 정보를 넣어줄 수 있기 때문입니다. 그런데 nft.storage 라이브러리에서 제공하는 File 함수 덕분에 이미지를 별도로 업로드하지 않고 메타정보 파일을 올리면서 동시에 이미지 파일도 함께 업로드할 수 있습니다. 리턴 값이 이미지 URI이기 때문에 image 항목에 그대로

넣으면 되겠습니다.

앞에서 작성한 `create.js` 에서 `saveImage()`를 호출 후 `uploadMetaData()`를 실행합니다.

```
const create = async (t, i) => {
  const face = await loadImage(`.${FILE_PATH}/Face/${t[0]}.png`);
  const body = await loadImage(`.${FILE_PATH}/Body/${t[1]}.png`);
  const background = await loadImage(`.${FILE_PATH}/Background/${t[2]}.png`);

  await ctx.drawImage(background, 0, 0, 500, 500);
  await ctx.drawImage(body, 0, 0, 500, 500);
  await ctx.drawImage(face, 0, 0, 500, 500);

  saveImage(canvas, i+1);

  await uploadMetaData(t, i+1); // metadata upload to IPFS
};
```

500개를 모두 업로드하는 것은 시간이 걸립니다. 시험삼아 몇 개만 테스트해보도록 합니다. 다음과 같이 IPFS의 CID, 즉 컨텐츠 해시 값이 콘솔에 출력됩니다.

```
ipfs://bafyreiep55j4fhvyqujqsqhtcuhxbmqvzbvtnvpefa460xdys2eas5ysiymetadata.json
ipfs://bafyreifqphue5mixtfa2s37tm2wqdrwnrqp3vhcy7xdf2ygscxfdhisamq/metadata.json
ipfs://bafyreibom6dptsrln2zsctasgq2k1wfkh05cx3ru3enigzpqqjjylewsu4a/metadata.json
...
```

크롬 브라우저에서 링크를 열면 메타정보가 나오는 것을 알 수 있습니다. ipfs:// 링크를 직접 열려면 IPFS 크롬 확장 프로그램을 설치하거나, 공용 게이트웨이 `gateway.ipfs.io`를 이용하면 볼 수 있습니다.

<https://gateway.ipfs.io/ipfs/bafyreiep55j4fhvyqujqsqhtcuhxbmqvzbvtnvpefa460xdys2eas5ysiymetadata.json>

```
{
  "description": "ALS::Alice Loves Sea NFT",
  "name": "ALS-467",
  "type": "Collectable",
  "attributes": [
    {"trait_type": "Face", "value": "ten"},
    {"trait_type": "Body", "value": "pinkbody"},
    {"trait_type": "Background", "value": "green"}],
  "image": "ipfs://bafybeig4zvwprykvu25...sb3evje14/N467.png"
}
```

메타정보의 image 항목을 보면 함께 업로드된 이미지의 IPFS 정보가 등록된 것을 알 수 있습니다. saveMetadataUri() 함수는 나중에 스마트 컨트랙트에서 토큰을 발행할 때 메타정보의 URI를 필요로 하므로 컨텐츠 해시를 파일에 기록하는 함수입니다.

```
const saveMetadataUri = (uri) => {
  const filename = `meta.txt`;
  fs.writeFileSync(`./${filename}`, uri + "\r\n", { flag: "a+" });
}
```

저장된 파일의 데이터는 다음과 같은 형식이 됩니다. 1=, 2= 등은 토큰 번호를 나타냅니다. 나중에 이 파일을 읽어서 토큰을 발행할 것입니다.

```
1=ipfs://bafyreihczimru3w77tved64uyrob2vc6s5kcnfs73q3a5awlpti36p56qu/metadata.json
2=ipfs://bafyreia6v2knwvzskcbwty2co777whitzfr5ty43xj3zoslzza4g642y6a/metadata.json
3=ipfs://bafyreibyq2x4msbxxsf5e62diy6oknolml2pw7igx3lndvoqvqd4utepe/metadata.json
...
```

파일을 읽는 함수는 다음과 같이 작성할 수 있습니다. 정규식을 사용하여 토큰 번호에 해당하는 라인을 찾은 후에 “번호=” 부분을 제거하면 ipfs://bafyr...utepe/metadata.json 형태의 문자열을 얻을 수 있습니다.

```
const readMetadataUri = async (index) => {
  const buffer = await fs.readFileSync(META_FILE);
  let tokenUri = "";

  let regexp = new RegExp(`(\r?\n)?${index}=(.* )\\metadata\\.json`, "g");
  let result = buffer.toString().match(regexp);

  if (result != null) {
    tokenUri = result[0].slice(result[0].indexOf("=")+1);
    //console.log(tokenUri);
  }
  return tokenUri;
}
```

메타정보의 처리를 웹에서 수월하게 하기 위하여 ipfs:// 형태의 URI를 HTTP에서도 접근 가능하도록 공용 게이트웨이 URL로 변환하는 함수를 만들기로 하겠습니다.

나중에 애플리케이션에서 메타정보를 가져올 때 fetch를 사용하게 되는데 이 때 HTTP 요청으로 보내야 하기 때문입니다. nft.storage가 제공하는 toGatewayURL() 함수를 이용하면 쉽게 변환할 수 있습니다.

```

const fs = require("fs");
const { toGatewayURL } = require("nft.storage");
const META_FILE = "./meta.href.txt";

const convertGatewayUrl = async (index) => {
    const buffer = await fs.readFileSync(META_FILE);
    let tokenUri = "";

    let regexp = new RegExp("(\\r?\\n)?^" + index + "=.*\\metadata\\.json", "g");
    let result = buffer.toString().match(regexp);

    if (result != null) {
        tokenUri = result[0].slice(result[0].indexOf("=")+1);
    }
    return `${index}=${toGatewayURL(tokenUri).href}`;
}

for (let k=1; k<=500; k++) {
    convertGatewayUrl(k).then(uri => {
        fs.writeFileSync(META_FILE, uri + "\r\n", { flag: "a+" });
    });
}

```

이렇게 만들어진 파일은 다음과 같이 https://로 접속할 수 있는 URL입니다.

```

1=https://dweb.link/ipfs/bafyreiesf...j26x774bqh7xp4shpy2cah55ccz3bwjzfuky/metadata.json
2=https://dweb.link/ipfs/bafyreibkf...vyhziqercye73f2xqgxguxtcfe5bdakwaxmge/metadata.json
...

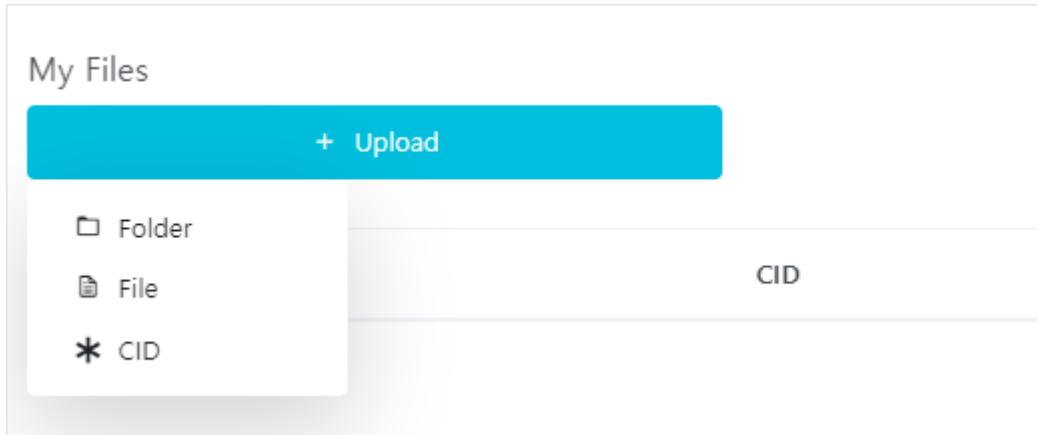
```

***500개의 이미지를 생성하고 메타정보를 업로드하는데 소요되는 시간은 대략 30-40분 정도 걸립니다. IPFS 업로드는 NFT 스토리지의 무료 서비스를 사용하는 것이므로 서비스 품질이 좋지 않을 수도 있습니다.**

● 피나타(Pinata) 이용하기

피나타(<https://pinata.cloud>)는 NFT 스토리지와 유사하게 IPFS 파일 업로드와 관리 그리고 “피닝(pinning)” 서비스를 제공하는 기업입니다. IPFS는 사실상 무료 공용 서버나 마찬가지라서 파일을 다운로드하려면 해당 파일들이 서버에서 유지되고 호스팅될 수 있어야 합니다. 이것을 “피닝”이라고 합니다. IPFS에 업로드된 후에 일정 기간동안 파일요청이 없으면 “garbage collection”에 의해 더 이상 그 파일이 가용하지 않게 될 수 있기 때문에 피닝이 필요합니다.

앞서 NFT 이미지와 메타정보를 NFT 스토리지 서비스를 이용하여 업로드했는데 피나타를 통해서 동일한 작업을 수행할 수 있습니다. 우선 피나타에 무료 계정을 만들어야 합니다. 피나타 사이트에 접속하여 로그인을 하면 파일 관리자를 통해 파일을 업로드할 수 있습니다.

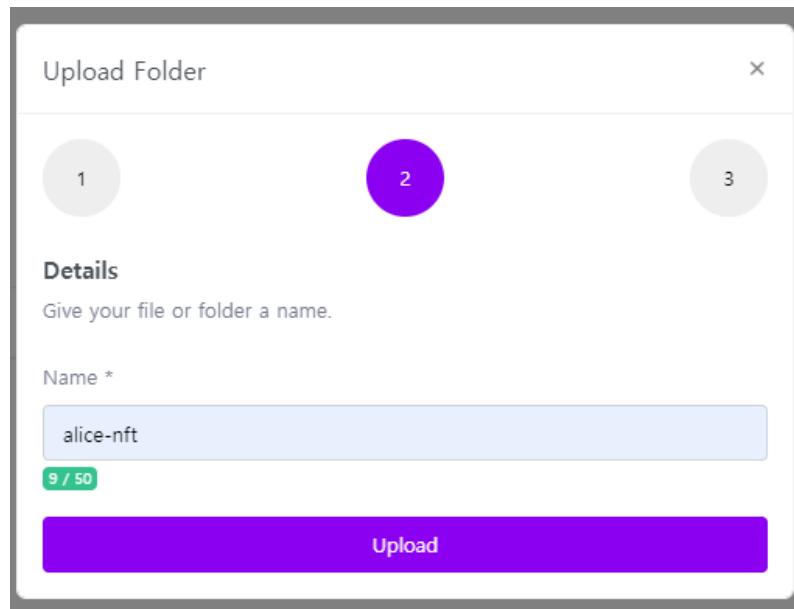


피나타는 개별 파일을 업로드하거나 디렉토리를 지정하여 업로드할 수 있는 기능을 제공합니다. 디렉토리를 업로드하는 경우 디렉토리 내의 모든 파일들이 순차적으로 올라가는데, IPFS의 컨텐츠 위치를 나타내는 대표 CID 해시 값이 부여되고 파일명이 그 뒤에 붙게 됩니다.

예를 들어 앞서 만들어진 500개의 이미지 파일들을 디렉토리로 업로드하면 다음과 같은 URL 형식이 생성됩니다.

```
https://gateway.pinata.cloud/ipfs/QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad/N001.png
https://gateway.pinata.cloud/ipfs/QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad/N002.png
https://gateway.pinata.cloud/ipfs/QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad/N003.png
...
https://gateway.pinata.cloud/ipfs/QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad/N499.png
https://gateway.pinata.cloud/ipfs/QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad/N500.png
```

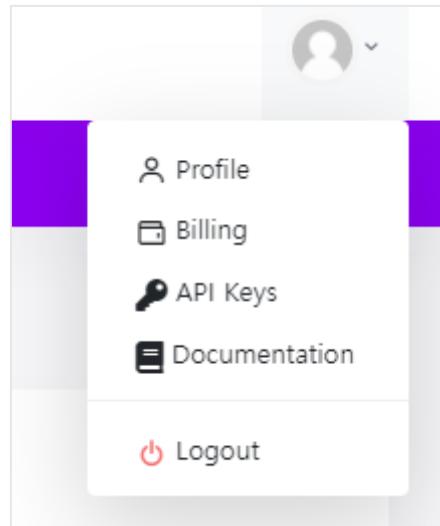
Upload 버튼을 클릭하고 Folder를 선택한 다음 500개의 이미지가 있는 디렉토리를 지정하면 되겠습니다.



파일의 개수와 크기에 따라 업로드하는 시간이 오래 걸릴 수 있습니다. 디렉토리 내의 파일들을 모두 업로드하면 대표 CID 값을 얻을 수 있습니다.

Name	CID
alice-nft 	QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad 

이렇게 파일을 업로드한 후에 이 정보를 ALS 토큰의 메타정보를 만들 때 나오는 image 필드에 넣어 주면 되겠습니다. 앞서 작성한 `create.js`를 수정하여 피나타용 메타정보 업로드를 작성해보도록 하겠습니다. 우선 피나타에서 API 키를 발급받습니다. API 키 발급 메뉴는 우측 상단의 사용자 아이콘을 클릭하면 됩니다.



API Keys 메뉴를 클릭하면 다음과 같은 옵션들을 선택할 수 있습니다.

View Key Info

Admin Admin Keys have access to all endpoints and account settings

API Endpoint Access Pinning hashMetadata hashPinPolicy pinByHash pinFileToIPFS pinJobs pinJSONToIPFS unpin Data pinList userPinnedDataTotal Pinning Services API

Limit Max Uses When enabled, you can set the maximum number of times a key can be used

Key Name

Done

여기서는 “pinJSONToIPFS”를 선택합니다. API 키를 발급받으면 API를 호출할 때 사용해야 하는 키 값들이 표시됩니다. 이 키 값은 나중에 다시 조회할 수 없으므로 복사하여 따로 저장해두는 것이 좋습니다. 여기서는 API Key와 API Secret Key만 사용합니다.

```
API Key: baaec6cd5079dea84322  
API Secret: 90f4...a038  
JWT: eyJhbGciOiJIU...GNOrFGSbQHmuKQ0Jh0
```

피나타 API를 호출하기 위해서는 피나타 SDK를 설치해야 합니다.

```
yarn add @pinata/sdk
```

이제 `ceate.js`를 수정합니다. 피나타 SDK를 사용할 때는 위에서 발급받은 API 키와 Secret API 키 두 개가 필요합니다. `IPFS_URL`은 피나타의 IPFS 게이트웨이 URL이고 `IPFS_IMAGE_HASH`는 NFT 이미지를 업로드한 후 나온 CID 해시 값이 되겠습니다.

```
const pinataSDK = require('@pinata/sdk');

const pinata = pinataSDK('baaec6cd5079dea84322', '90f4...a038');
const IPFS_URL = "https://gateway.pinata.cloud/ipfs";
const IPFS_IMAGE_HASH = "QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad";
```

`create.js`에서 수정해야 하는 부분은 `uploadMetaData()` 함수입니다. 피나타 사이트를 통해 미리 NFT 이미지를 모두 업로드하고 대표 해시 값을 얻었으므로 메타정보의 `image` 필드에 넣는 값은 다음과 같이 바꿀 수 있습니다. 끝에 파일명만 해당 NFT의 이미지로 치환하면 됩니다.

```
const filename = `N${i.toString().padStart(3,0)}`;
metadata.image = `${IPFS_URL}/${IPFS_IMAGE_HASH}/${filename}.png`;
```

피나타에 메타정보 JSON 파일을 업로드하는 것은 `pinJSONToIPFS()`를 호출하면 됩니다. `options`에 전달되는 `pinataMetadata`는 NFT의 메타정보가 아니라 피나타 사이트의 파일 목록에 표시되는 정보들입니다.

```
const options = {
  pinataMetadata: {name: "alice-nft-meta"},
  pinataOptions: {
    cidVersion: 0
  }
}
```

```

        }
    };

    try {
        const result = await pinata.pinJSONToIPFS(metadata, options);
        //console.log(result);

        saveMetadataUri(`#${i}=${IPFS_URL}/${result.IpfsHash}`);
    } catch (err) {
        console.log(err);
    }
}

```

리턴되는 값에 IpfsHash 값이 포함되어 있으므로 메타정보 파일을 만들 때 이 값을 피나타 게이트웨이 URL과 결합하여 저장하면 되겠습니다. 생성된 meta.txt는 다음과 같은 형태가 됩니다.

```

1=https://gateway.pinata.cloud/ipfs/QmPyE52gz5XP7hxNvRPzmFFTqk72KiB9zgDUGJUxUKvikG
2=https://gateway.pinata.cloud/ipfs/QmSznZ41C5nzttdup1HazQea4MH6CXh1bHzj1GXhc6r9Ab
3=https://gateway.pinata.cloud/ipfs/QmPB9en6DgXjdBH461S7jXBXcLvatrjRRDZRfzrGxoaN0R
...

```

피나타를 사용하여 메타정보를 생성하는 createPinata.js의 전체 소스는 아래와 같습니다.

```

const fs = require("fs");
const { createCanvas, loadImage } = require("canvas");
const { face, body, background } = require("./traits.js");
const { saveMetadataUri } = require("./file.js");
const pinataSDK = require('@pinata/sdk');

const pinata = pinataSDK('baaec6cd5079dea84322', '90f...a038');
const IPFS_URL = "https://gateway.pinata.cloud/ipfs";
const IPFS_IMAGE_HASH = "QmV8iZsU1XxdrAbkW81H3snAzwGZ1H2XUw5CwfJNi2pTad";

const canvas = createCanvas(500, 500);
const ctx = canvas.getContext('2d');

const FILE_PATH = "./images";

const getAttributes = (v, k) => {

    let attributes = {};
    let trait_type = "";
    let value = "";

    switch (k) {
        case 0:
            trait_type = "Face";
            value = face[v-1].name;
            break;
        case 1:
            trait_type = "Body";
            value = body[v-1].name;
    }

    attributes[k] = {
        trait_type,
        value
    };
}

fs.writeFileSync(FILE_PATH, JSON.stringify(getAttributes));

```

```

        break;
    case 2:
        trait_type = "Background";
        value = background[v-1].name;
        break;
    default:
        trait_type = "";
        value = "";
    }

    attributes.trait_type = trait_type;
    attributes.value = value;

    return attributes;
}

const saveImage = (canvas, index) => {
    const filename = `N${index.toString().padStart(3,0)}`;
    fs.writeFileSync(`.${FILE_PATH}/_Final/${filename}.png`, canvas.toBuffer("image/png"));
    //console.log(filename);
};

const create = async (t, i) => {

    const face = await loadImage(`.${FILE_PATH}/Face/${t[0]}.png`);
    const body = await loadImage(`.${FILE_PATH}/Body/${t[1]}.png`);
    const background = await loadImage(`.${FILE_PATH}/Background/${t[2]}.png`);

    await ctx.drawImage(background, 0, 0, 500, 500);
    await ctx.drawImage(body, 0, 0, 500, 500);
    await ctx.drawImage(face, 0, 0, 500, 500);

    saveImage(canvas, i+1);

    await uploadMetaData(t, i+1);

};

const uploadMetaData = async (t, i) => {

    let metadata = {
        description: "ALS::Alice Loves Sea NFT",
        name: `ALS-${i}`,
        type: "Collectable",
        image: "https://",
        attributes: [],
    };

    for (let k=0; k<3; k++) {
        metadata.attributes.push(getAttributes(t[k], k));
    }
    const filename = `N${i.toString().padStart(3,0)}`;
    metadata.image = `${IPFS_URL}/${IPFS_IMAGE_HASH}/${filename}.png`;

    const options = {
        pinataMetadata: {name: "alice-nft-meta"},
        pinataOptions: {
            cidVersion: 0
        }
    };
}

```

```

try {
    const result = await pinata.pinJSONToIPFS(metadata, options);
    //console.log(result);

    saveMetadataUri(`#${i}=${IPFS_URL}/${result.IpfsHash}`);
} catch (err) {
    console.log(err);
}
}

module.exports = {
    createPinata: create
}

```

메타정보 파일을 읽는 함수는 다음과 같이 수정하면 되겠습니다.

```

const readMetadataPinataUri = async (index) => {

    const buffer = await fs.readFileSync(META_FILE);
    let tokenUri = "";

    let regexp = new RegExp(`(\r?\n)?${index}=(.*)`, "g");
    let result = buffer.toString().match(regexp);

    if (result != null) {
        tokenUri = result[0].slice(result[0].indexOf("=")+1);
    }
    return tokenUri;
}

```

6.4. 스마트 컨트랙트

6.4.1. ERC-721 컨트랙트

ERC-721 표준 컨트랙트는 오픈제펠린을 사용하면 비교적 쉽게 작성할 수 있습니다. 4장에서 이미 자세히 살펴보았기 때문에 여기서는 약간의 수정만 하도록 하겠습니다.

우선 Dapp 개발을 위한 프로젝트 디렉토리 alice를 새로 생성합니다. 이 디렉토리에는 컨트랙트와 나중에 만들 리액트 애플리케이션 프로젝트가 함께 존재하게 될 것입니다.

컨트랙트 개발도구는 하드햇을 사용하겠습니다. alice 디렉토리로 이동하여 프로젝트를 초기화하고 하드햇과 오픈제펠린 컨트랙트를 설치합니다(5장을 참고하세요).

```

> md alice
> cd alice

```

```
> yarn init -y
> yarn add hardhat --dev
> yarn add @openzeppelin/contracts --dev
```

설치된 오픈제펠린 라이브러리의 버전을 확인해봅니다. 4.3.3으로 표시됩니다.

```
> yarn list --pattern openzeppelin*
yarn list v1.22.5
  └─ @openzeppelin/contracts@4.3.3
Done in 0.30s.
```

하드햇 설정은 5장에서 설명한대로 Create a basic sample project를 선택하고 필요한 패키지들을 추가적으로 설치합니다. 그리고 테스트넷에 배포할 예정이므로 미리 networks에 Rinkeby를 추가하도록 합니다. Rinkeby 접속은 알케미를 이용할 것입니다.

컨트랙트는 오픈제펠린의 ERC-721 구현체 중 하나인 ERC721URIStorage를 사용하기로 합니다. 이 컨트랙트는 필수 ERC-721 표준을 모두 구현하고 있습니다. 오픈제펠린이 이미 컨트랙트 검증까지 마쳤을 것이므로(개발팀을 믿을 수 밖에?) 문제가 없을 것입니다.

ALSnft.sol을 다음과 같이 작성할 수 있습니다. 이미 대부분의 기능이 구현된 ERC721URIStorage을 상속받으므로 코드가 길지 않습니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract ALSnft is ERC721URIStorage {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIds;

    constructor() ERC721("Alice Loves Sea NFT", "ALS") {}

    function mint(address toAddr, string memory tokenURI) public returns (uint256) {
        _tokenIds.increment();
        uint256 newItemId = _tokenIds.current();
        _mint(toAddr, newItemId);
        _setTokenURI(newItemId, tokenURI);

        return newItemId;
    }
}
```

이 컨트랙트는 오픈제펠린이 제공하는 Counters 라이브러리를 이용하여 컨트랙트 내에서 토큰 번호를 순번으로 생성합니다. 그런데 여기서는 토큰 번호를 외부에서 전달하기로 하겠습니다.

mint() 함수의 인자를 수정하여 NFT를 받을 계정과 발행할 토큰 번호 그리고 메타정보 URI를 파라미터로 전달하도록 하겠습니다. 또 토큰 발행은 컨트랙트 배포자만 하기로 합니다. modifier onlyOwner를 추가해보겠습니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";

contract ALSnft is ERC721URIStorage {

    address owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() ERC721("Alice Loves Sea NFT", "ALS") {
        owner = msg.sender;
    }

    function mint(address toAddr, uint256 tokenId, string memory tokenURI)
        public onlyOwner returns (uint256) {

        uint256 newItemId = tokenId;
        _mint(toAddr, newItemId);
        _setTokenURI(newItemId, tokenURI);

        return newItemId;
    }
}
```

ALSnft 컨트랙트에서 외부로 노출된 함수들은 다음과 같습니다.

함수명	구분	ERC-721	ERC-165
approve	트랜잭션	O	-
safeTransferFrom	트랜잭션	O	-
setApproveForAll	트랜잭션	O	-
transferFrom	트랜잭션	O	-
balanceOf	조회	O	-
getApproved	조회	O	-
isApproveForAll	조회	O	-

name	조회	O	-
ownerOf	조회	O	-
supportsInterface	조회	-	O
tokenURI	조회	O	
symbol	조회	O	
mint	트랜잭션	-	-

ERC-165와 ERC-721 표준에 있는 함수가 모두 구현되어 있고 토큰 발행 함수 mint도 작성했습니다. mint는 다시 internal 함수인 _mint를 호출합니다.

토큰 번호는 애플리케이션에서 전달해주기로 합니다. 이 번호는 앞에서 생성한 ALS의 토큰 번호(1~500)와 매칭시키면 되겠습니다. NFT의 토큰 번호는 그 번호 자체에 특별한 의미가 있는 것은 아니고 컨트랙트 내에서 유일하게 식별되기만 하면 됩니다.

6.4.2. 단위 테스트

단위 테스트는 컨트랙트의 기본적인 입출력이 올바른지를 확인하는 단계입니다. 물론 개발자가 작성한 컨트랙트를 개발자 스스로 테스트하는 것이므로 모두 통과했다고 해서 컨트랙트가 완전무결하다고 할 수는 없습니다. 하지만 단위 테스트는 기본적인 기능을 확인하는 단계이고 너무 당연하지만 미처 생각하지 못했던 오류들을 발견할 수도 있으므로 반드시 거쳐야 하는 단계라고 할 수 있습니다.

앞에서 작성한 ALSnft 컨트랙트의 테스트 스크립트를 작성해보도록 하겠습니다. 우선 테스트 케이스를 나열해보겠습니다.

- ✓ ERC-721과 ERC-165 인터페이스를 구현했는지 검사
- ✓ 토큰 발행 후 해당 토큰의 소유자가 맞는지 검사
- ✓ 발행되지 않은 토큰을 소유한 계정이 있는지 검사
- ✓ 토큰 소유자의 토큰 개수가 맞는지 검사
- ✓ 토큰 발행 권한이 없는 사용자가 토큰을 발행했을 때 revert가 되는지 검사
- ✓ 토큰을 다른 계정에게 전송했을 때 수취인 계정이 토큰을 받았는지 검사
- ✓ 전송 후 정해진 이벤트가 발생하는지 검사

- ✓ 특정 토큰에 대해 전송 권한을 위임할 수 있는지 검사
- ✓ 위임 계정으로 토큰을 전송할 수 있는지 검사
- ✓ 위임을 철회했을 때 전송 권한이 없는지 검사

이외에도 다양한 테스트 케이스를 만들 수 있습니다. 5장에서 설명한 와플을 사용하겠습니다. 먼저 컴파일 결과물 ALSnft.json을 가져옵니다.

```
const hre = require("hardhat");
const { deployContract, provider } = hre.waffle;
const {expect, assert} = require("chai");

const ALSnft = require("../artifacts/contracts/ALSnft.sol/ALSnft.json");

describe("ALS NFT Test", async () => {
  //Test case
});
```

테스트를 할 때는 여러 계정이 필요합니다. 와플에서는 다음과 같이 테스트 계정을 가져올 수 있습니다. 테스트에 사용할 4개의 계정에 그 역할을 알 수 있도록 각각 이름을 줄 수 있습니다. 첫 번째 계정은 보통 배포 계정으로 사용하기 때문에 "deployer"로 주었습니다. "approved"는 위임 테스트를 할 때 사용할 계정입니다.

```
const [deployer, james, mary, approved] = provider.getWallets();
```

테스트 케이스 실행 전에 컨트랙트를 배포하는 코드를 작성합니다.

```
before(async () => {
  this.instance = await deployContract(deployer, ALSnft);
});
```

테스트 케이스 몇 가지를 살펴보겠습니다. 첫 번째는 인터페이스 구현 여부를 검사하는 것입니다. ERC-165 인터페이스를 구현하기로 표준에서 정하고 있으므로 최소한 2개의 인터페이스가 구현되어

있어야 합니다. 인터페이스를 확인하는 함수는 supportsInterface입니다.

```
const ERC721IFID = "0x80ac58cd";
const ERC165IFID = "0x01ffc9a7";

it ("Should implement ERC-721 and ERC-165", async () => {
    const v1 = await this.instance.supportsInterface(ERC721IFID);
    const v2 = await this.instance.supportsInterface(ERC165IFID);

    assert.isOk(v1&&v2);
});
```

다음에는 토큰을 발행해보고 지정된 계정에게 전송되는지를 확인하는 테스트입니다. mint() 호출 후에 ownerOf()로 소유자를 확인합니다. 토큰 번호는 차례대로 1,2,3이 생성됩니다.

```
const tokenURI1 = "ipfs://...somu/metadata.json";
const tokenURI2 = "ipfs://...3u6e/metadata.json";
const tokenURI3 = "ipfs://...x7xi/metadata.json";

it ("Should mint 3 ALS", async () => {

    await this.instance.mint(deployer.address, 1, tokenURI1);
    await this.instance.mint(deployer.address, 2, tokenURI2);
    await this.instance.mint(deployer.address, 3, tokenURI3);

    const v = await this.instance.ownerOf(1);
    assert.equal(v, deployer.address, "Not the same address");
});
```

배포 계정만 토큰을 발행할 수 있도록 작성했으므로 다른 계정이 발행을 시도한다면 revert가 되어야 합니다. james라는 계정으로 발행을 해보고 revert가 되는지 검사합니다.

```
it ("Should revert when minting by anyone", async () => {
    await expect(this.instance.connect(james).mint(james.address, 1, tokenURI1))
        .to.be.reverted;
});
```

다른 계정으로 토큰을 전송해보겠습니다. transferFrom()을 호출하여 mary에게 1번 토큰을 전송합니다. 여기서 확인하는 것은 1번 토큰의 소유자가 mary인지, 또 보유한 토큰의 개수가 맞는지를 검사할 수 있습니다. mary는 1개, deployer는 2개가 남았을 것입니다.

```
it ("Should transfer the token to new owner", async () => {

    await this.instance.transferFrom(deployer.address, mary.address, 1);
    const v = await this.instance.ownerOf(1);
```

```

const b1 = await this.instance.balanceOf(mary.address);
const b2 = await this.instance.balanceOf(deployer.address);

assert(v === mary.address, "Failed to transfer");
assert(b1.eq(hre.ethers.BigNumber.from("1")));
assert(b2.eq(hre.ethers.BigNumber.from("2")));
});

```

balanceOf()는 계정이 가진 토큰 수량을 리턴합니다. 그런데 보통 숫자는 BigNumber라는 자바스크립트 객체로 전달됩니다. Ethers.js에서는 BigNumber 연산을 위한 유ти리티를 제공하므로 그것을 사용해서 비교를 하면 되겠습니다(==으로 비교하면 실패할 수 있습니다).

전송 후에 발생하는 이벤트는 다음과 같이 테스트할 수 있겠습니다.

```

it ("Should emit Transfer event when transfer", async () => {
    await expect(this.instance.transferFrom(deployer.address, mary.address, 1))
        .to.emit(this.instance, "Transfer");
});

```

만약에 토큰 소유자가 아닌 계정이 임의로 토큰을 전송한다면 어떻게 될까요? 당연히 revert되어야 합니다.

```

it ("Should revert if invalid token owner", async () => {
    await expect(this.instance.connect(james).transferFrom(deployer.address, mary.address, 1))
        .to.be.reverted
});

```

approve 계정으로 2번 토큰을 위임해보겠습니다. 토큰이 위임되었는지 여부는 getApproved()로 확인 할 수 있습니다.

```

it ("Can approve the ownership to operator", async () => {
    await this.instance.approve(approved.address, 2);
    const v = await this.instance.getApproved(2);
    assert(v, approved.address, "Wrong approved operator");
});

```

approved 계정은 위임을 받은 토큰을 전송할 수 있을까요? mary에게 2번 토큰을 전송하고 소유자가 mary인지 확인합니다.

```

it ("Can transfer the token by the operator", async () => {

```

```

        await this.instance.connect(approved).transferFrom(deployer.address, mary.address, 2);
        const v = await this.instance.ownerOf(2);
        assert(v === mary.address, "Failed to transfer");
    });

```

다음은 safeTransferFrom()을 테스트해보겠습니다. ERC-721 표준에서 safeTransferFrom()은 두 개가 존재하는데 차이점은 전달인자인 bytes 타입의 데이터 유무입니다. Ethers.js 5.x 버전에서는 함수 정의의 모호함 때문에 호출방식을 다르게 해주어야 합니다.

예를 들어서 소유자가 아닌 계정으로 safeTransferFrom()을 실행했을 때 revert되는 것을 테스트한다면 아래와 같이 함수를 호출할 때 시그너처를 명시하여 호출합니다.

```

it ("Should revert transfer token to contract", async () => {
    await expect(
        this.instance.connect(james)[ 'safeTransferFrom(address,address,uint256)' ](mary.address,
        james.address, 1))
        .to.be.reverted;
});

```

전체 테스트 스크립트는 다음과 같습니다. 테스트 케이스에 it.skip을 하면 해당 케이스는 수행되지 않으므로 상황에 따라 적절하게 활용하면 되겠습니다.

```

const hre = require("hardhat");
const { deployContract, provider } = hre.waffle;
const {expect, assert} = require("chai");

const ALSnft = require("../artifacts/contracts/ALSnft.sol/ALSnft.json");

const ERC721IFID = "0x80ac58cd";
const ERC165IFID = "0x01ffc9a7";

const tokenURI1 = "ipfs://bafy...somu/metadata.json";
const tokenURI2 = "ipfs://bafy...3u6e/metadata.json";
const tokenURI3 = "ipfs://bafy...x7xi/metadata.json";

describe("ALS NFT Test", async () => {
    const [deployer, james, mary, approved] = provider.getWallets();

    before(async () => {
        this.instance = await deployContract(deployer, ALSnft);
    });

    it ("Should implement ERC-721 and ERC-165", async () => {
        const v1 = await this.instance.supportsInterface(ERC721IFID);
        const v2 = await this.instance.supportsInterface(ERC165IFID);

        assert.isOk(v1&&v2);
    });
});

```

```

it ("Should mint the 1st ALS", async () => {
    await this.instance.mint(deployer.address, 1, tokenURI1);
    await this.instance.mint(deployer.address, 2, tokenURI2);
    await this.instance.mint(deployer.address, 3, tokenURI3);

    const v = await this.instance.ownerOf(1);
    assert.equal(v, deployer.address, "Not the same address");
});

it ("Should revert when token ID does not exist", async () => {
    await expect(this.instance.ownerOf(100))
        .to.be.reverted;
});

it ("Should have the balance", async () => {
    const v = await this.instance.balanceOf(deployer.address);
    expect(v).to.equal(3);
});

it.skip ("Should have the right token URI", async () => {
    const v = await this.instance.tokenURI(1);
    assert.equal(v, tokenURI1, "Wrong TokenURI");
});

it ("Should revert when minting by anyone", async () => {
    await expect(this.instance.connect(james).mint(james.address, 1, tokenURI1))
        .to.be.reverted;
});

context("Transfer Test", async () => {

    it ("Should revert if the recipient is zero address", async () => {
        await expect(this.instance.connect(james)
            .transferFrom(deployer.address, "0x0", 1))
            .to.be.reverted
    });

    it ("Should revert if invalid token owner", async () => {
        await expect(this.instance.connect(james)
            .transferFrom(deployer.address, mary.address, 1))
            .to.be.reverted
    });

    it ("Should transfer the token to new owner", async () => {

        await this.instance.transferFrom(deployer.address, mary.address, 1);
        const v = await this.instance.ownerOf(1);

        const b1 = await this.instance.balanceOf(mary.address);
        const b2 = await this.instance.balanceOf(deployer.address);

        assert(v === mary.address, "Failed to transfer");
        assert(b1.eq(hre.ethers.BigNumber.from("1")));
        assert(b2.eq(hre.ethers.BigNumber.from("2")));
    });

});

context("Approve Test", async () => {

```

```
it ("Can approve the ownership to operator", async () => {
    await this.instance.approve(approved.address, 2);
    const v = await this.instance.getApproved(2);
    assert(v, approved.address, "Wrong approved operator");
});

it ("Can transfer the token by the operator", async () => {
    await this.instance.connect(approved).transferFrom(deployer.address, mary.address, 2);
    const v = await this.instance.ownerOf(2);
    assert(v === mary.address, "Failed to transfer");
});

it ("Can revoke the ownership from approved", async () => {
    await this.instance.setApprovalForAll(approved.address, false);
    const v = await this.instance.isApprovedForAll(deployer.address, approved.address)
    expect(v).to.be.false;
});

});
```

6.5. 리액트 애플리케이션

자바스크립트 프론트엔드 라이브러리는 여러 개가 존재합니다. 모두 나름의 장점들을 가지고 있고 개발자들의 선호도가 다르기 때문에 선택의 문제라고 할 수 있겠습니다.

2020년 “스택오버플로우(stackoverflow)” 개발자 설문에 의하면 리액트는 두 번째로 많이 쓰는 자바스크립트 기술로 나타났습니다(1위는 jQuery).

<https://insights.stackoverflow.com/survey/2020#technology-web-frameworks>

리액트는 뷰를 담당하는 라이브러리, 즉 화면을 만드는 자바스크립트 라이브러리입니다. 현재 대부분의 애플리케이션들이 웹브라우저를 통해서 서비스되기 때문에 웹 기반의 프론트엔드는 사실상 표준이 된 것이나 다름없습니다. 여기서는 리액트를 사용하여 ALS 토큰을 관리하는 Dapp을 만들어볼 것입니다(하지만 리액트 사용법을 자세히 다루지는 않습니다).

우선 리액트 애플리케이션 프로젝트를 생성합니다. alice 디렉토리 아래에 애플리케이션 프로젝트도 함께 배치하겠습니다.

alice 아래에서 npx create-react-app을 실행하여 프로젝트 app를 생성합니다. create-react-app은 리액트 애플리케이션을 쉽게 개발할 수 있도록 해주는 프로젝트 생성 도구입니다.

```
alice> npx create-react-app app
alice> dir app

Mode           LastWriteTime      Length  Name
----           -----          -----   --
d----  2021-11-21  오전 9:45            node_modules
d----  2021-11-21  오전 9:44            public
d----  2021-11-21  오전 9:44            src
-a---  2021-05-31  오후 8:45        310  .gitignore
-a---  2021-11-21  오전 9:45        807  package.json
-a---  2021-11-21  오전 9:44       3362  README.md
-a---  2021-11-21  오전 9:45     510354  yarn.lock
```

app는 순수하게 리액트 애플리케이션 프로젝트로 생성되었습니다. app 프론트 애플리케이션은 컨트랙트를 컴파일할 때 생성된 ABI(Application Binary Interface)를 통해 컨트랙트와 인터페이스하게 될 것입니다.

5장에서 hardhat-deploy를 설명할 때 --export 옵션이 있었습니다. 이 옵션이 바로 애플리케이션이 필요한 위치에 ABI를 생성해주는 기능을 합니다. 일단 화면의 구조를 잡은 다음에 ABI 파일을 생성

하도록 하겠습니다.

리액트 애플리케이션의 모든 소스파일은 app/src 아래에서 작성합니다. 기본적으로 단일 페이지 애플리케이션(Single Page Application, SPA)으로 구성되기 때문에 HTML 파일은 하나이고 자바스크립트 JS 파일들이 대부분입니다. JS 파일들이 각각 다른 뷰를 보여주면서 마치 여러 개의 페이지가 열리는 것처럼 보일 뿐입니다.

리액트 프로그래밍은 이렇게 뷰를 만드는 함수(컴포넌트라는 용어를 쓰기도 합니다)를 작성하는 것이라고 볼 수 있습니다. 예를 들어 app/src 아래에 이미 생성된 App.js를 열어서 보면 다음과 같습니다.

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

이것을 단순화하면 이런 구조가 됩니다.

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      ...
    </div>
  );
}

export default App;
```

App()라는 함수는 결국 리턴 값으로 화면 요소인 HTML 태그 <div></div>를 리턴합니다. 조금 정확히 말하면 HTML과 비슷하게 보이지만 컴포넌트들을 HTML 태그처럼 쓸 수 있도록 확장된 "JSX"라는 것을 리턴하는 것입니다.

App() 함수는 index.js에서 참조되는데, index.js는 ReactDOM.render()라는 함수로 index.html의 특정 위치에 App()에서 리턴하는 JSX를 그려주게 됩니다. 여기서는 document.getElementById('root')라는 위치에 App() 컴포넌트의 리턴 값을 렌더링합니다.

```
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
>;
```

App.js처럼 컴포넌트 JS 파일들이 index.js으로 모이고 "번들링(bundling)"이라는 과정을 거쳐 하나의 SPA가 됩니다. SPA는 서버에 배포되고 웹브라우저를 통해 서비스되는 것입니다.

*이제부터 화면을 개발하는데 필요한 패키지 설치와 소스 파일 작성은 *alice/app* 디렉토리에서 수행합니다.

6.5.1. 상단 메뉴

화면 UI를 개발하는 것은 시간과 비용이 소요되는 일 중 하나입니다. 웹의 특성을 생각해야 하고, 애플리케이션의 컨셉, 사용자 경험(UX) 등을 분석하는 과정이 필요합니다. 특히 Dapp은 일반 애플리케이션과는 다른 지갑 소프트웨어와 결합되는 부분이 있기 때문에 소위 말하는 "온보딩(onboarding)"을 어떻게 할 것인지도 중요합니다.

이 글의 주제를 벗어나지 않는 범위 내에서 화면을 구성하기로 하겠습니다. 그래서 사용하기 쉬운 UI 라이브러리를 활용합니다. MUI(<https://mui.com/>)는 이전에 "Material UI"로 불리던 라이브러리였습니다. 컴포넌트가 잘 되어 있어서 그대로 쓰기 좋습니다.

```
yarn add @mui/material @emotion/react @emotion/styled
```

간단하게 MUI의 <Box/> 컴포넌트를 사용해보겠습니다. App.js를 다음과 같이 수정합니다.

```

import { Fragment } from "react";
import Box from '@mui/material/Box';

function App() {
  return (
    <Fragment>
      <Box sx={{padding: "10px 0px 10px 10px",
        width: "600px",
        color: "white",
        background: "blue" }}>
        Hello, Blockchain!
      </Box>
    </Fragment>
  );
}

export default App;

```

create-react-app으로 프로젝트를 생성하면 기본적인 패키지들이 설치되는데 개발 서버도 포함되어 있습니다. 위에서 수정한 결과를 확인하기 위해 개발 서버를 실행합니다.

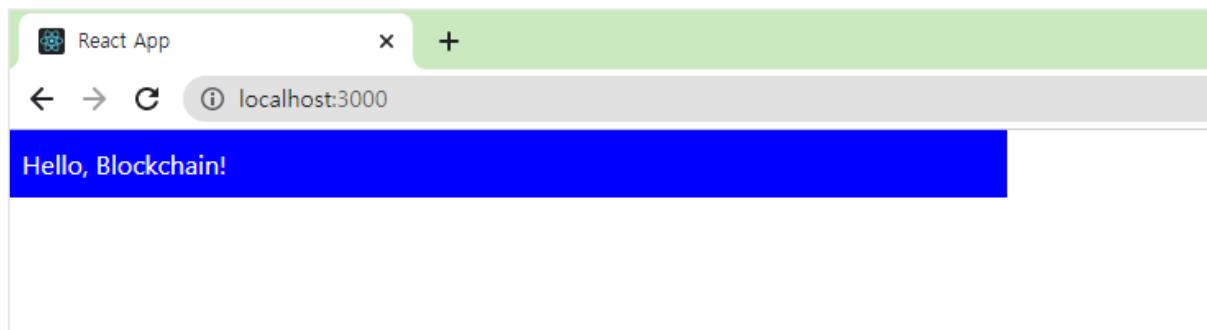
```

yarn start

yarn run v1.22.5
$ react-scripts start
i [wds]: Project is running at http://192.168.0.10/
i [wds]: webpack output is served from
i [wds]: Content not from webpack is served from C:\Users\song\bookapp\alice\app\public
i [wds]: 404s will fallback to /
Starting the development server...

```

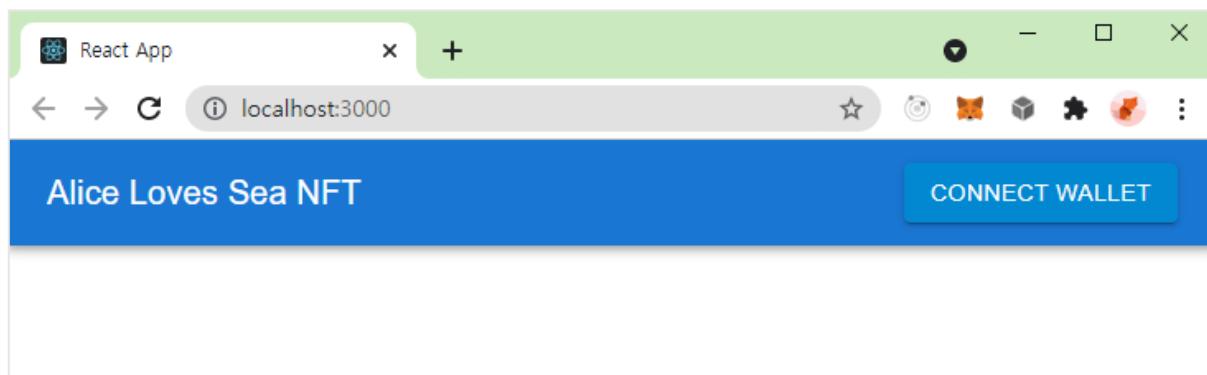
웹브라우저가 실행되면서 localshot:3000으로 자동 접속되고 다음과 같은 화면이 열립니다. 아래 그림과 같이 <Box/> 컴포넌트가 렌더링 됩니다. 이렇게 MUI에서 제공하는 다양한 UI 컴포넌트를 이용하여 화면을 구성할 수 있습니다.



이제부터 화면 요소들을 하나씩 만들어보겠습니다. 상단 메뉴로 <AppBar/> 컴포넌트를 사용하기로 합니다. 다음과 같이 <Fragment/> 사이에 <AppBar/> 컴포넌트를 배치합니다.

```
<AppBar position="static">
  <Toolbar>
    <Typography variant="h6" sx={{ flexGrow: 1 }}>
      Alice Loves Sea NFT
    </Typography>
    <Button color="info" variant="contained">connect wallet</Button>
  </Toolbar>
</AppBar>
```

<AppBar/> 안에 <Toolbar/> 컴포넌트를 사용하여 간단한 메뉴를 구성할 수 있습니다. 여기서는 타이틀과 지갑 연결 버튼을 만들었습니다. 버튼은 <Button/> 컴포넌트를 사용합니다.



바로 아래 주석으로 표시한 /* NFT List */에는 NFT의 목록을 보여줄 계획입니다. 미리 <Box/> 컴포넌트를 추가하여 자리를 만들어 두기로 합니다.

```
<AppBar position="static">
  <Toolbar>
    <Typography variant="h6" sx={{ flexGrow: 1 }}>
      Alice Loves Sea NFT
    </Typography>
    <Button color="info" variant="contained">connect wallet</Button>
  </Toolbar>
</AppBar>
<Box sx={{paddingLeft: "20px", paddingTop: "20px"}}>
  /* NFT List */
</Box>
```

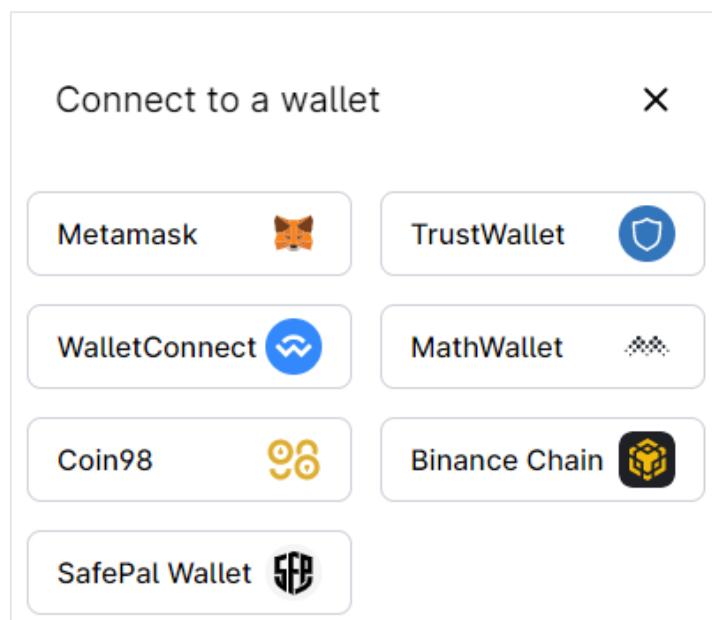
다음에는 "CONNECT WALLET" 기능을 구현하도록 하겠습니다.

6.5.2. 지갑 연결

대부분의 Dapp들은 웹브라우저의 지갑과 연결되어 동작합니다. 대표적인 지갑은 메타마스크입니다. 메타마스크는 웹브라우저 크롬의 플러그인으로, 크롬 웹스토어에서 설치할 수 있습니다. 지갑이라는 것은 전자서명을 하기 위한 소프트웨어이기 때문에 어느 특정 애플리케이션에 종속적인 것이 아닙니다. 사용자들이 한 가지 지갑만 쓸 수 있게 하는 것보다는 여러 지갑 중 하나를 선택하게 하는 것이 바람직합니다.

지갑을 Dapp과 연결시키려면 해당 지갑 개발사가 제공하는 라이브러리를 추가하여 별도의 작업이 필요합니다. 여기서는 다루지 않지만 여러 지갑을 선택할 수 있게 도와주는 라이브러리도 있습니다. "Web3Modal"이라는 패키지를 참고하기 바랍니다.

<https://github.com/Web3Modal/web3modal>



여기서는 메타마스크와 WalletConnect(<https://walletconnect.com/>)라는 것을 사용해보기로 하겠습니다. 사실 WalletConnect는 지갑 그 자체는 아니고 Dapp과 모바일 지갑을 연결시켜주는 오픈 프로토콜입니다.

누구나 WalletConnect 프로토콜을 이용하여 모바일 지갑을 만들 수 있습니다. 또는 기존의 모바일 지갑이 WalletConnect를 지원할 수도 있습니다. 예를 들어 메타마스크 모바일이나 트러스트월렛(TrustWallet)에서 WalletConnect를 사용할 수 있습니다.

"CONNECT WALLET" 버튼을 클릭하면 지갑을 선택할 수 있는 모달(Modal) 팝업을 열기로 하겠습니다

다. 우선 메타마스크와 WalletConnect를 사용하기 위해서 web3-react를 설치합니다. 이 패키지는 리액트 기반의 Dapp을 개발할 때 많이 사용하는 라이브러리입니다. 아래 깃허브를 참고하기 바랍니다.

<https://github.com/NoahZinsmeister/web3-react>

여기서는 지갑 연결을 위한 두 개의 패키지만 설치합니다.

```
yarn add @web3-react/injected-connector @web3-react/walletconnect-connector
```

"injected-connector"은 메타마스크와 같은 웹브라우저 플러그인 지갑을 말하고 "walletconnect-connector"은 WalletConnect를 지원하는 지갑을 말합니다. 또 이더리움 자바스크립트 라이브러리인 Ethers.js를 설치합니다.

```
yarn add ethers
```

먼저 지갑을 선택할 수 있는 팝업으로 WalletModal.js를 작성하겠습니다. MUI에는 <Modal/>이라는 모달 팝업 컴포넌트를 제공합니다.

```
import React from "react";
import {Box, Button, Modal} from "@mui/material";
import { ethers } from "ethers";

const style = {
  position: 'absolute',
  top: '50%',
  left: '50%',
  transform: 'translate(-50%, -50%)',
  width: 320,
  backgroundColor: 'white',
  border: '1px solid #000',
  boxShadow: 24,
  p: 4,
  display: 'flex',
  flexDirection: 'column'
};

function WalletModal(props) {

  const handleConnectMM = async () => {
    //TODO
  }

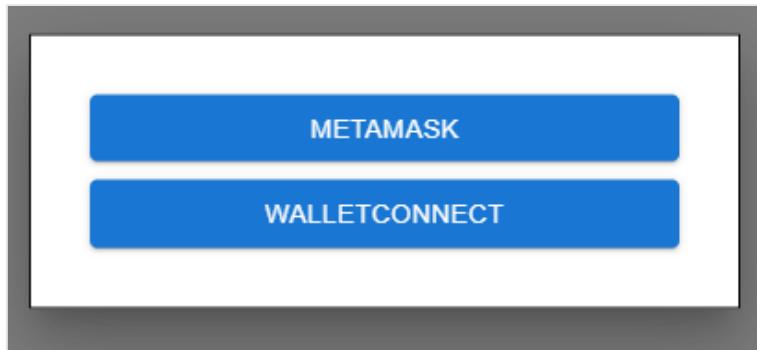
  const handleConnectWC = async () => {
    //TODO
  }
}
```

```

        return (
          <Modal open={props.open} onClose={props.close}>
            <Box sx={style}>
              <Button variant="contained"
                sx={{marginBottom: '10px'}}
                onClick={handleConnectMM}>Metamask</Button>
              <Button variant="contained" onClick={handleConnectWC}>WalletConnect</Button>
            </Box>
          </Modal>
        );
      }

      export default WalletModal;
    
```

상단에 복잡하게 보이는 style은 모달 팝업의 컨텐츠 영역에 적용되는 스타일입니다. WalletModal의 모습은 아래와 같습니다. METAMASK와 WALLETCONNECT 버튼을 상하로 배치합니다.



METAMASK 버튼을 누르면 메타마스크 지갑이 열리고 WALLETCONNECT를 클릭하면 모바일 지갑으로 스캔할 수 있는 QR코드가 표시되도록 할 것입니다.

<Modal/> 컴포넌트의 속성인 open의 값이 true/false 여부에 따라 열리기도(보이기도) 하고 닫히기(숨기도) 합니다. open이라는 속성으로 모달 팝업을 제어하는 것입니다. open이라는 속성을 <WalletModal/> 컴포넌트의 상태로 정의할 수 있습니다.

리액트에서는 컴포넌트의 상태를 정의하여 화면을 제어하는데, 이 과정에서 사용하는 것이 useState()라는 함수입니다. open이라는 상태를 변경하여 모달 팝업을 열거나 닫는 것처럼 보이도록 만드는 것입니다.

open이라는 상태와 그 상태를 업데이트하는 setOpen을 리액트의 useState()를 사용하여 다음과 같이 정의합니다. open의 값은 setOpen으로만 변경할 수 있습니다. 값이 업데이트 되면 해당 함수 컴포넌트가 다시 렌더링 됩니다.

```
const [open, setOpen] = useState(false);
```

이러한 동작원리를 모달 팝업을 추가하면서 조금 더 자세히 알아보겠습니다. useState()를 사용하여 상태를 정의하고 <WalletModal/>을 App.js에 추가합니다. <Box/> 뒤에 넣기로 하겠습니다.

```
import {Fragment, useState} from "react";
import WalletModal from "./modals/WalletModal";

const [open, setOpen] = useState(false);

const handleOpen = () => {
    setOpen(true);
}

const handleClose = () => {
    setOpen(false);
}

<AppBar position="static">
    <Toolbar>
        <Typography variant="h6" sx={{ flexGrow: 1 }}>
            Alice Loves Sea NFT
        </Typography>
        <Button color="info" variant="contained" onClick={handleOpen}>connect wallet</Button>
    </Toolbar>
</AppBar>
<Box sx={{ paddingLeft: "20px", paddingTop: "20px" }}>
</Box>
<WalletModal open={open} close={handleClose}/>
```

WalletModal(props) 함수는 props를 통해 부모 컴포넌트로부터 상태 업데이트에 필요한 정보를 전달 받습니다. 여기서는 open과 close를 받게 됩니다. 즉 모달 팝업을 열고 닫는 것은 부모 컴포넌트인 App.js에 정의된 handleOpen()과 handleClose()에서 하게 되는 것입니다.

handleOpen() 함수는 모달 팝업을 여는 함수입니다. setOpen(true)을 실행하면 open이 true가 되고 다시 <WalletModal/>을 통해서 <Modal/>의 속성 open으로 전달됩니다. open이 true이면 <Modal/> 컴포넌트가 나타나게 됩니다.

<Modal/> 컴포넌트는 닫기 버튼이 없어도 외부 영역을 클릭하면 닫혀야 합니다. 즉 외부를 클릭하는 이벤트가 발생했을 때 open을 false로 변경해야 하는 것입니다. 그래서 onClose라는 속성이 있고 onClose에 handleClose() 함수를 전달해 주면 되겠습니다.

다음은 지갑연결 버튼을 구현해보도록 하겠습니다. 먼저 메타마스크 버튼 onClick 이벤트가 호출하는 handleConnectMM를 작성합니다. 메타마스크는 injected-connector를 사용하므로 @web3-

react/injected-connector를 import 합니다.

```
import { InjectedConnector, UserRejectedRequestError as UserRejectedRequestErrorMM }  
from "@web3-react/injected-connector";
```

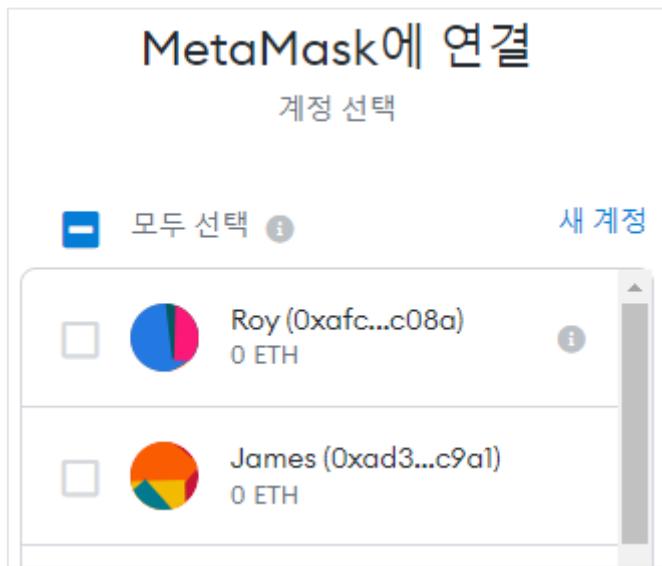
InjectedConnector가 메타마스크 지갑을 연결할 때 사용할 함수입니다. 우선 인스턴스를 생성합니다. WalletModal(props) 함수 밖에서 작성하도록 합니다. 생성 전달인자는 연결할 이더리움 네트워크의 체인 번호입니다. 여기서는 편의상 메인넷 1번과 Rinkeby 4번으로 정합니다(예제에서는 Rinkeby에만 연결합니다).

```
const injectedWeb3Connector = new InjectedConnector({ supportedChainIds: [1,4] });  
  
function WalletModal(props) {  
  ...
```

이제 버튼을 클릭했을 때 메타마스크 지갑을 활성화시키면 됩니다. 활성화는 사용자가 지갑을 여는 것을 말합니다. 그냥 닫았을 경우에는 UserRejectedRequestError 예외가 발생하므로 try-catch를 사용하여 모달 팝업을 닫도록 하겠습니다.

```
const handleConnectMM = async () => {  
  try {  
    await injectedWeb3Connector.activate();  
  } catch (err) {  
    if (err instanceof UserRejectedRequestErrorMM) {  
      console.log("UserRejectedRequest...");  
      props.close();  
    }  
  }  
}
```

처음 Dapp에 연결하면 사용자가 자신의 지갑을 Dapp에 연결할지 결정해야 합니다. 다시 말해서 메타마스크를 Dapp과 연결할지, 또 어느 계정으로 연결할지를 선택하는 과정입니다.



이렇게 Dapp과 메타마스크가 연결이 되고나면 앞으로 Dapp이 사용할 Web3 Provider(5장 참조)를 애플리케이션에서 전역적으로 유지할 필요가 있습니다.

모달 팝업은 지갑 연결 후에 닫히게 되므로 Web3 Provider를 최상위 컴포넌트인 App.js에 유지해주세요. 그래서 App.js에 다음과 같은 상태를 추가합니다.

```
const [web3, setWeb3] = useState({});
```

모달 팝업에서 지갑이 연결되면 Web3 Provider를 생성하여 최상위 컴포넌트인 App.js의 web3에 저장할 것입니다. Web3 Provider를 최상위 컴포넌트가 가지고 있어야 하위의 다른 컴포넌트에 전달해 줄 수 있습니다.

하위 컴포넌트로 상태를 전달하는 방법은 여러 가지가 있습니다. 모달 팝업처럼 props를 통해서 전달할 수도 있는데 그런 경우 하위 컴포넌트가 몇 단계 아래에 존재하면 계속 거쳐서 전달해야 하기 때문에 번거로울 수 있습니다.

다른 방법으로 리액트의 컨텍스트 API(Context API)를 이용할 수도 있습니다. 컨텍스트는 다음과 같이 createContext() 사용하여 생성한 후에 export를 해주면 되겠습니다.

```

import {Fragment, useState, createContext} from "react";

const Context = createContext();
const Provider = Context.Provider;

function App() {
...
}

export default App;

export {
    Context
}

```

컨텍스트 API의 <Provider/>는 전역적인 상태를 하위 컴포넌트에 제공해주는 컴포넌트입니다. <Provider/> 내부에 포함된 모든 컴포넌트들은 전달된 상태를 다른 컴포넌트들을 거치지 않고도 직접 참조할 수 있게 되는 것입니다.

사용법은 간단합니다. 다음과 같이 하위 컴포넌트들을 감싸주면 됩니다.

```

<Provider value={{setWeb3}}>
    <AppBar position="static">
        <Toolbar>
            <Typography variant="h6" sx={{ flexGrow: 1 }}>
                Alice Loves Sea NFT
            </Typography>
            <Button color="info" variant="contained"
                    onClick={handleOpen}>connect wallet</Button>
        </Toolbar>
    </AppBar>
    <Box sx={{paddingLeft: "20px", paddingTop: "20px"}}>
        <WalletModal open={open} close={handleClose}/>
    </Box>
</Provider>

```

그리고나서 <Provider/>의 value 속성에 하위 컨포넌트로 전달하고자 하는 객체를 지정하면 됩니다. 여기서는 setWeb3 함수를 전달했습니다(나중에 다른 것도 추가될 것입니다). 하위 컨포넌트 WalletModal에서는 이렇게 넘어온 객체를 useContext() 함수를 이용하여 받습니다.

```

import { Context } from "../App";

function WalletModal(props) {

```

```

const setWeb3 = useContext(Context).setWeb3;

const handleConnectMM = async () => {
    try {
        await injectedWeb3Connector.activate();

        injectedWeb3Connector.getProvider().then(p => {
            setWeb3(new ethers.providers.Web3Provider(p));
        });
    } catch (err) {
        if (err instanceof UserRejectedRequestErrorMM) {
            console.log("UserRejectedRequest...");
            props.close();
        }
    }
}

```

지갑이 연결되고나면 Ethers.js의 ethers.provider.Web3Provider를 사용하여 Web3 Provider를 생성시키고 그것을 setWeb3를 통해 App.js의 상태인 web3에 저장합니다. Web3 Provider가 web3에 저장되었으므로 앞으로는 Ethers.js를 사용하여 이더리움과 데이터를 주고 받을 수 있습니다. 예를 들어서 계정 주소를 가져오려면 다음과 같이 getSigner(0).getAddress()를 하면 되겠습니다.

```
web3.getSigner(0).getAddress().then(v=>console.log(v))
```

이제 계정 주소와 잔액을 상단에 표시하기로 합니다. 주소와 잔액을 나타내는 상태를 만듭니다.

```

const [account, setAccount] = useState("0x");
const [balance, setBalance] = useState("0");

```

web3가 만들어지면 주소와 잔액을 조회한 후 상태를 변경해주면 됩니다. web3의 변경을 감지하여 애플리케이션의 다른 상태를 업데이트할 때 쓸 수 있는 것이 리액트의 useEffect()라는 함수입니다. 지갑이 연결된 후 주소와 잔액의 상태를 업데이트하는 것입니다.

useEffect()는 다음과 같은 형식입니다. 의존성 배열 [dependency]의 상태가 변할 때마다 안에 정의된 함수를 실행하는 것입니다.

```
useEffect(() => { ... }, [dependency]);
```

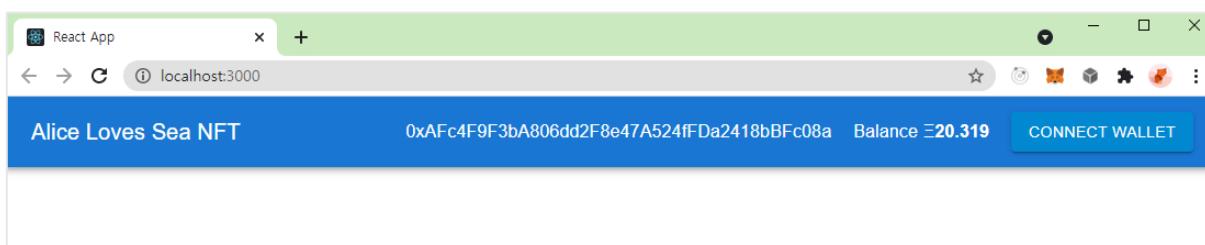
useEffect() 안에 들어가는 함수에는 web3가 새로 생성될 때 주소와 잔액을 가져오는 코드를 작성해 주면 되겠습니다. getAddress()와 getBalance()로 주소와 잔액을 가져온 후에 setAccount와 setBalance로 상태를 업데이트해주면 화면이 다시 렌더링되면서 정보가 표시될 것입니다.

```
useEffect(() => {
  if (!isEmpty(web3)) {
    web3.getSigner(0).getBalance().then(
      v=>setBalance(parseFloat(ethers.utils.formatEther(v)).toFixed(3)));
    web3.getSigner(0).getAddress().then(v=>setAccount(v));
    setOpen(false);
  }
}, [web3]);
```

<Toolbar/> 컴포넌트 안에 주소와 잔액을 표시할 <Box/>를 넣습니다.

```
<AppBar position="static">
  <Toolbar>
    <Typography variant="h6" sx={{ flexGrow: 1 }}>
      Alice Loves Sea NFT
    </Typography>
    <Box sx={{ paddingRight: "20px" }}>
      <Typography sx={{ fontSize: "medium" }}>
        {account}
      </Typography>
    </Box>
    <Box sx={{ paddingRight: "20px" }}>
      <Typography sx={{ fontSize: "medium" }}>
        Balance ⚡<b>{balance}</b>
      </Typography>
    </Box>
    <Button color="info" variant="contained"
      onClick={handleOpen}>connect wallet</Button>
  </Toolbar>
</AppBar>
```

완성 화면은 다음과 같습니다.



연결된 이후에는 "CONNECT WALLET"을 "DISCONNECT" 버튼으로 바꾸는 것이 좋을 것 같습니다. 지갑을 연결 또는 종료 할 때마다 버튼의 캡션을 변경하고 상태를 업데이트합니다. 지갑 연결 버튼은 토글 버튼처럼 연결 전에는 "CONNECT"로 연결 후에는 "DISCONNECT"가 되는 것입니다. App.js의 전체 소스코드는 다음과 같습니다.

```
import {Fragment, useState, createContext, useEffect} from "react";
import Box from '@mui/material/Box';
import {
  AppBar,
  Button,
  Toolbar,
  Typography
} from "@mui/material";

import { ethers } from "ethers";
import WalletModal from "./modals/WalletModal";
import {isEmpty} from "lodash";

const Context = createContext();
const Provider = Context.Provider;

function App() {

  const CONNECT_TEXT = "connect wallet";
  const DISCONNECT_TEXT = "disconnect";

  const [open, setOpen] = useState(false);
  const [web3, setWeb3] = useState({});

  const [account, setAccount] = useState("0x");
  const [balance, setBalance] = useState("0");
  const [btnText, setBtnText] = useState(CONNECT_TEXT);

  const handleOpen = async () => {
    if (btnText === DISCONNECT_TEXT) {
      setWeb3(null);
      setBalance(0);
      setAccount("0x");
      setBtnText(CONNECT_TEXT);
    } else {
      setOpen(true);
    }
  }
  const handleClose = () => {
    setOpen(false);
  }

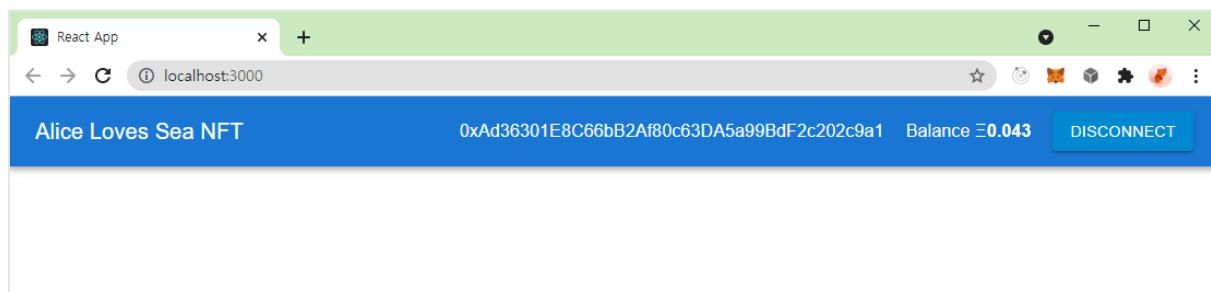
  useEffect(() => {
    if (!isEmpty(web3)) {
      web3.getSigner(0).getAddress().then(v=>setAccount(v));
      web3.getSigner(0).getBalance().then(
        v=>setBalance(parseFloat(ethers.utils.formatEther(v)).toFixed(3)))
      setOpen(false);
      setBtnText(DISCONNECT_TEXT);
    }
  }, [web3]);
}
```

```

        return (
            <Fragment>
                <Provider value={setWeb3}>
                    <AppBar position="static">
                        <Toolbar>
                            <Typography variant="h6" sx={{ flexGrow: 1 }}>
                                Alice Loves Sea NFT
                            </Typography>
                            <Box sx={{ paddingRight: "20px" }}>
                                <Typography sx={{ fontSize: "medium" }}>
                                    {account}
                                </Typography>
                            </Box>
                            <Box sx={{ paddingRight: "20px" }}>
                                <Typography sx={{ fontSize: "medium" }}>
                                    Balance ⚡<b>{balance}</b>
                                </Typography>
                            </Box>
                            <Button color="info" variant="contained"
                                onClick={handleOpen}>{btnText}</Button>
                        </Toolbar>
                    </AppBar>
                    <Box sx={{ paddingLeft: "20px", paddingTop: "20px" }}>
                        <WalletModal open={open} close={ handleClose }/>
                    </Provider>
                </Fragment>
            );
        }

        export default App;
        export {
            Context
        }
    
```

연결이 된 후에는 “DISCONNECT”로 버튼 텍스트를 바꾸고 클릭하여 연결을 종료하면 생성된 web3, account, balance 등의 정보를 초기화하여 애플리케이션을 처음 상태로 만들면 됩니다.



다음에는 WalletConnect를 지원하는 모바일 지갑을 연결하는 handleConnectWC 함수 작성하겠습니다. 우선 @web3-react/walletconnect-connector를 import 합니다.

```
import { WalletConnectConnector, UserRejectedRequestError as UserRejectedRequestErrorWC }  
from "@web3-react/walletconnect-connector";
```

WalletConnectConnector가 인스턴스를 생성하는 함수입니다. WalletModal(props) 함수 밖에서 작성하도록 합니다. 전달인자는 이더리움 네트워크별 RPC endpoint입니다. 여기서는 1번 메인넷과 4번 Rinkeby를 사용합니다.

```
const walletConnecter = new WalletConnectConnector({  
    rpc: {  
        1: "https://eth-mainnet.alchemyapi.io/v2/...LWAZ",  
        4: "https://eth-rinkeby.alchemyapi.io/v2/...svMCb"  
    },  
    qrcode: true  
});
```

RPC endpoint를 위한 이더리움 API 서비스는 5장에서 사용했던 알케미를 이용하도록 하겠습니다. 알케미에서 받은 API 키를 체인 번호에 맞게 rpc 항목에 설정합니다. Rinkeby를 사용하기로 했으므로 4번만 작성해도 무방합니다.

handleConnectWC 함수는 다음과 같습니다. 메타마스크 연결과 거의 유사하고 다만 사용자가 연결을 거절하는 예외처리에서 조금 차이가 있습니다. 마찬가지로 Web3 Provider를 만들어서 setWeb3에 전달합니다.

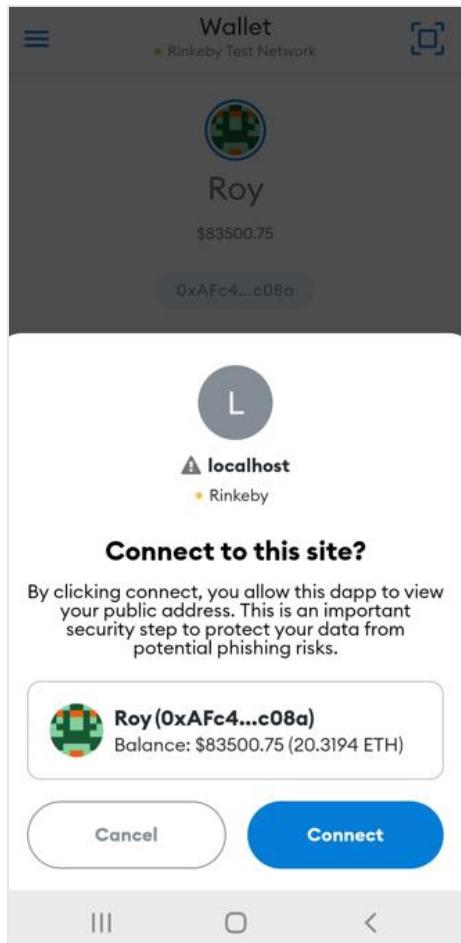
```
const handleConnectWC = async () => {  
    try {  
  
        await walletConnecter.activate();  
  
        walletConnecter.getProvider().then(p => {  
            setWeb3(new ethers.providers.Web3Provider(p));  
        });  
  
    } catch (err) {  
  
        if (err instanceof UserRejectedRequestErrorWC) {  
            console.log("UserRejectedRequest...");  
            walletConnecter.walletConnectProvider = undefined;  
            props.close();  
        }  
    }  
}
```

모달 팝업에서 WALLETCONNECT 버튼을 클릭하면 아래 그림과 같은 QR 코드가 나타납니다. QR 코드를 모바일 기기에 설치된 WalletConnect 지원 지갑으로 스캔합니다.

모바일 지갑으로 QR코드를 스캔하면 브릿지 서버(공용 서버 bridge.walletconnect.org)가 Dapp과 모바일 지갑을 중개하면서 암호화된 세션을 생성합니다.



WalletConnect를 지원하는 지갑, 예를 들면 메타마스크 모바일 지갑으로 QR코드를 스캔하면(브릿지 서버와의 연결이 지연될 수 있어서 연결 수락 화면이 다소 늦게 나올 수도 있겠습니다) 모바일 기기에서 "Connect to this site?"라는 알림이 표시됩니다.



Connect를 누르면 Dapp에 연결되고 메타마스크와 동일하게 주소와 잔액이 표시됩니다. 네트워크상태에 따라 약간 지연이 발생할 수도 있습니다. 이후에는 암호화된 세션을 통해서 서명 요청과 서명데이터가 만들어지고 앞서 설정된 RPC endpoint로 보냅니다.

WalletConnect를 테스트하는 것은 다소 번거로운 부분이 있습니다. Dapp, RPC서버, 브릿지 서버 그리고 모바일 지갑이 서로 통신을 해야 하고 응답이 느린 경우도 있습니다. 이렇게 연결된 세션을 지우려면 모바일 지갑에서 Dapp과 연결된 세션을 지우거나 웹브라우저의 개발자 모드(F12)의 Application 탭의 로컬 스토리지에 저장된 WalletConnect 정보를 수동으로 삭제하면 됩니다.

Sources	Network	Performance	Memory	Application	Security	Lighthouse	Redux	
				walletconnect	{"connected":true,"accounts":["0xAFc4F9F3bA806dd2F8e47A524ffFDa2418bBFc08a"],"chainId":4,"bridge":"https://mainnet.infura.io/v3/...LWAZ","ethereumAddress":"0xAFc4F9F3bA806dd2F8e47A524ffFDa2418bBFc08a","gasPrice":1000000000,"gasPriceWei":1000000000000000000,"gasUsed":0,"gasUsedWei":0,"isConnected":true,"method":null,"nonce":0,"publicKey":null,"publicKeyHex":null,"signature":null,"signatureHex":null,"transactionHash":null,"transactionHashHex":null,"transactionIndex":0,"transactionType":0,"version":1}			

메타마스크와 WalletConnect는 이전에 연결을 수락한 Dapp의 정보를 유지하고 있기 때문에 연결 정보를 수동으로 초기화하지 않는 한 재접속할 때 자동으로 연결됩니다.

WalletModal.js의 전체 소스코드는 아래와 같습니다.

```

import React, { useContext } from "react";
import { Box, Button, Modal } from "@mui/material";
import { InjectedConnector, UserRejectedRequestError
as UserRejectedRequestErrorMM } from "@web3-react/injected-connector";
import { WalletConnectConnector, UserRejectedRequestError
as UserRejectedRequestErrorWC } from "@web3-react/walletconnect-connector";

import { Context } from "../App";
import { ethers } from "ethers";

const style = {
  position: 'absolute',
  top: '50%',
  left: '50%',
  transform: 'translate(-50%, -50%)',
  width: 320,
  backgroundColor: 'white',
  border: '1px solid #000',
  boxShadow: 24,
  p: 4,
  display: 'flex',
  flexDirection: 'column'
};

const injectedWeb3Connector = new InjectedConnector({ supportedChainIds: [1,4] });
const walletConnecter = new WalletConnectConnector({
  rpc: {
    1: "https://eth-mainnet.alchemyapi.io/v2/...LWAZ",
    4: "https://eth-rinkeby.alchemyapi.io/v2/...vMCb"
  },
  qrcode: true
});

function WalletModal(props) {
  const setWeb3 = useContext(Context);
  const handleConnectMM = async () => {
    ...
  }
}

```

```

try {
    await injectedWeb3Connector.activate();

    injectedWeb3Connector.getProvider().then(p => {
        setWeb3(new ethers.providers.Web3Provider(p));
    });
}

} catch (err) {

    if (err instanceof UserRejectedRequestErrorMM) {
        console.log("UserRejectedRequest...");
        props.close();
    }
}
}

const handleConnectWC = async () => {
try {

    await walletConnecter.activate();

    walletConnecter.getProvider().then(p => {
        setWeb3(new ethers.providers.Web3Provider(p));
    });

} catch (err) {

    if (err instanceof UserRejectedRequestErrorWC) {
        console.log("UserRejectedRequest...");
        walletConnecter.walletConnectProvider = undefined;
        props.close();
    }
}
}

return (
    <Modal open={props.open} onClose={props.close}>
        <Box sx={style}>
            <Button variant="contained"
                sx={{marginBottom: '10px'}} onClick={handleConnectMM}>Metamask</Button>
            <Button variant="contained" onClick={handleConnectWC}>WalletConnect</Button>
        </Box>
    </Modal>
);
}

export default WalletModal;

```

*깃허브에 `@web3-react/injected-connector`를 이용하여 약간 다른 방식으로 구현한 메타마스크 연결 예제가 있습니다. 아래 링크를 참고하세요.

<https://github.com/boyd-dev/alice-NFT/tree/main/injectedMM>

6.5.3. NFT 메타정보

화면에 NFT를 보여주기 위해서 메타정보를 활용할 수 있습니다. 메타정보에 NFT의 이미지와 속성 정보가 있기 때문입니다.

6.3에서 이미지를 조합하여 NFT 이미지 500개를 생성했습니다. 이미지 정보를 목록으로 보여주는 것은 여러 가지 방법으로 구현할 수 있는데, 500개의 이미지를 한 화면에 보여주는 것은 비효율적이므로 몇 개씩 나누어서 보여주기로 하겠습니다.

앞에서 리액트 애플리케이션은 기본적으로 단일 페이지(SPA)로 구성된다고 했습니다. 보통 웹에서는 다른 페이지를 보여주려면 웹서버로 요청을 하고 그 응답으로 새로운 페이지를 받으면 웹브라우저가 그것을 보여주는 방식입니다.

하지만 리액트의 프론트엔드는 처음부터 미리 모든 화면요소들을 렌더링할 수 있는 JS 파일들을 다운로드하고 데이터만을 서버에 요청하는 방식으로 동작합니다. 이것을 “클라이언트 사이드 렌더링(client-side rendering)”이라고 하는데, 이를 위해서는 요청 URL에 따라 다른 컴포넌트로 전환해주는 기능이 있어야 합니다.

이것을 가능하게 해주는 것이 바로 리액트 “라우터(Router)”입니다. 리액트 라우터를 사용하면 URL 경로에 따라 지정된 컴포넌트로 “라우팅”할 수 있습니다. 마치 서버에 요청하여 새로운 페이지를 받아오는 것처럼 보이지만 사실은 웹브라우저 안에서 특정 뷰로 바뀌는 것입니다.

이미지 목록을 분할해서 보여주기 위해서는, 예를 들면 처음에는 1번부터 6번, 그 다음에는 7번부터 12번, 이런 식으로 다른 페이지를 렌더링해야 하기 때문에 리액트 라우트를 사용할 것입니다. 5번째 페이지를 요청하는 것은 아래와 같은 형식의 URL이 될 것입니다. 5가 경로 파라미터가 되고 해당 페이지를 렌더링합니다.

`http://localhost:3000/5`

리액트 라우터 중에 가장 많이 사용되는 것이 “react-router-dom”이라는 패키지입니다. 다음과 같이 설치합니다. 여기서 사용하는 버전은 6.0.2입니다.

```
yarn add react-router-dom
```

앞서 App.js에 이미 NFT목록이 표시될 자리를 만들어 두었습니다. `/* NFT List */`라고 주석 처리된 부분에 NFT 이미지 리스트를 보여줄 것입니다.

```

<Fragment>
  <Provider value={setWeb3}>
    <AppBar position="static">
      <Toolbar>
        <Typography variant="h6" sx={{ flexGrow: 1 }}>
          Alice Loves Sea NFT
        </Typography>
        <Box sx={{ paddingRight: "20px" }}>
          <Typography sx={{ fontSize: "medium" }}>
            {account}
          </Typography>
        </Box>
        <Box sx={{ paddingRight: "20px" }}>
          <Typography sx={{ fontSize: "medium" }}>
            Balance ⚡<b>{balance}</b>
          </Typography>
        </Box>
        <Button color="info" variant="contained"
          onClick={handleOpen}>{btnText}</Button>
      </Toolbar>
    </AppBar>
    <Box sx={{ paddingLeft: "20px", paddingTop: "20px" }}>
      {/* NFT List */}
    </Box>
    <WalletModal open={open} close={ handleClose } />
  </Provider>
</Fragment>

```

리액트 라우터는 URL에 따라 컴포넌트를 선택적으로 표시하므로 {/* NFT List */}에 라우터를 배치할 것입니다. react-router-dom에서 제공하는 <BrowserRouter/>를 사용하면 되는데, 이 컴포넌트는 이름 그대로 웹브라우저의 history API를 사용하는 라우터입니다(웹페이지에서 뒤로 가기를 구현할 때 사용했던 window.history.back()이 바로 history API를 이용한 것입니다). 리액트 라우터를 연결하는 것은 6.5.5에서 설명합니다.

먼저 NFT 목록을 생성하는 컴포넌트를 만들어야 합니다. 컴포넌트 이름을 <AlsList/>라고 하겠습니다. <BrowserRouter/>가 경로에 따라 <AlsList/>로 라우팅시키면 NFT 이미지 리스트가 표시되도록 할 것입니다.



NFT 이미지의 정보는 MUI에서 흔히 “카드(Card)”라고 말하는 컴포넌트를 사용할 것입니다. 카드는 이미지와 함께 카드 스타일로 표시하는 형식을 말합니다. 왼쪽 그림과 같은 형태가 전형적인 카드입니다.

NFT 이미지가 상단에 표시되고 그 아래에 토큰 번호와 속성, 그리고 기능 버튼들을 배치하게 될 것입니다. 속성이 세 개라서 모두 표시가 가능하지만 많을 경우에는 다른 방식을 사용해야 합니다.

이와 같은 형태를 MUI의 `<Card/>` 컴포넌트로 표현하면 아래와 같습니다.

```
<Card sx={{maxWidth: "220px"}}>
  <CardActionArea>
    <CardMedia component="img" image={...}/>
    <CardContent>
      Token ID: 500
      <br/>
      Face: gogo
      <br/>
      Body: orangebody
      <br/>
      Background: blue
    </CardContent>
  </CardActionArea>
  <CardActions>
    <Button color="primary" variant="contained">mint</Button>
    <Button color="primary" variant="contained">sale</Button>
  </CardActions>
</Card>
```

`<CardActionArea/>`에 이미지와 정보를 넣고 `<CardActions/>`에 기능 버튼을 넣습니다. 카드 아이템들을 목록으로 표시할 때는 MUI의 `<ImageList/>`를 사용합니다. `<Card/>`를 `<ImageListItem/>` 안에 배치하면 이미지 리스트가 만들어지게 될 것입니다.

```
<ImageList sx={{ width: "800px" }} cols={3}>
  <ImageListItem key={i}>
    <Card sx={{maxWidth: "220px"}}>
      ...
    </Card>
  </ImageListItem>
  <ImageListItem key={i}>
    <Card sx={{maxWidth: "220px"}}>
      ...
    </Card>
  </ImageListItem>
</ImageList>
```

```
</Card>
</ImageListItem>
...
</ImageList>
```

<ImageList/> 컴포넌트의 cols 속성은 컬럼의 수를 의미하기 때문에 가로 방향으로 세 개씩 보여주려면 3을 넣으면 되겠습니다.

이렇게 목록을 구성하고 500개의 이미지 데이터를 순차적으로 읽어서 카드에 넣어주면 되겠습니다. 이미 6.3에서 이미지 500개를 생성했으므로 이것을 <CardMedia/>의 image 속성에 전달할 수 있습니다.

NFT의 메타정보는 로컬에 생성되어 있거나 IPFS에 업로드 되어 있습니다. 로컬에 있는 데이터를 보여줄 수도 있지만 여기서는 IPFS에 있는 것을 가져오기로 합니다. 다만 IPFS 공용서버를 통해 이미지를 가져오기 때문에 다소 느리다는 문제는 있습니다.

먼저 외부의 메타정보를 조회하는 함수를 만들도록 하겠습니다. IPFS에서 리턴되는 메타정보는 JSON 형식이므로 자바스크립트의 fetch 함수를 사용합니다. 다음과 같이 getMetadata()라는 함수를 작성할 수 있습니다.

```
const getMetadata = (index) => {
  return new Promise((resolve, reject) => {
    fetch(data)
      .then(r => r.text())
      .then(text => {
        const regexp = new RegExp("(\\r?\\n)?\" + index + "=(.*)\\metadata\\.json", "g");
        const result = text.toString().match(regexp);

        if (result != null) {
          fetch(result[0].slice(result[0].indexOf("=")+1))
            .then(res => res.json())
            .then(data => {
              resolve({
                "image": data.image.toString()
                  .replaceAll("ipfs://", "https://dweb.link/ipfs/"),
                "attributes": data.attributes
              });
            });
        } else {
          reject("Failed to fetch metadata from uri");
        }
      });
  });
}
```

getMetadata() 함수는 토큰 번호를 파라미터로 받아서 이미 생성된 메타정보 파일을 읽고 IPFS의

URL을 가져온 뒤에 다시 그것을 요청합니다. 요청한 결과로부터 image와 attributes 값을 추출하여 리턴하도록 되어 있습니다. image URL은 HTTP로 접근 가능하도록 https://dweb.link/로 대체했습니다.

이제 메타정보를 보여주는 컴포넌트 <Metadata/>를 작성합니다. 메타정보는 <CardActionArea/>에 표시되므로 아래와 같이 분리할 수 있습니다. image와 attributes는 getMetadata() 함수를 호출하여 얻은 값을 넣으면 되겠습니다.

```
<CardActionArea>
  <CardMedia component="img" image={image}/>
  <CardContent>
    {attributes}
  </CardContent>
</CardActionArea>
```

<Metadata/> 컴포넌트는 다음과 같이 구현할 수 있습니다. 토큰 번호를 받아서 getMetadata()를 호출하고 <CardActionArea/>로 만들어서 리턴합니다.

```
import React, {useEffect, useState} from 'react';

import {getMetadata} from "../utils";
import {CardActionArea, CardContent, CardMedia} from "@mui/material";

function Metadata(props) {

  const [image, setImage] = useState("");
  const [attributes, setAttributes] = useState("");

  useEffect(() => {
    getMetadata(props.id).then(v => {
      setImage(v.image);
      if (v.attributes instanceof Array) {
        setAttributes(v.attributes.map((o,k)=>
          (<p key={k}>{o.trait_type}: {o.value}</p>)));
      }
    });
  }, []);

  return (
    <CardActionArea>
      <CardMedia component="img" image={image}/>
      <CardContent>
        Token ID: {props.id}
        <br/>
        {attributes}
      </CardContent>
    </CardActionArea>
  );
}

export default Metadata;
```

props.id가 토큰 번호에 해당합니다. 리액트의 useEffect()는 외부의 데이터를 가져올 때 사용하는데, 지금처럼 IPFS의 메타정보를 HTTP를 통해 받는 경우에 해당합니다.

이제 리스트를 만들어주는 함수를 작성해야 합니다. 위에서는 토큰을 카드 형태로 하여 메타정보를 보여주는 것을 구현했으므로 이것을 목록을 보여주는 작업이 필요합니다.

전달인자 i에 따라 <ImageListItem/> 하나를 만들어주는 함수 item()을 작성합니다.

```
import Metadata from "./Metadata";

const item = (i) => (<ImageListItem key={i}>
  <Card sx={{maxWidth: "220px"}}>
    <Metadata id={i+1}/>
    <CardActions>
      <Button color="primary" variant="contained">mint</Button>
      <Button color="primary" variant="contained">sale</Button>
    </CardActions>
  </Card>
</ImageListItem>)
```

<AlsList/>는 최종적인 토큰 목록을 리턴하는 컴포넌트입니다. Item()함수를 list.map()을 이용하여 호출하고 NFT 목록 <ImageList/>를 생성합니다. 여기서 list는 배열인데 배열의 길이가 한 페이지에 보여줄 아이템 개수에 해당합니다. 나중에 목록 페이지 분할에서 구현될 것입니다.

<AlsList/>의 전체 소스는 다음과 같습니다.

```
import React, { Fragment } from 'react';
import {
  Button, ButtonGroup,
  Card, CardActionArea, CardActions, CardContent, CardMedia, ImageList,
  ImageListItem
} from "@mui/material";

import ALSImages from "../images/_Final";

function AlsList() {

  const item = (i) => (<ImageListItem key={i}>
    <Card sx={{maxWidth: "220px"}}>
      <Metadata id={i+1}/>
      <CardActions>
        <Button color="primary" variant="contained">mint</Button>
        <Button color="primary" variant="contained">sale</Button>
      </CardActions>
    </Card>
  </ImageListItem>)

  return (
    <Fragment>
```

```

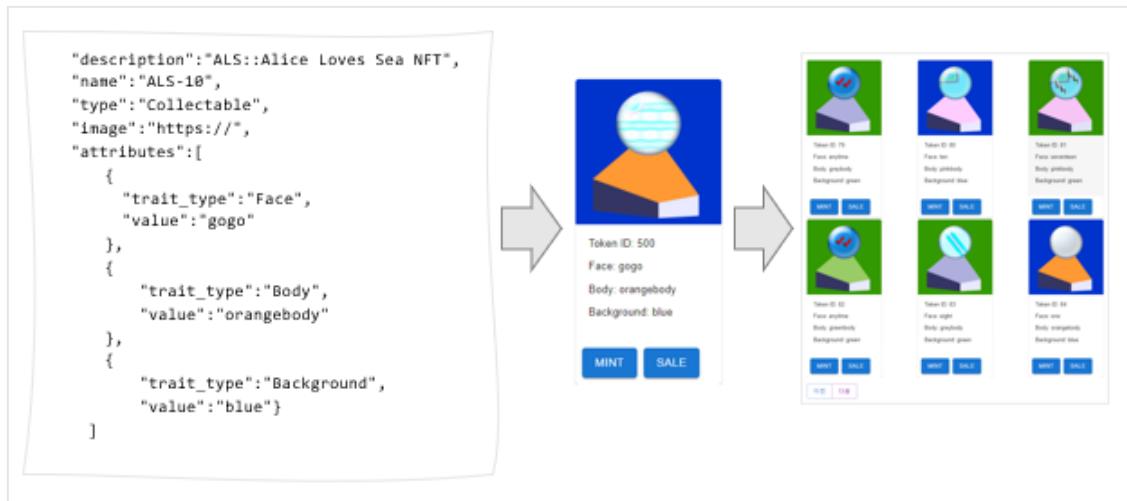
        <ImageList sx={{ width: "800px" }} cols={3}>
          {
            list.map(v => (item(v)))
          }
        </ImageList>
        <ButtonGroup>
          <Button color="primary" onClick={handlePrev}>이전</Button>
          <Button color="secondary" onClick={handleNext}>다음</Button>
        </ButtonGroup>
      </Fragment>
    );
}

export default AlsList;

```

지금까지 과정을 요약하면 다음과 같습니다.

NFT 토큰의 이미지를 만들기 위해 랜덤으로 조합된 500개의 이미지를 생성하고 메타정보와 함께 IPFS에 업로드했습니다(URL정보를 파일로 저장합니다). 이렇게 생성된 토큰들을 화면에 보여주기 위해 <Card/> 컴포넌트를 만들고 그것을 다시 <ImageListItem/>과 <ImageList/>로 화면에 표시했습니다. 메타정보는 IPFS에 업로드된 것을 fetch()를 사용하여 가져옵니다.



그런데 500개의 이미지가 한 화면에 나열된다면 시간도 오래 걸리고 무척 불편할 것입니다. 그래서 한 화면에 일정한 개수의 이미지들만 보여주기로 합니다.

6.5.4. NFT 목록과 페이지 분할

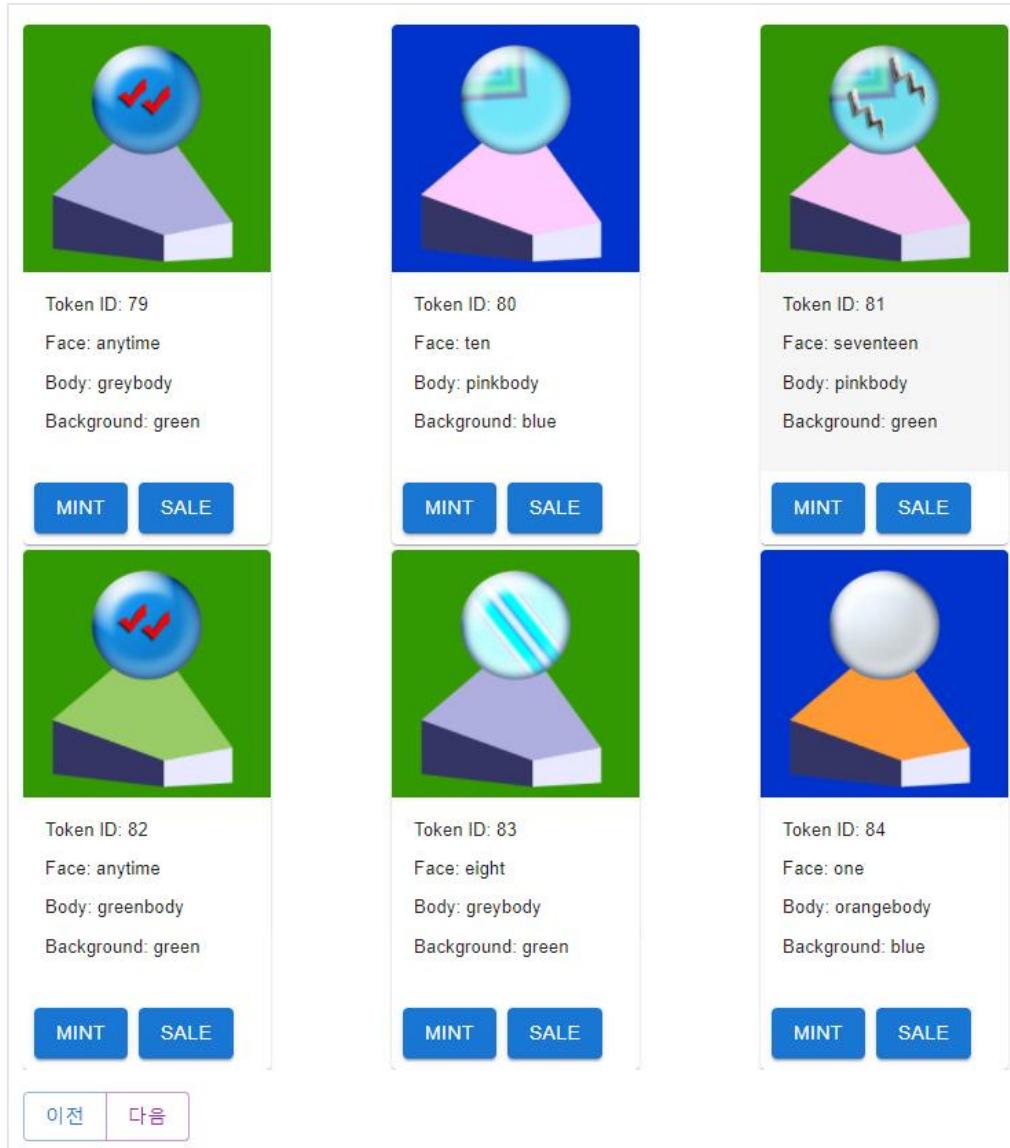
목록 페이지는 아래 그림과 같은 형태가 되겠습니다. <ImageList/>의 col 속성을 3으로 주었기 때문에 가로 방향으로 3개씩 그리고 세로로 두 개 행이 표시가 됩니다. “다음”이나 “이전” 버튼을 클릭하여 다음 페이지로 이동할 수 있습니다.

이제부터 할 일은 한 화면에 표시되는 이미지의 개수를 제한하는 것입니다. 아마 게시판 페이징과 유사한 로직이 적용될 수 있을 것 같습니다.

페이징을 하기 위해서는 기본적으로 전체 건수와 한 페이지에 보여줄 게시물의 수 등의 정보가 결정되어야 합니다. ALS NFT의 전체 건수 TOTAL_COUNT는 500이고 한 페이지에 보여주는 이미지 ITEMS_PER_PAGE는 6으로 하겠습니다. 이 경우에 다음과 같은 상수들이 계산될 수 있습니다.

```
const ITEMS_PER_PAGE = 6;
const TOTAL_COUNT = 500;
const FIRST_PAGE = 1;
const LAST_PAGE = Math.ceil(TOTAL_COUNT/ITEMS_PER_PAGE);
const START_INDEX_ON_LAST_PAGE = TOTAL_COUNT - (TOTAL_COUNT % ITEMS_PER_PAGE);
```

LAST_PAGE는 마지막 페이지인데 TOTAL_COUNT를 ITEMS_PER_PAGE로 나눈 몫을 올림(ceil)한 값이 됩니다. FIRST_PAGE와 LAST_PAGE는 내비게이션에서 더 이상 이동할 페이지가 없는 경우를 판단하기 위해 미리 계산합니다.



START_INDEX_ON_LAST_PAGE는 마지막 페이지에 나오는 첫 번째 토큰의 인덱스입니다. 500개를 6개씩 보여주면 남은 2개가 마지막 페이지에 표시되고 시작 인덱스는 498이 됩니다(인덱스는 항상 0부터 시작합니다). 마지막 페이지에서 NFT 목록을 표시할 때는 6회 반복하지 않고 2회만 반복하면 되겠습니다.

<AlsList/> 컴포넌트에서 NFT 목록을 표시할 때 list.map()으로 원소들을 하나씩 꺼내서 item()에 전달하도록 했습니다.

```

<Fragment>
  <ImageList sx={{ width: "800px" }} cols={3}>
    {
      list.map(v => (item(v)))
    }
  </ImageList>
</Fragment>

```

```
</ImageList>
<ButtonGroup>
    <Button color="primary" onClick={handlePrev}>이전</Button>
    <Button color="secondary" onClick={handleNext}>다음</Button>
</ButtonGroup>
</Fragment>
```

한 화면에 ITEMS_PER_PAGE 개수만큼 보여주기로 했으므로 list는 다음과 같이 만들 수 있습니다. 원소의 값은 중요하지 않기 때문에 그냥 인덱스를 원소로 하면 됩니다.

```
for (let i=0; i<ITEMS_PER_PAGE; i++) {
    list.push(i);
}
```

마지막 페이지에서는 TOTAL_COUNT % ITEMS_PER_PAGE 즉 2번만 반복합니다(%는 나머지 연산).

```
if (startIndex === START_INDEX_ON_LAST_PAGE) {
    list.splice(0);
    for (let i=0; i<(TOTAL_COUNT % ITEMS_PER_PAGE); i++) {
        list.push(i);
    }
}
```

목록을 페이지로 분할하는 로직을 간단히 요약하면, 페이지 번호에 해당하는 값을 받아서 해당 페이지의 첫 번째 인덱스를 구하고 ITEMS_PER_PAGE만큼 반복하는 것입니다. 각 페이지의 첫 번째 인덱스 startIndex는 다음과 같이 계산할 수 있습니다. 여기서 pageNo는 페이지 번호에 해당합니다.

```
startIndex = ITEMS_PER_PAGE*(pageNo-1);
```

예를 들어 1페이지 즉 페이지 번호 pageNo=1이면 시작 인덱스는 0이 될 것이고 ALS 토큰 1번부터 6번까지, 2페이지는 시작 인덱스는 6이 되고 7번부터 12번이 표시될 것입니다. pageNo는 나중에 리액트 라우터에 넘겨주는 경로 파라미터가 될 것입니다.

한 화면에 보여주는 개수인 ITEMS_PER_PAGE를 변경하더라도 코드 변경없이 동일하게 화면이 구성될 수 있습니다. 페이지를 이동하는 “이전”과 “다음” 버튼은 나중에 나오는 리액트 라우터의 navigate를 이용할 것입니다.

```
const handleNext = () => {
    //TODO navigate
}

const handlePrev = () => {
    //TODO navigate
}
```

● 리액트 라우터 연결

이제 pageNo에 따라서 화면을 전환해야 합니다. 앞에서 언급한 것처럼 리액트 라우터를 이용할 것입니다.

App.js에서 NFT 목록이 들어갈 위치는 `/* NFT List */`입니다. 바로 여기에 pageNo에 따라서 목록을 보여주도록 라우팅하면 되겠습니다. react-router-dom에서 제공하는 `<BrowserRouter>`을 사용합니다. `<BrowserRouter>`의 기본 사용법은 간단합니다. 요청 URL에 따라 표시될 `<Route>` 컴포넌트들을 내부에 배치하면 되겠습니다.

```
<Box sx={{paddingLeft: "20px", paddingTop: "20px"}}>
    {/* NFT List */}
    <BrowserRouter>
        <Routes>
            <Route path="/">
                <Route path=":pageNo" element={<AlsList />} />
                <Route path="" element={<AlsList />} />
            </Route>
        </Routes>
    </BrowserRouter>
</Box>
```

`<Route>`의 경로 속성인 `path`에 따라 `element` 속성에 지정된 컴포넌트가 표시됩니다. 처음 페이지가 열리는 경우는 루트(/)이므로 루트에 매칭되는 컴포넌트 `<AlsList/>`가 표시될 것입니다.

경로 파라미터는 `path=":pageNo"`로 설정하는데 `pageNo`가 있는 경우에는 `pageNo`에 속한 NFT 목록으로 라우팅될 것입니다. 예를 들어 `http://localhost:3000/5` 요청은 `pageNo=5`로 전달될 것입니다.

`<AlsList/>`컴포넌트에서 `pageNo`를 받으려면 `useParams()`를 이용합니다. 이제 `startIndex`는 다음과 같이 계산할 수 있습니다.

```
import {useParams, useNavigate } from "react-router-dom";
...

```

```

const params = useParams();
const navigate = useNavigate();

...

if (!isEmpty(params)) {
    startIndex = ITEMS_PER_PAGE*(params.pageNo-1);

    if (startIndex === START_INDEX_ON_LAST_PAGE) {
        list.splice(0);
        for (let i=0; i<(TOTAL_COUNT % ITEMS_PER_PAGE); i++) {
            list.push(i);
        }
    }
}

```

handleNext()와 handlePrev()에서는 useNavigate()를 사용하여 라우팅, 즉 페이지 이동을 할 수 있습니다. 페이지 이동은 navigate(path)로 하면 됩니다.

그런데 가장 첫 번째 페이지와 가장 마지막 페이지에서는 “이전”이나 “다음” 버튼을 클릭했을 때 더 이상 이동할 수 없도록 막아야 합니다. 그래서 다음 페이지의 인덱스를 localStorage에 저장해두기로 하겠습니다.

```

const next = localStorage.getItem("ALS.next");
localStorage.setItem("ALS.next", startIndex+ITEMS_PER_PAGE);

const handleNext = () => {
    const v = parseInt(next/ITEMS_PER_PAGE)+1;
    if (v <= LAST_PAGE) {
        navigate(`/${v}`);
    }
}

const handlePrev = () => {
    const v = parseInt(next/ITEMS_PER_PAGE)-1;
    if (v >= FIRST_PAGE) {
        navigate(`/${v}`);
    }
}

```

완성된 <AlsList/> 컴포넌트 전체 소스는 아래와 같습니다.

```

import React, { Fragment } from 'react';
import {
    Button, ButtonGroup,
    Card,
    CardActions,
    ImageList,
    ImageListItem
} from "@mui/material";

```

```

import {useParams, useNavigate } from "react-router-dom";
import {isEmpty} from "lodash";
import Metadata from "./Metadata";

const ITEMS_PER_PAGE = 6;
const TOTAL_COUNT = 500;
const FIRST_PAGE = 1;
const LAST_PAGE = Math.ceil(TOTAL_COUNT/ITEMS_PER_PAGE);
const START_INDEX_ON_LAST_PAGE = TOTAL_COUNT - (TOTAL_COUNT % ITEMS_PER_PAGE);

function AlsList() {

    let startIndex = 0;
    let list = [];
    const params = useParams();
    const navigate = useNavigate();

    for (let i=0; i<ITEMS_PER_PAGE; i++) {
        list.push(i);
    }

    if (!isEmpty(params)) {

        startIndex = ITEMS_PER_PAGE*(params.pageNo-1);

        if (startIndex === START_INDEX_ON_LAST_PAGE) {
            list.splice(0);
            for (let i=0; i<(TOTAL_COUNT % ITEMS_PER_PAGE); i++) {
                list.push(i);
            }
        }
    }

    const next = localStorage.getItem("ALS.next");
    localStorage.setItem("ALS.next", startIndex+ITEMS_PER_PAGE);

    const handleNext = () => {
        const v = parseInt(next/ITEMS_PER_PAGE)+1;
        if (v <= LAST_PAGE) {
            navigate(`/${v}`);
        }
    }

    const handlePrev = () => {
        const v = parseInt(next/ITEMS_PER_PAGE)-1;
        if (v >= FIRST_PAGE) {
            navigate(`/${v}`);
        }
    }

    const item = (i) => (<ImageListItem key={i}>
        <Card sx={{maxWidth: "180px"}}>
            <Metadata id={i+1}/>
            <CardActions>
                <Button color="primary" variant="contained">mint</Button>
                <Button color="primary" variant="contained">sale</Button>
            </CardActions>
        </Card>
    </ImageListItem>

    return (

```

```
<Fragment>
  <ImageList sx={{ width: "800px" }} cols={3}>
    {
      list.map(v => (item(startIndex+v)))
    }
  </ImageList>
  <ButtonGroup>
    <Button color="primary" onClick={handlePrev}>이전</Button>
    <Button color="secondary" onClick={handleNext}>다음</Button>
  </ButtonGroup>
</Fragment>

);
}
export default AlsList;
```

6.5.5. 스마트 컨트랙트 호출

이제 스마트 컨트랙트의 함수를 호출하여 NFT를 발행할 차례입니다. 컨트랙트는 이미 6.4에서 Rinkeby에 배포되어 있고 메타정보도(이미지도 함께) IPFS에 업로드되어 있습니다. NFT “발행”이라는 것은 메타정보가 나타내는 것(여기서는 이미지가 되겠습니다)에 대한 소유권 정보를 컨트랙트에 기록하는 일입니다.

Dapp에서 이더리움에 배포된 스마트 컨트랙트와 데이터를 주고받기 위해서는 ABI(Application Binary Interface)를 사용해야 합니다. ABI는 컨트랙트를 컴파일한 결과물(artifacts)에 포함되어 있습니다.

5장에서 사용한 hardhat-deploy에서는 배포 결과 ALSnft.json이 deployments 폴더에 저장되어 있습니다. Rinkeby에 배포된 경우에는, networks 설정이 “rinkeby”로 되어 있다면, rinkeby 폴더에 생성되어 있을 것입니다.

alice\deployments\rinkeby			
Mode	LastWriteTime	Length	Name
---	-----	-----	-----
d-----	2021-12-01	오후 7:58	solcInputs
-a----	2021-12-01	오후 7:58	1 .chainId
-a----	2021-12-01	오후 7:58	106926 ALSnft.json

이것을 애플리케이션 프로젝트의 ./app/src/contracts로 옮길 수 있는데 여기서 사용할 수 있는 옵션이 --export입니다. 이렇게 하면 컨트랙트의 정보를 파일로 만들어주므로 애플리케이션에서 이것을 참조하면 되겠습니다.

```
npx hardhat deploy --export ./app/src/contracts/ALSnft.json --network rinkeby
```

컨트랙트의 함수를 호출하려면 Ether.js 라이브러리에서 제공하는 ethers.Contract()를 사용합니다. App.js에 다음을 추가합니다. 지갑이 연결되어 Web3 Provider가 생성된 후에 컨트랙트의 인스턴스를 생성해야 하므로 useEffect()에 추가해야 합니다. web3를 useEffect()의 의존성 배열에 추가합니다. 그리고 컨트랙트를 저장할 상태를 useState()로 정의합니다.

```
import { ethers } from "ethers";
import artifact from "./contracts/ALSnft.json";

...
const [alsNft, setAlsNft] = useState(null);
```

```

useEffect(() => {
    ...
    web3.getSigner(0).getAddress().then(v=>setAccount(v));
    web3.getSigner(0).getBalance()
        .then(v=>setBalance(parseFloat(ethers.utils.formatEther(v)).toFixed(3)));
    const als = new ethers.Contract(artifact.contracts.ALSnft.address,
        artifact.contracts.ALSnft.abi,
        web3.getSigner());
    ...
    setAlsNft(als);
    ...
}, [web3])

```

`ethers.Contract()`에 전달해주어야 하는 인자는 배포 컨트랙트의 주소, ABI, 그리고 현재 애플리케이션과 연결된 지갑 계정입니다. 컨트랙트 주소와 ABI는 모두 `ALSnft.json`에 있습니다. 연결된 계정은 `web3.getSigner()`로 가져옵니다. 이렇게 생성된 컨트랙트 인스턴스를 `setAlsNft()`를 사용하여 컴포넌트 상태로 저장합니다. 또 "DISCONNECT" 버튼을 클릭하면 컨트랙트 인스턴스 역시 초기화해야 하므로 `handleOpen()` 함수에 `setAlsNft(null)`을 추가합니다.

```

const handleOpen = async () => {
    if (btnText === DISCONNECT_TEXT) {
        setWeb3(null);
        setBalance(0);
        setAccount("0x");
        setBtnText(CONNECT_TEXT);
        setAlsNft(null);
    } else {
        setOpen(true);
    }
}

```

`App.js`의 컨트랙트 인스턴스는 하위 컴포넌트들에서 사용할 수 있어야 합니다. 역시 Context API를 이용하기로 합니다. `<Provider/>` 컴포넌트의 `value` 속성에 다음과 같이 추가합니다. 6.5.2. 지갑 연결에서 전달했던 `setWeb3`와 함께 컨트랙트 인스턴스 `alsNft`를 추가로 전달합니다.

```

<Fragment>
    <Provider value={{setWeb3, alsNft}}>
        <AppBar position="static">
            ...
        </AppBar>
        <Box sx={{paddingLeft: "20px", paddingTop: "20px"}}>
            <BrowserRouter>
                <Routes>
                    <Route path="/">

```

```

        <Route path=":pageNo" element={<AlsList />} />
        <Route path="" element={<AlsList />} />
    </Route>
</Routes>
</BrowserRouter>

</Box>
<WalletModal open={open} close={handleClose}/>
</Provider>
</Fragment>

```

이렇게 전달된 alsNft는 AlsList에서 참조합니다. useContext()를 사용하면 되겠습니다.

```

function AlsList() {

    const alsNft = useContext(Context).alsNft;
    ...
}

```

이제 ALS 토큰을 발행하기 위해 각 NFT 카드에 있는 "MINT"버튼의 기능을 구현하겠습니다. 컨트랙트 ALNsft.sol에 작성된 mint 함수를 다시 보면 다음과 같습니다.

```

function mint(address toAddr, uint256 tokenId, string memory tokenURI)
public
onlyOwner
returns (uint256) {
...
}

```

전달해야 할 파라미터는 발행된 토큰을 소유할 계정, 토큰 번호 그리고 메타정보 URI입니다. "MINT" 버튼을 클릭했을 때 처리하는 함수를 handleMint()라고 하고 아래와 같이 작성합니다. AlsList.js에 handleMint() 함수를 추가하고 버튼의 onClick 속성에 연결합니다.

```

const OWNER = "0xAFc...c08a";

function AlsList() {
    ...

    const handleMint = async (e) => {
        if (alsNft !== null) {

            const tokenId = e.target.getAttribute("tokenid");
            const tokenURI = await getTokenURI(tokenId);

            const tx = await alsNft.mint(OWNER, tokenId, tokenURI, {gasLimit: 3000000});
            try {
                await tx.wait();
            }
        }
    }
}

```

```

        } catch (error) {
            console.log(error.reason);
        }

    } else {
        console.log("WALLET DISCONNECTED");
    }
}

...

const item = (i) => (<ImageListItem key={i}>
    <Card sx={{maxWidth: "180px"}}>
        <Metadata id={i+1}/>
        <CardActions>
            <Button color="primary" variant="contained"
                onClick={handleMint} tokenid={i+1}>mint</Button>

            <Button color="primary" variant="contained">sale</Button>
        </CardActions>
    </Card>
</ImageListItem>
)

```

여기서 토큰 번호를 전달해야 그 토큰을 발행할 수 있으므로 <Button/> 컴포넌트에 커스텀 속성인 "tokenid"를 추가합니다. handleMint() 함수에서 target.getAttribute("tokenid")로 이 값을 참조할 수 있습니다.

getTokenURI() 함수는 6.3.4에서 생성된 메타정보 파일로부터 토큰 번호에 해당하는 URI값을 읽어서 리턴하는 함수인데 utils 폴더에 작성되어 있습니다.

"MINT" 버튼을 클릭하면 컨텍스트 API를 통해 전달받은 컨트랙트 인스턴스 alsNft를 사용하여 컨트랙트의 mint 함수를 호출합니다. 컨트랙트의 함수를 호출할 때는 항상 비동기 호출인 async-await를 써야 합니다. gasLimit은 트랜잭션에 소요되는 가스 값인데 충분한 값 3,000,000을 주도록 하겠습니다.

트랜잭션 실행 중에 오류가 나는 경우, 예를 들어서 컨트랙트 코드에서 revert가 발생하는 경우에는 Ethers.js의 경우 "CALL_EXCEPTION"이라는 에러코드가 반환됩니다. 이러한 정보는 Web3 Provider와 사용하는 라이브러리마다 차이가 있으므로 상황에 따라 맞추어야 합니다.

보통 컨트랙트의 함수가 상태를 변경하는 트랜잭션의 경우에는 리턴 값은 트랜잭션 정보일 뿐이며, 블록에 저장되어 블록체인에 기록되기 전까지는 실제 결과를 알 수 없습니다. 트랜잭션의 최종 결과는 "영수증"에 해당하는 리시트(receipt)로 전달되는데 애플리케이션에서는 리시트의 로그 항목에 있는 이벤트를 참조하여 결과를 확인합니다.

mint가 정상적으로 처리된 경우 ERC-721 표준에 의해서 Transfer 이벤트가 발생해야 합니다. 따라서 애플리케이션에서는 Transfer 이벤트를 확인하여 해당 토큰이 생성되었음을 알 수 있습니다. 일종의 이벤트 구독(subscription)이 필요한 시점입니다. Ethers.js의 Provider 객체는 다음과 같은 형식으로 이벤트 리스너(listener)를 만들 수 있습니다.

```
provider.on( eventName , listener )
```

이벤트 구독은 지갑을 통해서 하기 보다는 인퓨라 또는 알케미의 이더리움 게이트웨이 서비스를 이용하는 것이 좋습니다. Ethers.js에서는 JsonRpcProvider나 WebSocketProvider를 사용할 수 있는데 권장사항은 defaultProvider입니다. 예를 들면 다음과 같이 이벤트 전용 Provider를 만들 수 있습니다.

```
const defaultProvider = ethers.getDefaultProvider(<RPC endpoint>);
```

<RPC endpoint>에 인퓨라 또는 알케미가 제공하는 API 키가 포함된 URL을 넣으면 됩니다. 예를 들어 알케미의 경우는 아래와 같습니다.

```
const RPC_ENDPOINT = "https://eth-rinkeby.alchemyapi.io/v2/bMk...vMCb";
const defaultProvider = ethers.getDefaultProvider(RPC_ENDPOINT);
```

defaultProvider는 하위 컴포넌트에서 사용할 수 있도록 최상위 컴포넌트인 App.js에서 만들어서 컨텍스트 API로 전달하도록 하겠습니다.

이제 <AlsList/>에서 이벤트 리스너를 작성하도록 하겠습니다. 이벤트를 구독하는 것은 외부와 연결하여 데이터를 조회하는 것으로 useEffect()에서 수행합니다. 컨트랙트가 있어야 이벤트 구독의 의미가 있으므로 alsNft를 의존성 배열에 넣습니다.

```
import React, {Fragment, useContext, useEffect, useState} from 'react';
import {
  Button, ButtonGroup,
  Card,
  CardActions,
  ImageList,
  ImageListItem
} from "@mui/material";

import {useParams, useNavigate } from "react-router-dom";
import {isEmpty} from "lodash";
import Metadata from "./Metadata";
import {getTokenURI, isMintedItems, saveMintedItems} from "../utils";
```

```

import { Context } from "../App";
import { ethers } from "ethers";
import artifact from "../contracts/ALSnft.json";

...
function AlsList() {

  let startIndex = 0;
  let list = [];

  ...

  const alsNft = useContext(Context).alsNft;
  const defaultProvider = useContext(Context).defaultProvider;

  const [minted, setMinted] = useState("");

  ...

  const next = localStorage.getItem("ALS.next");
  localStorage.setItem("ALS.next", startIndex+ITEMS_PER_PAGE);

  ...

  useEffect(() => {

    if (alsNft !== null) {

      const alsNftInterface = new ethers.utils.Interface(artifact.contracts.ALSnft.abi);

      defaultProvider.on({address: alsNft.address}, (logs) => {
        const result = alsNftInterface.parseLog(logs);
        saveMintedItems(result.args tokenId.toString());
        setMinted(result.args tokenId.toString());
      });

    } else {
      console.log("NULL");
    }
  }, [alsNft]);

  ...
}

```

컨트랙트에서 발생하는 이벤트를 구독하는 것이므로 alsNft가 생성된 후에 하는 것이 바람직합니다. 이벤트 구독은 Provider의 on을 사용하여 대상이 되는 컨트랙트 주소와 블록의 위치를 지정합니다. 여기서는 컨트랙트 주소만을 지정합니다. 이 부분을 조금 더 자세히 살펴보면 아래와 같습니다.

```

const alsNftInterface = new ethers.utils.Interface(artifact.contracts.ALSnft.abi);

defaultProvider.on({address: alsNft.address}, (logs) => {

  const result = alsNftInterface.parseLog(logs);
  saveMintedItems(result.args tokenId.toString());
  setMinted(result.args tokenId.toString());

});

```

컨트랙트의 주소는 alsNft.address를 가져오고 이벤트 필터로 사용하면 됩니다. 이벤트가 발생하면 콜백으로 이벤트 로그 데이터가 전달됩니다.

리턴 받은 이벤트 로그는 복잡한 구조로 되어 있어서 직접 그것을 처리하기 보다는 Ethers.js에서 제공하는 함수를 이용하는 것이 좋습니다. Interface라는 객체로 이더리움의 이벤트를 디코딩할 것입니다. Interface는 말 그대로 컨트랙트의 ABI를 사용하여 컨트랙트가 제공하는 인터페이스, 즉 함수나 이벤트에 대한 정보를 생성합니다. 이렇게 만들어진 정보를 바탕으로 parseLog메소드로 이벤트 로그의 데이터를 쉽게 해독할 수 있습니다.

parseLog의 리턴 값은 Transfer 이벤트가 전달해주는 세 가지인데, ERC-721 표준에 따르면 from 계정과 to 계정 그리고 토큰 번호입니다. 최초 발행에서는 from 계정은 0x0이 되고 to 계정과 토큰 번호는 mint() 함수를 호출하면서 전달한 값들을 다시 받는 것입니다.

saveMintedItems() 함수는 발행된 토큰 번호를 웹브라우저의 localStorage에 저장하는데, 페이지를 다시 열었을 때 이미 발행된 토큰들의 정보를 유지하기 위해 사용합니다.

```
const saveMintedItems = (tokenId) => {

  const previousMinted = localStorage.getItem("ALS.minted");

  if (previousMinted === null) {
    localStorage.setItem("ALS.minted", tokenId);
  } else {
    const mintedArray = previousMinted.toString().split(",");
    if (mintedArray.indexOf(tokenId.toString()) < 0) {
      localStorage.setItem("ALS.minted", previousMinted + "," + tokenId);
    }
  }
  console.log(`Saved ${tokenId}`);
}
```

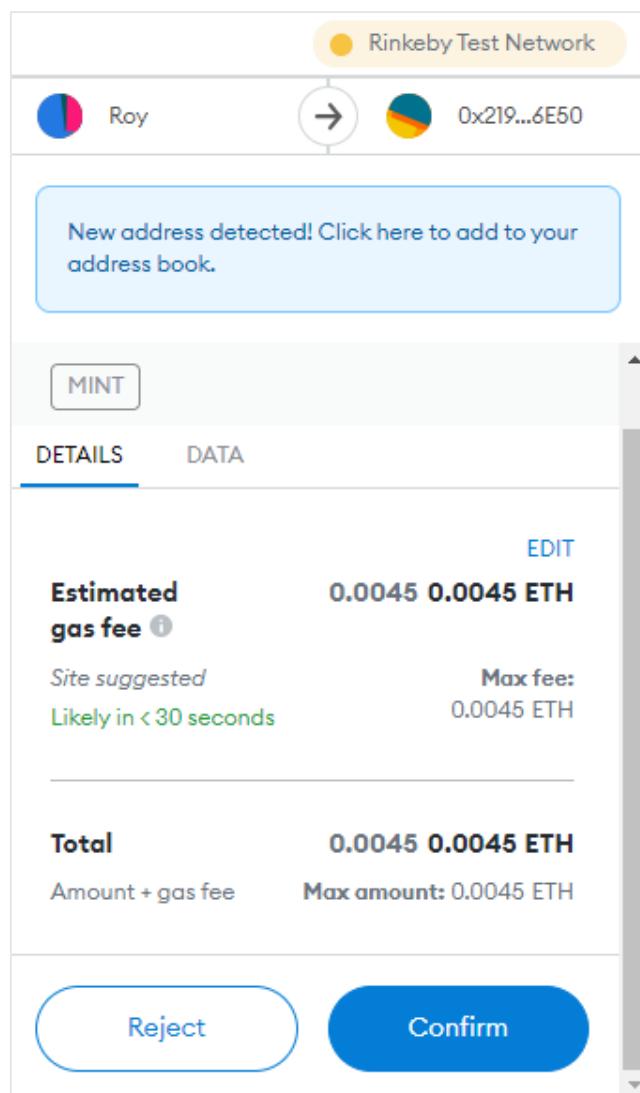
발행된 토큰의 경우에는 NFT 카드에서 MINT 버튼이 나오지 않도록 합니다(물론 이미 발행된 토큰을 다시 발행하더라도 컨트랙트에서 오류가 나게 되어 있습니다).

```
const item = (i) => (<ImageListItem key={i}>
  <Card sx={{maxWidth: "180px"}}>
    <Metadata id={i+1}/>
    <CardActions>
      {
        isMintedItems(i+1)
        ?
        ""
        : <Button color="primary" variant="contained"
          onClick={handleMint} tokenid={i+1}>mint</Button>
      }
    </CardActions>
  </Card>
</ImageListItem>)
```

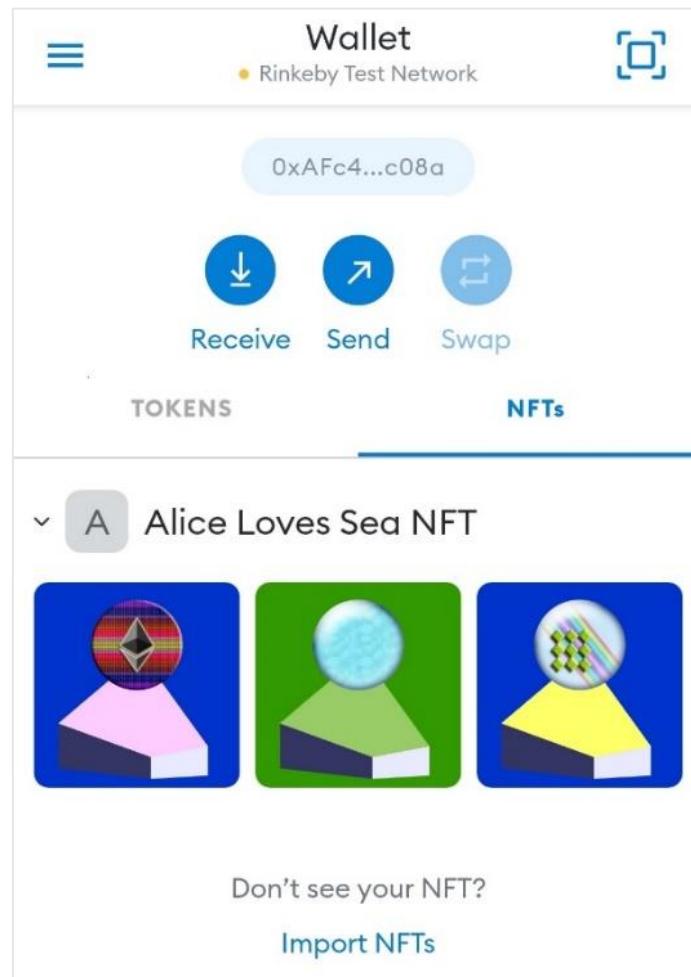
```
<Button color="primary" variant="contained" onClick={handleSale} tokenId={i+1}>
    sale
</Button>
</CardActions>
</Card>
</ImageListItem>
```

isMintedItems() 함수는 localStorage를 읽어서 발행된 토큰인지를 판별하는 함수입니다(utils 폴더에 작성되어 있습니다).

이제 지갑을 연결하고 MINT를 클릭하여 ALS 토큰을 발행하겠습니다. 가스비를 내야 하므로 Rinkeby의 테스트 이더가 있어야 합니다. 토큰 발행 계정과 컨트랙트 배포 계정의 동일 여부를 검사하는 modifier가 적용되어 있으므로 반드시 배포 계정으로 트랜잭션을 전송해야 합니다.



발행이 된 후에는 이더스캔에서 확인할 수 있고 또 메타마스크 모바일에서는 ALSnft 컨트랙트의 주소와 토큰 번호를 넣으면 NFTs 목록에 NFT 이미지들이 표시될 것입니다.



이 예제에서 작성된 ALS 컨트랙트는 이더스캔에서 찾아볼 수 있습니다.

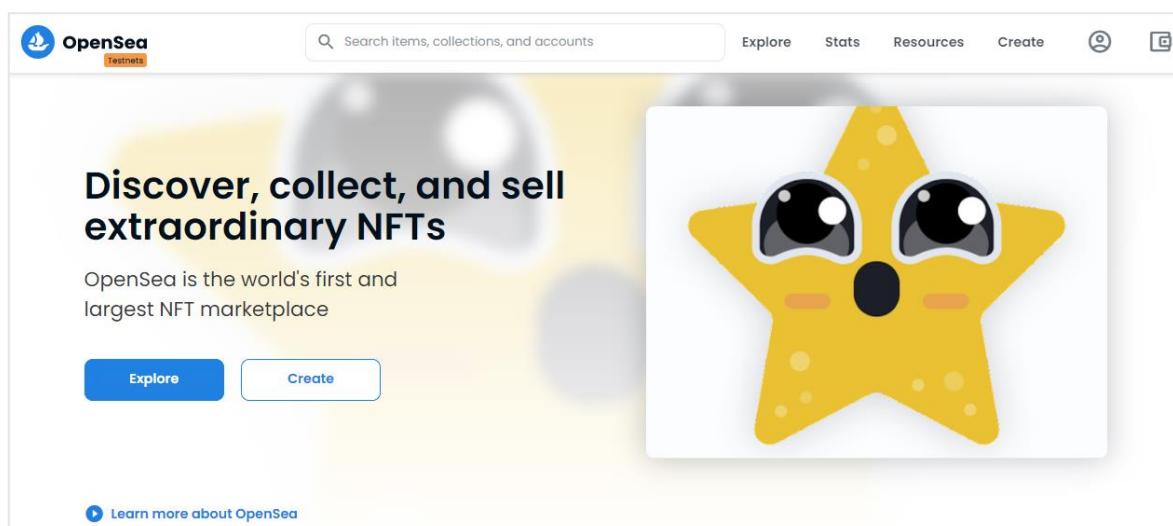
<https://rinkeby.etherscan.io/address/0x219EA9d770aDC079259f6811BF49E2EE600a6E50>

6.6. OpenSea 게시하기

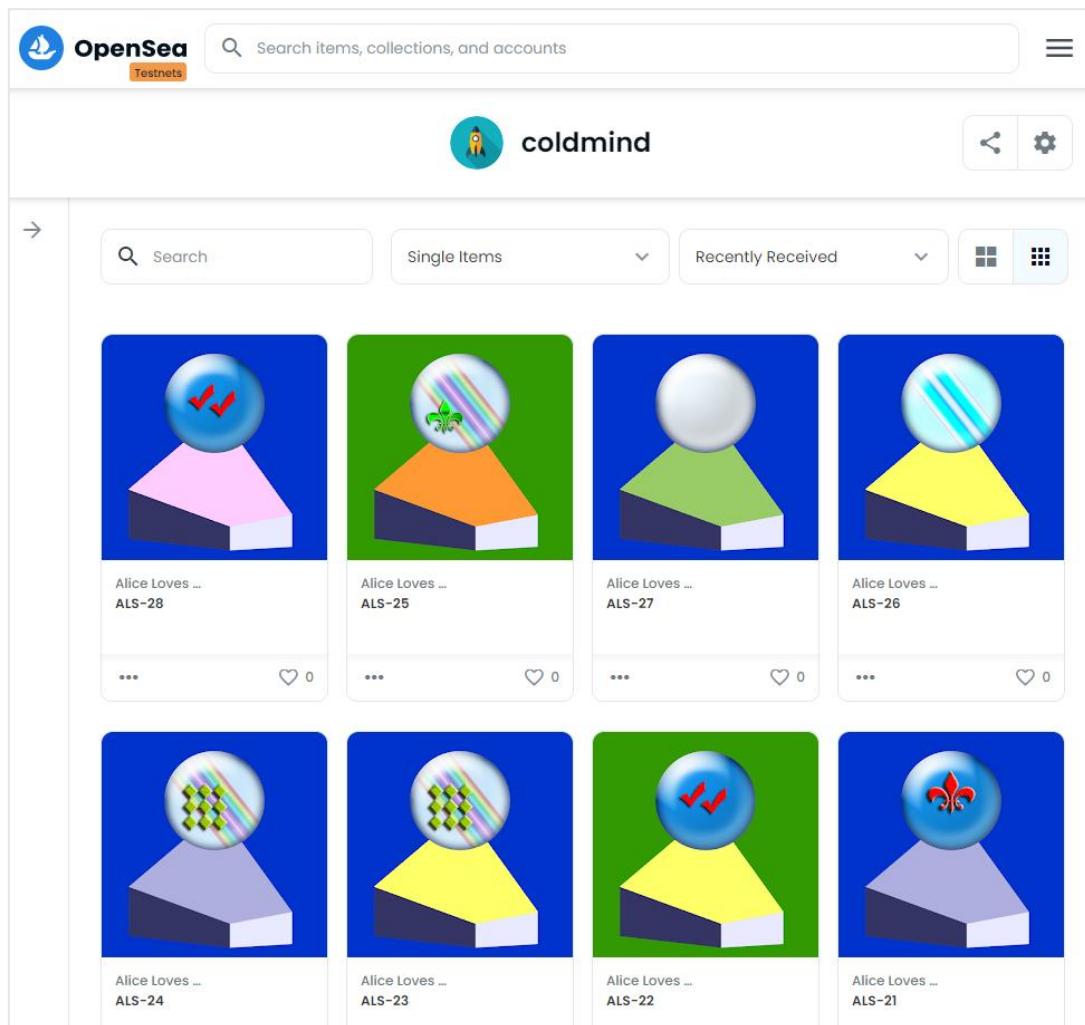
현재 발행된 "Alice Loves Sea", NFT들은 전부 컨트랙트를 배포한 계정이 소유하고 있습니다. 이것을 다른 사람에게 팔기 위해서는 NFT 거래소, 소위 말하는 "마켓플레이스"가 있어야 합니다. 마켓플레이스를 직접 만들 수도 있지만 사람들이 많이 이용하는 기존의 NFT 오픈마켓을 이용할 수도 있습니다. 현재 가장 많은 거래가 이루어지는 곳 중 하나는 OpenSea(<https://opensea.io>)입니다.

OpenSea는 회원 가입 절차 없이 메타마스크와 같은 지갑으로 즉시 로그인할 수 있습니다. ALS는 Rinkeby에 배포되어 있으므로 Rinkeby OpenSea에 접속해야 합니다. 테스트넷과 연동되는 OpenSea는 실제 거래가 이루어지는 곳이 아니라 어디까지나 테스트 용도로 사용되는 것으로 주의해야 합니다(특히 진짜 이더를 전송하지 않도록 주의합니다).

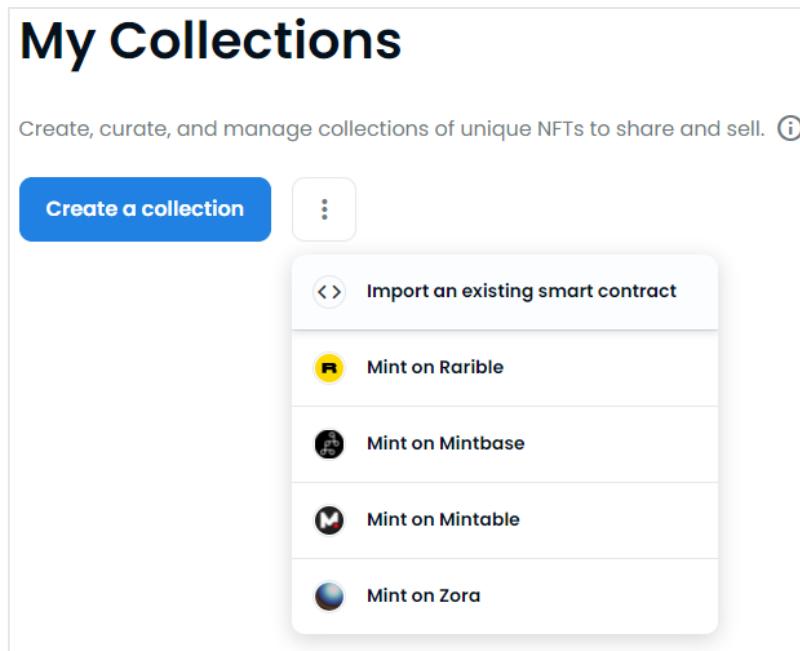
<https://testnets.opensea.io>



메타마스크로 OpenSea에 접속한 적이 있다면 프로필에 본인이 소유한 NFT들이 표시될 것입니다. ALS 역시 자동으로 수집품 목록에 나타납니다.



또는 Collection을 만들어서 이미 배포된 NFT 컨트랙트를 연결시킬 수도 있습니다. 메뉴에서 My Collections를 클릭하고 Create a collection 옆에 있는 메뉴를 열어보면 컨트랙트를 가져올 수 있는 "import an existing smart contract"를 선택할 수 있습니다.

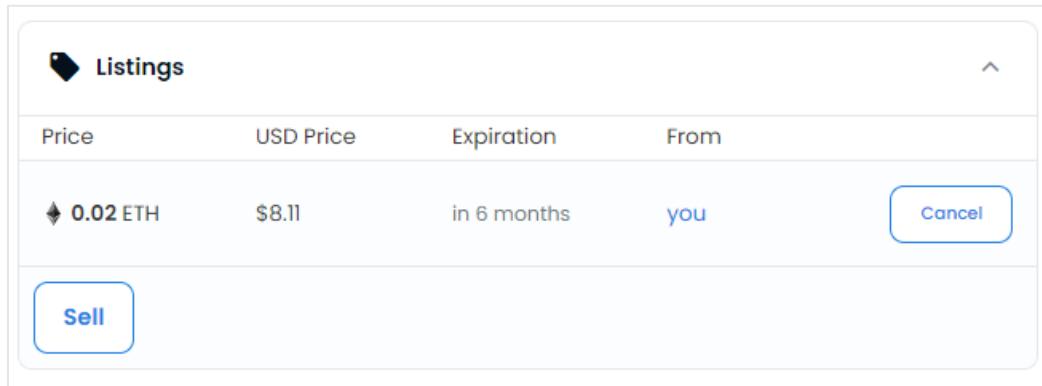


Collection 목록에 NFT들이 표시되고 각 NFT들을 클릭해보면 IPFS에 업로드한 메타정보 그대로 속성들이 나타나는 것을 알 수 있습니다.

A screenshot of an NFT item details page. At the top, there is a section titled 'Description' with the text 'Created by you' and 'ALS::Alice Loves Sea NFT'. Below this is a section titled 'Properties' with three cards: 'BACKGROUND Blue' (50% have this trait), 'BODY Pinkbody' (18% have this trait), and 'FACE Heart' (3% have this trait). There are also sections for 'About Alice Loves Sea NFT' and 'Details', which include information like Contract Address (0x219e...6e50), Token ID (28), Token Standard (ERC-721), and Blockchain (Rinkeby).

ERC-721 표준에 맞추어 작성되었기 때문에 별다른 작업 없이도 데이터가 표시될 수 있는 것입니다. NFT의 각 속성값들은 메타정보를 바탕으로 만들어지고 현재 발행된 토큰을 기준으로 해당 속성의 비율이 표시됩니다.

OpenSea에서 제공하는 판매 기능을 통해 NFT를 판매 상태로 전환할 수 있습니다(이것을 Listing이라고 표현합니다).



Sell 버튼을 클릭하여 판매 방식과 가격 그리고 판매 기간을 입력할 수 있습니다. OpenSea에 납부할 서비스 수수료는 판매 금액의 2.5%로 책정됩니다.

The screenshot shows the "List item for sale" form on the OpenSea platform. On the left, there are several input fields and options:

- Type**: A radio button group between "\$ Fixed Price" and "Timed Auction".
- Price**: A dropdown menu set to "ETH" and a text input field containing "0.02". Below this is a note: "\$8.11 Total (\$8.11 each)".
- Duration**: A dropdown menu set to "6 months".
- Fees**: A section showing "Service Fee" at 2.5%.

On the right, there is a "Preview" section showing a thumbnail of the NFT (a green triangle with two hearts) and its details:
Alice Loves Sea NFT
ALS-2
Price: 0.02

At the bottom left of the form is a large blue "Complete listing" button.

6.6.1. opensea-js

OpenSea는 외부 애플리케이션과 연결될 수 있도록 API와 SDK를 제공합니다. OpenSea SDK를 사용하려면 원래 API 키를 발급받아야 합니다. API 키는 아래 링크를 통해 요청하면 됩니다.

<https://docs.opensea.io/reference/request-an-api-key>

그런데 API 키를 넣지 않아도 사용은 가능합니다(호출 회수에 제약이 따릅니다). OpenSea SDK는 자바스크립트 패키지로 되어 있으므로 npm이나 yarn으로 설치할 수 있습니다. app 디렉토리에서 설치합니다.

<https://www.npmjs.com/package/opensea-js>

```
yarn add opensea-js
```

사용법은 간단합니다. 현재 발행된 ALS-5 토큰의 정보를 조회하려면 seaport.api.getAsset()을 사용할 수 있습니다.

```
import { OpenSeaPort, Network } from "opensea-js";

const seaport = new OpenSeaPort(defaultProvider, {
  networkName: Network.Rinkeby,
  apiKey: ""
});

const asset = await seaport.api.getAsset({tokenAddress: alsNft.address, tokenId: "5",});
```

SALE 버튼을 클릭하면 현재 OpenSea에 게시된 토큰의 정보를 API를 통해 조회해서 팝업에 보여주기로 하겠습니다. MUI의 <Popover/> 컴포넌트를 사용하기로 합니다. AlsList.js에 다음을 추가하도록 합니다.

```
import {
  Button, ButtonGroup,
  Card,
  CardActions,
  ImageList,
  ImageListItem,
  Popover, Typography
} from "@mui/material";

...

const [anchorEl, setAnchorEl] = useState(null);
const [text, setText] = useState("");
```

```

const open = Boolean(anchorEl);

...

const handleClose = () => {
    setText("");
    setAnchorEl(null);
}

const item = (i) => (<ImageListItem key={i}>
    <Card sx={{maxWidth: "180px"}}>
        <Metadata id={i+1}/>
        <CardActions>
            {
                isMintedItems(i+1)
                ?""
                    :<Button color="primary" variant="contained"
                        onClick={handleMint} tokenid={i+1}>mint</Button>
            }

            <Button color="primary" variant="contained" onClick={handleSale} tokenid={i+1}>
                sale
            </Button>
            <Popover
                open={open}
                anchorEl={anchorEl}
                onClose={handleClose}
                anchorOrigin={{
                    vertical: 'bottom',
                    horizontal: 'left',
                }}>
                <Typography
                    style={{ paddingTop: 20, paddingRight: 20,
                            paddingBottom: 20, paddingLeft: 20 }}>
                    {text}
                </Typography>
            </Popover>
        </CardActions>
    </Card>
</ImageListItem>

```

<Popover/>는 바로 위에 위치한 버튼을 기준으로 표시되므로 SALE <Button/> 아래에 배치합니다. OpenSea API를 호출하고나서 그 리턴 값을 상태 변수 text에 넣고 <Typography/> 컴포넌트를 사용하여 텍스트를 표시할 것입니다.

SALE 버튼을 클릭했을 때 실행되는 handleSale()은 아래와 같이 작성할 수 있습니다. OpenSea의 API를 호출하여 현재 토큰의 상태를 조회합니다.

```

const handleSale = async (e) => {
    if (alsNft !== null) {
        setAnchorEl(e.currentTarget);
        let m = "";

```

```

const tokenId = e.target.getAttribute("tokenid");
const asset = await seaport.api.getAsset({tokenAddress: alsNft.address, tokenId});

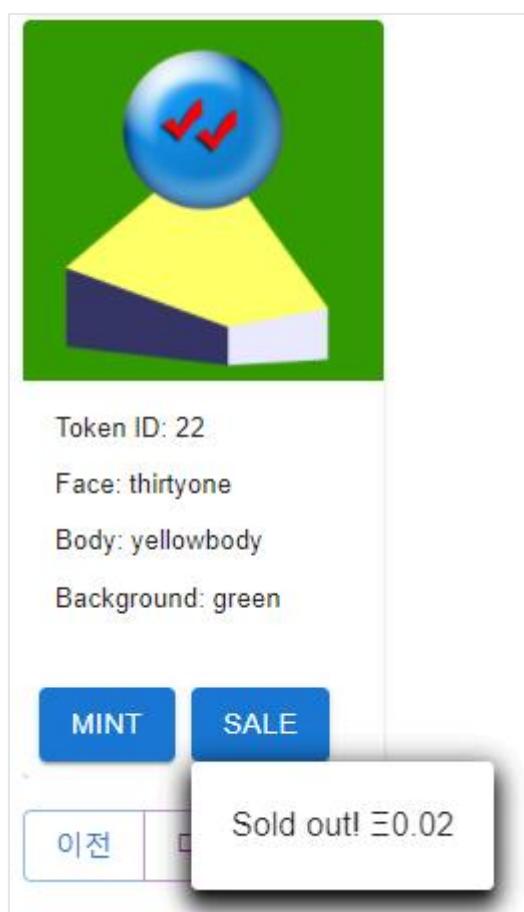
if (asset.lastSale !== null) {
    m = `Sold out! ⚡${ethers.utils.formatEther(asset.lastSale.totalPrice)}`;
} else if (asset.sellOrders.length > 0){
    m = `Listed ⚡${ethers.utils.formatEther(asset.sellOrders[0].basePrice.toString())}`;
} else {
    m = "To be listed";
}

setText(m);
}

```

seaport.api.getAsset() 호출로 받을 수 있는 정보들 중에 lastSale은 해당 토큰이 판매되었을 때 정보를 가지고 있습니다. 그리고 sellOrders는 판매 가격 정보를 리턴합니다. 그 외의 경우는 아직 판매를 위해 게시하지 않은 토큰입니다. 이것을 조건문으로 선별해서 정보를 표시하기로 합니다.

SALE을 클릭하면 아래와 같이 <Popover/>로 정보가 나타납니다.



- 판매 주문(Sell Order) 처리

앞서 OpenSea 내에서 NFT를 판매할 수도 있었는데 API를 이용하여 애플리케이션에서 OpenSea에 판매 주문을 생성하는 것도 가능합니다(판매되거나 구매 제안이 들어오면 이벤트를 수신할 수도 있지만 여기서는 생략하겠습니다).

그런데 OpenSea SDK가 Ethers.js와 호환되지 않는 부분이 있어서 여기서는 Web3.js를 사용하기로 하겠습니다. Web3.js는 다음과 같이 설치합니다(현재 버전은 1.6.1).

```
yarn add web3
```

판매 주문은 OpenSea의 판매 컨트랙트에 기록을 하는 트랜잭션이기 때문에 메타마스크 지갑을 통해 주문이 생성되어야 합니다(가스비 지불). Web3.js에서는 다음과 같이 provider를 생성합니다.

```
const web3Provider = new Web3(window.ethereum);
```

window.ethereum은 웹브라우저 플러그인으로 설치된 메타마스크와 같은 injected 지갑입니다. 앞에서 이미 메타마스크로 애플리케이션에 연결했을 것입니다.

OpenSeaPort() 객체를 만들 때 필요한 provider는 web3Provider.currentProvider를 넣습니다.

```
const seaport = new OpenSeaPort(web3Provider.currentProvider, {
    networkName: Network.Rinkeby,
    apiKey: ""
});
```

OpenSea SDK에서 판매 주문은 createSellOrder()입니다. asset에는 판매할 토큰 번호와 토큰 컨트랙트 주소를 전달합니다. accountAddress는 현재 NFT를 소유한 계정이고 당연히 메타마스크의 전자서명 계정과도 일치해야 합니다.

```
const tokenId = e.target.getAttribute("tokenid");
const tokenAddress = alsNft.address;
const accountAddress = OWNER;
const expirationTime = 0;

const sellOrder = await seaport.createSellOrder({
    asset: { tokenId, tokenAddress },
    accountAddress,
    startAmount: 0.1,
    endAmount: 0.1,
```

```
        expirationTime  
});
```

startAmount와 endAmount는 “더치 경매(Dutch Auction)” 방식에서 필요한 값인데, 처음 판매자가 제시한 가격인 startAmount에서부터 시작하여 조금씩 판매 가격을 내리는 경매입니다. 결국 경매 종료일(expirationTime)에 이르면 판매 가격은 endAmount까지 내려가게 됩니다. 경매 종료일(expirationTime)은 유닉스 타임스탬프(초)로 설정합니다.

경매가 아닌 고정가로 판매하는 경우는 startAmount와 endAmount를 같은 값으로 설정하고 경매 종료일은 0으로 하면 됩니다. 여기서는 정가 0.1이더로 판매하는 것으로 되어 있습니다.

OpenSea API와 SDK 상세한 사용법은 아래 링크를 참고하면 되겠습니다.

<https://docs.opensea.io/reference/api-overview>

<https://github.com/ProjectOpenSea/opensea-js>

6.7. NFT 발행 방식

이더리움과 같은 퍼블릭 블록체인 상에서 NFT를 발행할 때는 비용이 발생합니다. 예를 들어 어떤 행정의 입장권을 NFT로 만드는 경우에는 한 번에 많은 수량을 발행해야 하고 그에 따른 비용이 상당하기 때문에 부담스러울 수 있습니다.

이더리움의 거래 수수료에 해당하는 가스비는 일정하지 않고 아직까지는 경쟁 방식이기 때문에 이렇게 대량 발행의 경우에는 여러 가지 고민해야 할 문제들이 많습니다.

저렴하고 빠르게 발행하는 방법 중 하나는 흔히 말하는 “레이어2” 솔루션을 사용하는 것입니다. 이렇게 되면 사실상 이더리움이 아닌 레이어2 기술을 제공하는 개발팀의 솔루션에서 NFT를 발행하는 셈이 되므로 특정 레이어2 기술에 종속될 수 있고 또 이더리움 메인넷으로 NFT를 옮기려면 번거로운 작업이 필요할 수도 있습니다.

온체인에서 구현할 수 있는 방식은 4장 오픈제펠린 컨트랙트에서도 본 것처럼 컨트랙트 레벨에서 한 번의 트랜잭션으로 여러 개의 토큰을 발행할 수 있는 함수를 제공하는 것입니다. 기본적으로 내부에서 반복문을 이용하는 것인데 주의할 점은 블록의 최대 가스 제한이 있으므로 한 트랜잭션에 너무 많은 수량을 발행하는 것은 한계가 있다는 것입니다. 레이어2에 비하여 NFT 발행 비용을 획기적으로 줄일 수는 없지만 어느 정도 절약할 수는 있겠습니다. 예를 들어 다음과 같은 형태가 될 수 있습니다.

```
function bulkMint(address[] memory to, uint256[] memory tokenId, string[] memory tokenURI)
    public returns (bool) {

    for (uint i = 0; i < to.length; i++) {
        _mint(to[i], tokenId[i]);
        _setTokenURI(tokenId[i], tokenURI[i]);
    }
    return true;
}
```

자바스크립트에서는 배열 그대로 전달하여 호출하면 됩니다.

```
const tos = ["0x547...eAF", "0xfc...08a", ..., "0x22d...32b"];
const tokenIds = [1, 2, ..., 10];
const tokenURIs = ["https://dweb.link/ipfs/1.json", ..."https://dweb.link/ipfs/10.json"];

await this.instance.bulkMint(tos, tokenIds, tokenURIs);
```

다른 방법으로 “lazy minting”이라는 것이 있습니다. 이것은 원래 OpenSea에서 크리에이터들의 발행 비용의 부담을 덜어주기 위해 채택한 방식으로, NFT를 발행하는 시점을 뒤로 미루는 것입니다. 즉 발행에 필요한 데이터만을 저장해 놓았다가 누군가 그 NFT를 구매할 때 비로소 발행이 되는 방식입니다. 최초 발행자인 크리에이터들이 가스 비용을 내지 않기 때문에 자신이 만든 NFT를 부담 없이 전시하고 판매할 수 있게 됩니다.

6.8. 정리

이번 장에서는 솔리디티 스마트 컨트랙트와 자바스크립트, 리액트를 사용하여 간단한 애플리케이션을 만들어 보았습니다. 컨트랙트는 오픈제페린을 이용해서 비교적 쉽게 ERC-721 컨트랙트를 구현했고 자바스크립트를 사용하여 NFT 생성과 메타정보 업로드 그리고 최종적으로 컨트랙트의 ABI를 결합하여 이더리움과 데이터를 주고받는 애플리케이션을 작성했습니다.

예제에서는 데이터베이스나 백엔드 애플리케이션 없이 웹브라우저의 localStorage를 이용하여 필요한 데이터를 저장했습니다. 또 직접 마켓플레이스를 구현하지 않아도 OpenSea와 같은 대형 NFT 마켓플레이스의 API를 활용하여 NFT를 판매할 수 있습니다.

EOF

감사합니다.

Don't trust, verify!