# Chapter 7. 맛보기 웹디앱 프로젝트

어느 프로그래밍 언어나 처음에는 Hello를 개발해본다. Solidity 언어로 블록체인에 스마트 컨트랙 Hello의 맛보기 프로젝트를 만들고, 웹 클라이언트에서 자바스크립트로 호출까지 전 과정을 개발해보게 된다. 잇달아 맛보기로서 복잡하지 않은 Greeter, Counter를 구현하면서 '아! 블록체인 앱은 이렇게 만드는구나'를 이해하도록 하자.

# 1. Hello World 디앱

어떤 언어를 배우든 처음으로 "Hello World"를 출력해보게 된다. 간단하지만 코딩, 컴파일, 실행, 테스트의 전과정을 경험하면서 그 언어의 특징을 보다 잘 이해하게 된다.

블록체인 프로그래밍은 코딩, 컴파일, 테스트의 절차는 필수적이고, C 또는 자바의 그것과 비슷하고 별로 다르지 않다. 단계별로 하는 작업을 설명하면:

* 첫째, 코딩은 Solidity 언어로 작성하고,
* 둘째로 소스코드는 컴파일해서 ABI(Application Binary Interface)와 바이트 코드(Byte code)를 생성한다. ABI는 컨트랙을 사용하기 위해 만들어지는 인터페이스 표준으로, JSON으로 표현된다. 또한 바이트 코드는 Solidity 소스코드를 컴파일해서 생성된 것으로 이더리움 가상 기계에서 실행된다.
* 세째로 컴파일한 다음에는 블록체인에 올려야 하는데, 이 배포 작업은 독특하다. 배포하고 나면 분산 블록체인의 메모리 번지같은 컨트랙 주소가 주어진다,
* 마지막으로, 방금 배포하고 얻은 컨트랙 주소를 통해 블록체인의 API를 비로서 사용할 수 있게 된다. 이른바 거래라고 하는데, 블록체인이라는 특성에 따라 디지털 서명을 하고, 마이닝하고, 블록체인으로 조립된다.

이런 절차를 다음 표에 요약해 놓았는데, 앞으로 이런 절차에 따라 프로그래밍이 수행될 것이다.

|  |  |  |
| --- | --- | --- |
| 단계 | 절차 | 설명 |
| 단계 1 | 개발 | 컨트랙의 **소스 코드** 개발 |
| 단계 2 | 컴파일 | 소스코드를 컴파일해서 **ABI**와 **바이트 코드** 생성 |
| 단계 3 | 컨트랙 배포 | 컴파일해서 얻은 ABI, 바이트 코드로 객체를 생성한 후 배포 요청, 마이닝해서 블록체인의 **컨트랙 주소(contract address)** 획득 |
| 단계 4 | 사용 | **ABI**, **컨트랙 주소**로 컨트랙 객체를 생성한 후, call() 또는 send() 함수로 API 호출. |

## 7.1 단계 1: 컨트랙 개발

코드를 작성하기 위해서 REMIX를 사용해보자. 단순한 편집기를 이용해서 컨트랙 코드를 작성해도 되지만, REMIX는 잘못 입력하면 알려주기도 하는 등 편리한 기능을 제공한다.

**REMIX**는 온라인 http://remix.ethereum.org/ 에서 제공되는 Solidity 통합개발환경 (IDE, Integrated Development Environment)이다. HTTPS로 연결할 경우에는, RPC 원격접근이 필요할 때 HTTP만 허용되지 않도록 주의한다 (메뉴에 보이는 'External http server'를 클릭하면, REMIX에서 팝업으로 설명을 띄우고 있으니 주의해서 읽어보자.)

통합개발환경으로서 REMIX는 컴파일, 디버깅, 편집, 배포, 테스트의 절차를 지원한다. 온라인에서 사용하는 것이 느리거나 불편하다면, https://github.com/ethereum/remix-live/tree/gh-pages 로 가서 zip 압축 파일을 내려받아 자신의 컴퓨터에서 실행하거나(index.html을 웹브라우저로 열기) 또는 https://github.com/ethereum/remix-desktop/releases/ 에서 본인 운영체제에 적합한 실행 파일을 내려받아 설치할 수 있다.

또한 github에서 제공하는 짧은 코드나 메모 등을 저장하고 공유할 수 있는 gist 서비스와 연동할 수 있다. &gist=여기에 id를 URL에 넣어서 전송하여, gist로 소스코드를 내보내거나 읽을 수 있다.

Solidity가 처음이라 명령어가 낯설겠지만, 객체지향언어이므로 클래스를 코딩하는 것처럼 시작하도록 하자. 기억하자. 지금은 간단한 프로그램을 만들고 배포, 실행하는 전과정을 완수하기 위해서 해보는 코딩 작업이다. 천천히 따라해보고, 명령어는 차츰 배워가자.

REMIX에서 코드에 오류가 없는지 테스트해보고 요구 사항대로 작동하면, 코드를 복사해서 sol 파일로 저장한다.

[파일명: src\Hello.sol]  
pragma solidity ^0.8.0;  
contract Hello {  
 function sayHello() pure public returns(string memory) {  
 return "Hello World";  
 }  
}

코드를 설명해보자.

|  |  |
| --- | --- |
| 줄 | 설명 |
| 1 | 버전 0.8.0이상 가운데, 최신 버전으로 컴파일. 앞의 햇(Hat, ^) 표시가 그런 의미이다. |
| 2 | 컨트랙 명칭을 Hello로 정한다. |
| 3 | sayHello() 함수, 몇 수식어가 쓰이는데 차츰 배운다. pure는 단순 조회, public은 누구나 호출 가능, string으로 지역변수 값을 반환한다. |
| 4 | 문자열 "Hello World" 반환. |

## 3.2 단계 2: 컴파일

다음 2단계는 컴파일이다. 그 결과 (1) ABI와 (2) 바이트 코드가 생성되는데, 이들은 다음 단계에서 필요로 한다.

컴파일하기 위해서는 (1) REMIX의 컴파일러 또는 (2) 직접 solc 컴파일러를 사용하면 된다. 여기서는 모두 배워보자. 우리는 두 번째 방법을 더 빈번하게 쓰게 된다. REMIX에서 컴파일하면 ABI, 바이트 코드를 복사해야 하지만, 직접 컴파일을 하면 그렇지 않아서 매우 편리하다.

### 첫 번째 방법: REMIX 컴파일

REMIX에서 버전 0.8.0으로 컴파일 해보자.

* Hello.sol을 코딩한 후,
* 메뉴 Compile 버튼을 누른다. 오류가 있다면 물론 코드를 수정해야 한다.
* Compilation Details 버튼을 누르고 WEB3DEPLOY를 선택하면 ABI, 바이트 코드가 포함된 코드를 발견할 수 있다. 우리가 사용하는 **web3.js 버전과 일치하는 문법으로 작성되었는지** 확인하자.

### 두 번째 방법: solc 컴파일

REMIX를 사용하지 않고, 터미널 환경에서 Solidity 컴파일러를 실행하여 ABI와 바이트 코드를 생성할 수도 있다. 버튼을 클릭하면 되는 REMIX와 달리, 타이핑을 해야 하니 불편하기도 하겠다.

먼저 컴파일러가 설치되었는지 확인하자. 최근 버전의 컴파일러 파일 이름은 solc-windows.exe이다. 혹시 운영체제와 그 버전에 따라 명령어가 solc일 수 있다. 내려받아 설치하면서 solc(또는 solc.exe)인지 solc-windows.exe인지 분별하여 사용하자. 명령창 프롬프트의 프로젝트 디렉토리 pjt\_dir>에서 확인해보자.

pjt\_dir> solc-windows --version 최근 윈도우 버전의 컴파일 명령어  
solc, the solidity compiler commandline interface  
Version: 0.8.1+commit.df193b15.Windows.msvc

컴파일을 해보자. 명령어에 --abi, --bin, --gas 스위치를 주고 있는데, 출력을 확인해보자.

pjt\_dir> solc-windows.exe --abi --bin --gas src\Hello.sol  
======= src/Hello.sol:Hello =======  
Gas estimation:  
construction:  
 123 + 76000 = 76123  
external:  
 sayHello(): infinite  
Binary:  
608060405234801561001057600080..생략..0008010033 ---> (A) 복사해서 아래 (X)에 붙여 넣는다.  
  
Contract JSON ABI  
[{"inputs":[],"name":"sayHello","outputs":[{"internalType":"string","name":"","type":"string"}],"stateMutability":"pure","type":"function"}] ---> (B) 복사해서 아래 (Y)에 붙여 넣는다.

* 필요한 Gas를 출력하고 있다. construction은 배포하면서 객체를 생성하는 비용이다. external에서 sayHello(): infinite는 반드시 무한대 gas가 필요하다는 의미는 아니다. 예를 들어, 바이트 코드에 backward jump 문이 포함되어 있거나 하는 경우 무한대로 계산된다.
* Binary 다음에는 바이트 코드가 출력 되었다. 여기에서는 불필요해서 중략한다.
* ABI는 컨트랙을 사용하는데 필요한 인터페이스이고 JSON(JavaScript Object Notation) 형식으로 작성된다. 읽어보면 sayHello() 입력 inputs, 출력 outputs를 정의하고 stateMutability가 pure, view, non-payable, payable 인지 정의하고 있다. 과거에는 constant, payable이 사용되었지만 지금은 제거되어 stateMutability로 대체된다.

### 다른 언어나 도구들에 내장된 컴파일러

geth 1.6이전에는 자체 컴파일러를 제공했지만, 중단되었다. 또는 node.js나 파이썬에서 제공하는 Solidity 컴파일러들도 있고, 파이썬 쉘(Python Shell)처럼 Solidity 언어를 한 줄씩 실행할 수 있는 Solidity REPL(Read-Eval-Print Loop)도 있다. 하지만 이 책에서는 다루지 않는다.

## 3.3 단계 3: 배포

컨트랙 배포의 목적은 주소이다. 컴파일하고 생성된 ABI와 바이트 코드를 배포한 후, 배포된 컨트랙의 블록체인 주소를 얻는 것이다.

몇 가지 세부적인 절차로 나누어 볼 수 있다.

* (3-1) 앞 2단계의 컴파일을 하고 얻은 **ABI, 바이트 코드를 사용해서 컨트랙 객체를 생성**하고,
* (3-2) 그 객체의 배포를 deploy() 함수로 요청하여 transactionHash를 얻고,
* (3-3) 마이닝을 하고 나면 블록체인에서의 컨트랙 주소 contractAddress를 받게 된다.

### 단계 3-1: 컨트랙 생성

객체 생성은 new() 명령어로 한다. 이 때 ABI가 필수적으로 필요하고, address, options은 선택적으로 사용할 수 있다.

new web3.eth.Contract(abi [, address] [, options])

* abi: 컨트랙에 대한 JSON 인터페이스
* address: 컨트랙이 배포된 블록체인 주소를 나타낸다. 처음 배포한다면 당연히 주소가 없으므로 비워놓는다. 배포된 컨트랙을 갱신하는 것이라면 해당 주소를 적어주면 된다.
* options: from, gasPrice, gas, data (컨트랙의 바이트코드)

코드로 예를 들어보자. 아래 코드에서 인자로 쓰이는 \_abi, \_bin 변수는 아직 값을 가지고 있지 않아서 오류가 발생하게 된다. \_bin은 16진수로 만들기 위해 앞에 '0x'를 붙이고 있다.

node> var myGreeter = new web3.eth.Contract(\_abi, {data:'0x'+\_bin});

### 단계 3-2 배포요청

#### deploy()

컨트랙을 배포하려면 deploy() 함수를 이용한다.

myGreeter.deploy(options)

options는 data와 arguments로 구성한다.

* data: 컨트랙의 바이트 코드를 넣는다.
* arguments: 컨트랙 생성자에 인자를 입력할 수 있는데, 그 인자를 넣고, 없으면 생략한다.

#### send()

컨트랙은 send() 함수를 이용해서 배포하는데, 블록체인에 전송하는 기능을 한다.

send(options [, callback])

* options에는 from, gasPrice, gas, value를 적을 수 있다.
* callback함수는 생략 가능한 인자이다. callback 함수가 전달되면, send() 코드를 실행한 후, 실행이벤트 ('transactionHash' 등)에 따라 callback 함수가 호출된다. 첫 인자는 'error'를 넣어주어야 한다. callback 함수는 **PromiEvent를 반환**한다. PromiEvent에 on, once, off 함수를 연결(chaining)하여 이벤트를 처리할 수 있다.

### 비동기함수에 대한 Promise Events

web3.js에서의 함수는 Promise를 반환한 다음 연결 함수 on, once, off를 통해 이벤트 event를 발생시킬 수 있다. 사용할 수 있는 PromiEvent는:

* transactionHash: 거래가 전송된 직후 transction hash가 발생하는 이벤트 발생
* receipt: transactionReceipt이 만들어지는 이벤트 발생
* confirmation 매 24번째 confirmation 마다 이벤트 발생
* error: 오류에 대해 이벤트 발생

다음 코드는 이벤트가 발생하면 그 이벤트를 출력하는 명령문이다.

web3.eth.sendTransaction({from: '0x123...', data: '0x432...'})  
 .once('sending', function(payload){ ... })  
 .once('sent', function(payload){ ... })  
 .once('transactionHash', function(hash){ ... })  
 .once('receipt', function(receipt){ ... })  
 .on('confirmation', function(confNumber, receipt, latestBlockHash){ ... })  
 .on('error', function(error){ ... })  
 .then(function(receipt){  
 // 마이닝이 끝나면 실행되는 코드  
 });

### 단계 3-3 주소 출력에 필요한 then()

컨트랙을 배포하는 이유는 무엇일까? 블록체인 주소를 알아내는 것이다. 블록체인에 배포된 주소가 있어야, 그 주소를 통해 그 컨트랙을 호출할 수 있기 때문이다. 그러나 deploy() 함수는 비동기적으로 실행되므로, 배포한 후 바로 주소가 주어지는 것이 아니라 Promise가 반환된다. 이를 해소해야만 주소를 얻을 수 있게 된다.

### 3단계 실행에 필요한 코드 종합: Web3 1.0 이상 버전

web3의 어떤 버전을 사용하는지 주의한다. 버전이 다르면 문제가 된다. web3.js 1.0 이상의 버전으로 실행이 되도록 작성한다. 실행하기 전, geth를 띄워놓아서 코드를 실행하면 어떤 변화가 발생하는지 지켜보는 것이 좋다. 포트 번호만 다르게 해서 ganache도 같이 띄워 놓자.

#### 계정 읽기

배포하려면 계정 주소가 필요하다. 계정을 읽어 보자. **geth console에서는 배열 인덱스로 가능**하다. **node.js에서 실행했을 경우는 배열 인덱스가 더 이상 지원되지 않는다**.

node> var address0 = web3.eth.accounts[0]; //이렇게 읽어올 수 없다.

web3.js 1.0 이상에서는 비동기 방식으로 계정을 읽어오며, Promise가 반환이 된다. 비동기 방식으로 계정 주소를 넣는 것은 나중에 하기로 하고, 일단 하드 코딩해서 넣자.

[파일명: src\getMyAddr.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"))  
web3.eth.getAccounts().then(console.log);

실행하고 출력된 계정은 자신이 실행시킨 ganache@8345의 것과 일치하는지 확인하자. 눈치채겠지만, 혼란을 피하기 위해 운용중인 포트 번호를 덧 붙여 ganache@8345로 적고 있다.

pjt\_dir> node src\getMyAddr.js  
  
[ '0xEb670169A5fcD2550035588A2ba3b45509703ABd', ---> (C) 주소를 복사해서 아래의 (Z)에 적는다.  
 '0xF724CDbfCf9C94f28547b1111B3A6E3672a9aDCE',  
 ...생략...  
 '0xe04852Cb018778545e75CC21EEC1849DdCdFa79f' ]

#### gas 계산

다음으로 gas도 필요하므로, 계산해보자. 이를 위해서는 data를 입력해 주면 그 바이트 크기와 Opcode에 따라 계산된다. gas를 산정하는 함수는 estimateGas()이다.

node 창을 열어서 한 줄씩 해도 물론 된다. 그러나 코드를 보라! 한 줄씩 입력하기에는 코드가 복잡하고, 복사-붙여넣기를 상당히 해야 한다.

명령어를 파일로 저장하고, 일괄 실행하는 방식으로 하자.

단계 2 컴파일에서 생성된 abi, 바이트 코드와 또한 계정주소도 복사, 붙여넣기를 하고 있다. 불편하지만 일단 그렇게 하자. 무엇을 어디에서 가져와 어디에 입력하는지 이해하는데 도움이 될 것이다. 나중에 코드로 처리하는 방법을 설명하게 된다.

자바스크립트 도움말을 중간에 적어 놓았다. 필요한 (A) 바이트 코드, (B) abi, (C) 주소의 세 항목을 복사해서 붙여야 한다.

[파일명: src\HelloDeployGas.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
// (Y) 여기에 위에서 생성한 abi (B)를 복사-붙여넣는다  
var shelloContract = new web3.eth.Contract([{"inputs":[],"name":"sayHello","outputs":[{"internalType":"string","name":"","type":"string"}],"stateMutability":"pure","type":"function"}]);  
// 계정 address와 data도 복사-붙여넣기 한다. 자신의 개발환경에 따라 계정, 데이터는 값이 다를 수 있다.  
shelloContract.deploy({  
 from: '0xEb670169A5fcD2550035588A2ba3b45509703ABd', <--- (Z) 여기에 위 주소 (C)를 복사해서 붙여 넣는다  
 data: 608060405234801561001057600080..생략..0008010033, <--- (X) 여기에 위 바이트 코드 (A)를 복사해서 붙여 넣는다.  
 }).estimateGas().then(function(myGas) {  
 console.log("Estimated gas: " + myGas);  
 gas = myGas;  
 })  
 .catch(console.error);

실행하면, 컨트랙을 생성하고 배포하는데 소요되는 gas는 120415이다.

pjt\_dir> node src\HelloDeployGas.js  
  
Estimated gas: 120415

실제 금액으로 얼마나 비용이 들게 되는지 계산하려면 gas의 가격을 알아야 한다. getGasPrice()를 이용하면 gas 가격을 확인할 수 있다.

node> web3.eth.getGasPrice().then(console.log);  
1000000000

getGasPrice()의 단위는 wei이다 앞에서 실행한 결과에서 gas 가격은 1000000000 wei이고 1 gwei이다. 이는 0.000000001 Ether에 해당한다.

참고로 사설망에서 gas 가격은 1 gwei로 고정되어 있다.

그러나 실제 거래에 사용되는 평균 가격은 보통 이 보다 높고 변동된다 (참조: https://ethgasstation.info/). 대략 gas와 gasPrice를 곱하고, 한화로 환산하면 약 24,000원이 필요하다. 물론 환전가격은 매일 변동하니까 정확하지 않다. 적지 않은 금액이다. gasPrice가 2배가 되는 경우도 많다. 그렇다면 48,000원으로 껑충 높아진다.

약 120,000 gas \* 0.000000050 gasPrice \* 약 4백만원 (환전가격) = 약 24,000원

게다가 곧 배포된 컨트랙을 사용하기 위해서 별도로 gas를 지불해야 한다. 이 금액을 더한다면 가격은 더 올라가게 된다.

#### ganache 배포 web3 1.20

이제 ganache@8345에 배포해보자. geth@8445는 곧 이어서 나중에 해보자. 앞서 gas 계산에서 하였듯이, abi는 컨트랙 인자로, 바이트 코드는 data, 그리고 지급계정주소(위 getMyAddr실행하고 출력된 목록에서 복사)는 from 필드에 복사해 넣었다. 지금은 복사해서 붙여 넣고 있지만, 나중에 자동으로 가져오도록 수정할 것이다.

[파일명: src/HelloDeploy.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
(Y) 여기에 위에서 생성한 abi (B)를 복사-붙여넣는다  
var shelloContract = new web3.eth.Contract([{"inputs":[],"name":"sayHello","outputs":[{"internalType":"string","name":"","type":"string"}],"stateMutability":"pure","type":"function"}]);  
shelloContract  
 .deploy({  
 data: 608060405234801561001057600080..생략..0008010033, <--- (X) 여기에 위 바이트 코드 (A)를 복사해서 붙여 넣는다.  
 })  
 .send({  
 from: '0xEb670169A5fcD2550035588A2ba3b45509703ABd', <--- (Z) 여기에 위 주소 (C)를 복사해서 여기에 붙여 넣는다  
 gas: '4700000'  
 }, function (error, transactionHash){   
 console.log(error, transactionHash);   
 })  
 .then(function(newContractInstance){  
 console.log(newContractInstance.options.address)  
 });

실행해보자. 위 프로그램의 console.log() 명령문의 transactionHash, address가 출력된다.

pjt\_dir> node src/HelloDeploy.js  
  
null '0xc255f82ab2150858ce0414f2ceaf0f919a64359f405ebded942bd04610f1a2f4' 이 것이 transactionHash  
0x0d3C29cAD3c40497699e13CC34bA099E1fe426Ba 이 것이 배포된 컨트랙 주소

#### geth배포 web3 1.20

ganache@8345에 비해 geth@8445는 지급계좌를 해제하고, 마이닝을 하는 등 추가 작업이 필요하다. 당연히 조금 복잡하겠다.

몇 번 설명하였지만, ganache에서는 계정를 해제해 줄 필요가 없다.

터미널 윈도우를 열고 지급 계정을 해제한다. 그 효과는 시간을 설정하지 않는한 1회에 그친다. 또한 **잔고**가 남아 있어, 거래에 필요한 gas비용을 충당할 수 있어야 한다.

터미널 환경에서 "geth attach http://localhost:8445"를 실행시킨다. geth 커맨드 프롬프트가 나타나면, 다음을 geth 명령행에서 실행시켜 계정을 해제하고 "exit" 명령으로 geth 프롬프트로부터 빠져 나온다.

geth> personal.unlockAccount(eth.coinbase); //또는 eth.accounts[0]  
Unlock account 0x6819f8376ef2da6A01082c7F45E51FF15Fc69cDa  
Passphrase: 여기에 비밀번호를 입력한다.  
true  
> exit

앞 ganache 배포 코드와 거의 동일하게 작성한 HelloDeploy2.js 코드를 실행시킨다. 여기서는 몇 가지 이벤트 처리를 추가해서 다르게 보인다.

코드에서 사용하는 계정은 geth를 실행한 후 획득한 eth.accounts[0]에 있는 것으로 10번째 줄의 "from:"에 입력한다. 만약 계정이 unlock되어 있지 않다면 3번 줄의 주석을 해제하고, 계정 번호와 "password"대신 본인의 비밀 번호를 입력한 후에 실행시켜야 한다. 비밀 번호는 따옴표 안에 입력하면 된다. 이렇게 비밀번호를 코드에 넣는 것은 보안상 권장할 수 없다. 따라서 번거롭더라도 앞에서 했던 것처럼 geth에서 unlock하는 방법을 사용하자.

[파일명: src/HelloDeploy2.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8445"));  
//web3.eth.personal.unlockAccount("0x6819f8376ef2da6A01082c7F45E51FF15Fc69cDa","password",1000).then(console.log('unlocked!'));  
var shelloContract = new web3.eth.Contract([{"constant":true,"inputs":[],"name":"sayHello","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"pure","type":"function"}]);  
shelloContract  
 .deploy({  
 data: 608060405234801561001057600080..생략..0008010033, <--- (X) 여기에 위 바이트 코드 (A)를 복사해서 붙여 넣는다.  
 })  
 .send({  
 from: '0xEb670169A5fcD2550035588A2ba3b45509703ABd', <--- (Z) 여기에 위 주소 (C)를 복사해서 여기에 붙여 넣는다  
 gas: '4700000'  
 }, function (error, transactionHash){   
 console.log(error, transactionHash);   
 })  
 .on('transactionHash', function(error,transactionHash) { 이 코드는 테스트 용도로 넣는다. 꼭 필요한 코드는 아니다.  
 console.log("hash-- "+transactionHash);  
 })  
 .on('receipt', function(receipt) {  
 console.log('receipt:: '+receipt.contractAddress);  
 })  
 .then(function(newContractInstance){  
 console.log(newContractInstance.options.address)  
 });

지급 계정이 풀렸는지 확인하고 다름 프로그램을 실행해본다. 계정을 해제하지 않고 거래를 요청하면 '불충분 가스' 오류가 생성된다 (Error: Returned values aren't valid, did it run Out of Gas?)

pjt\_dir> node src\HelloDeploy2.js  
  
null '0xa40eaf3185812b93c6a5432288a6efe6f6a1d3c644ca50592d850256afa3729c'  
hash-- undefined  
receipt:: 0xb8770393a75E5B88E7d5cFb5db1068e28B4dA8Ba  
0xb8770393a75E5B88E7d5cFb5db1068e28B4dA8Ba

node로 HelloDeploy2.js를 실행하고 나서, geth@8445 실행 화면의 출력을 확인하자. 로그는 쌓이기 때문에 잘 찾아봐야 한다. transaction이 생성되고 그 hash와 contract (컨트랙 주소를 의미한다)이 로그로 출력 된다. 그 메시지를 여기 복사해 놓으면 다음과 같다.

INFO [01-07|10:34:47.871] Submitted contract creation  
fullhash=0x851d565882f9758d03ef232a5badb6f23629a1f5768db2f3ca2c13e22a1ef40b  
contract=0xd29b9e81388F91658a2120587A0e123c29eC6c36

geth 콘솔에서 대기 거래를 아래와 같이 출력하면, 대기하고 있는 transaction hash 값이 서로 일치한다. 마이닝을 하고 나면 컨트랙 주소가 주어진다.

geth> eth.pendingTransactions  
[{  
 blockHash: null,  
 blockNumber: null,  
 from: "0xEb670169A5fcD2550035588A2ba3b45509703ABd",  
 gas: 4700000,  
 gasPrice: 1000000000,  
 hash: "0x851d565882f9758d03ef232a5badb6f23629a1f5768db2f3ca2c13e22a1ef40b",  
 input: "0x608060405234801561001057600080..생략..0008010033",  
 nonce: 265,  
 r: "0xe995ae41b4193839df768ad629a9f2dc09ec7cfc08d5f42ce1207021d3395835",  
 s: "0x26c380eb4f60e90cbdb54a45fbf691ce37b18b4b3e8603875f3cf5f6e30a0553",  
 to: null,  
 transactionIndex: null,  
 v: "0x66",  
 value: 0  
}]

### 마이닝

배포를 완성하려면 그 거래가 마이닝되어야 한다. 물론 geth@8445에서 배포할 경우에 그렇고, ganache@8345는 마이닝이 불필요하다.

geth> miner.start(1);admin.sleepBlocks(1);miner.stop()

마이닝이 되고 나면 비로서 컨트랙이 블록체인에 배포된다. 따라서 blockNumber가 증가하게 된다.

그리고 컨트랙이 배포된 블록체인의 주소 값이 생성된다. 아래 getTransactioinReceipt(해시주소)를 조회해보면 마이닝이 완료되기 전에는 null이다.

geth> eth.getTransactionReceipt("0x851d5...ef40b") 괄호 인자는 hash를 적어준다.  
> eth.getTransactionReceipt("0x851d5...ef40b").contractAddress 배포주소를 출력한다.  
> eth.getCode(greeter.address) 배포 주소로 deployed code를 출력한다.

## 3.4 단계 4: 사용

컨트랙이 배포되었고, 그 주소를 획득하였다. 이제 그 컨트랙의 함수(API)를 호출해보자.

그렇게 하려면: \* 먼저 컨트랙의 객체를 생성하고, \* 함수(API)를 호출한다.

### 4-1 주소를 가진 객체생성

배포한 컨트랙을 사용하려면 객체를 생성해야 하는데, 앞서 얻은 ABI, 컨트랙 주소가 필요하다.

web3 1.0 이후에는 아래와 같이 new 명령어와 함께, abi와 contractAddress를 인자로 넘겨줘야 한다.

var MyContract = new web3.eth.Contract(abi, Address);

이 명령어는 블록체인에 생성된 객체를 가져온다. 이러한 거래가 발생하면, transactionHash가 생성되고, 마이닝이 완성되면 블록체인에 기록된다.

### 4-2 methods

컨트랙의 API 함수를 호출하려면, methods와 함수 이름을 '.'으로 구분해서 붙인다. 즉 myContract.methods.myMethod(123) 형태로 코드를 작성해서 myContract의 myMethod 함수를 호출할 수 있다.

또는 함수 이름에 call() 또는 send()를 붙여서 함수를 호출할 수 있다.

myballot.methods.vote.call(1);  
myballot.methods.vote.send(1, {from: 계정주소를 여기에 넣는다})

인자가 여러 개인 경우, 콤마(',')로 분리해서 전달한다. 다음에서는 문자열 'param1', 정수 23 인자 2개가 콤마로 분리해서 전달되고 있다. 아래는 send()를 생략하고, 바로 함수의 인자로 직접 적고 있다. 어느 쪽을 선택해도 가능하다.

myContract.methods.myStateChangingMethod('param1', 23, {value: 200, gas: 2000});

### call, send

주소를 획득하면 컨트랙 객체를 사용할 수 있게 된다. 블록체인에 기록되지 않는 call() 또는 블록체인에 기록되는 send() 함수를 사용한다. call() 함수는 주로 데이터를 가져올 때 사용한다. 따라서 블록을 생성하지 않는다. send() 함수는 블록에 새로운 내용을 추가하거나 수정할 때 사용한다. gas가 소요된다.

|  |  |  |
| --- | --- | --- |
| 함수 | 설명 | 블록체인 |
| call() | 로컬에서 실행, 마이닝 불필요 | view, pure, constant 함수를 호출하는 경우. 블록체인에 기록되지 않는다. |
| send() | 블록체인에 기록하는 거래라서 마이닝 필요. transactionHash가 생성된다. | 블록체인에 기록된다. |

API를 사용하는 프로그램은 geth나 ganache 어떤 것을 사용해도 동작한다. 단 배포와 어긋나게 사용하지 않아야 한다. 배포를 geth@8445에 했다면, 일관되게 그 것을 사용해야 한다는 의미이다. 여기서는 geth@8445와 ganache@8345를 사용할 때 포트 번호를 다르게 지정했으므로, 다음 코드의 HttpProvier() 함수에 전달되는 URL을 지정할 때 포트 번호를 적절하게 지정하면 된다. 컨트랙의 주소는 정확하게 복사해서 사용한다.

[파일명: src/HelloUse.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var shelloContract = new web3.eth.Contract([{"constant":true,"inputs":[],"name":"sayHello","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"pure","type":"function"}], <--- 컴파일해서 얻은 abi를 복사-붙여넣는다  
 "0x0d3C29cAD3c40497699e13CC34bA099E1fe426Ba"); <--- ganache@8345의 배포후 얻은 컨트랙 주소를 복사-붙여넣는다  
shelloContract.methods.sayHello().call().then(function(str) {console.log(str);});

실행해보자.

pjt\_dir> node src/HelloUse.js  
  
Hello World

더 알아보기: **버전 문제**

Hello를 Geth@8445에서 Deploy는 성공하였지만, API를 사용하는 단계에서 반환 값의 오류가 나는 경우가 있다. 이런 오류를 경험하였다 **ERROR Returned values aren't valid. did it run Ouf of Gas?** 지급계좌도 해제되고, 잔고도 충분하고 gas를 지급할 수도 있었다. 이 경우 **Solidity 버전을 낮추어** 컴파일 한 후 얻은 ABI와 바이트 코드를 적용해보자 (0.4.25, 0.5.0, 0.5.1, 0.5.3 모두 성공, 0.5 후반대와 0.6은 모두 오류) 또는 getPastEvents() 함수를 사용하는 경우 web3 1.0.0 beta에서는 **ERROR Returned values aren't valid. did it run Ouf of Gas?** 오류가 나지만, **web3 버전을 낮추어** 0.20.7에서는 그렇지 않다는 이슈가 있기도 하다. 나중에 배우겠지만 **Library**를 배포하고, API를 사용하면서 Solidity 0.5.x에서는 올바르게 작동하지 않기도 하였다. 버전을 낮추어 0.4.25에서는 오류가 사라졌다. Solidity 언어가 상위 버전이 하위 버전을 동일하게 지원하는 **하위 호환성(backward compatibility)**이 보장되지 않아서 이러한 문제가 발생할 수 있다.

### 3.5 웹디앱

이제까지 "Hello World"를 출력하기 위한 절차를 완성하였다. 다른 언어에서의 경험과 사뭇 다르기도 하다.

앞서 구현한 Hello 컨트랙을 웹페이지로 만들어 보자. 단순하지만 웹페이지에서 블록체인과 인터페이스하는 경험을 하게 된다.

지금까지 경험한 웹페이지는 서버가 어디 한 곳에 있고, 서비스를 요청하면 응답하는 중앙 처리 방식이다.

여기서 만드는 웹페이지는 서비스가 처리되는 중앙 서버가 없는 **분산 애플리케이션(Distriubted Application, dApp)**이다. 서버가 없고, 블록체인에서 서비스가 제공된다는 점이 특이하다.

먼저 웹서버를 띄우자.

pjt\_dir> python -m http.server 8045  
Serving HTTP on 0.0.0.0 port 8045 (http://0.0.0.0:8045/) ...  
127.0.0.1 - - [30/Apr/2020 06:38:09] "GET / HTTP/1.1" 200 -

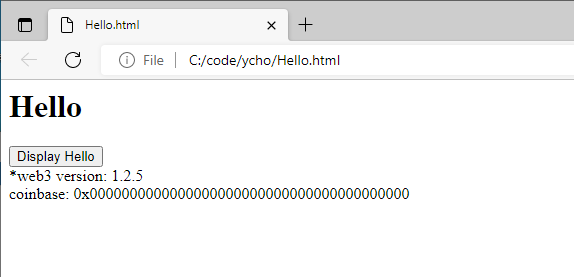
다음으로 Hello 컨트랙으로부터 결과를 받아서, "Hello World!"를 출력해보자. 처음으로 만드는 dApp이다.

아래는 web3 1.2에 맞추고 있다. web3는 cdn에서 호출하고 있으니 버전을 올려도 되고, 작동에는 문제가 없을 것으로 보인다. 단 하위 0.20 버전은 문법이 다르므로, 주석처리 해놓았고 필요하면 활성화하면 된다. 코드 내용은 웹 페이지와 버튼 등이 눌렸을 때 페이지에 내용을 보여주기 위해 사용되는 자바스크립트 코드를 제외하면, node.js를 이용해서 실행시켰던 자바스크립트 코드를 복사해서 붙인 정도이므로 더 이상의 설명은 생략한다.

[파일명: scripts/hello.html]  
<!doctype>  
<html>  
<head>  
<script src="https://cdn.jsdelivr.net/npm/web3@1.2.5/dist/web3.min.js"></script>  
<!-- script src="https://cdn.jsdelivr.net/npm/web3@0.20.5/dist/web3.min.js"></script -->  
<script type="text/javascript">  
 //var Web3 = require('web3'); // node에서는 필요하지만 여기서는 필요하지 않다  
 var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
 function displayHello() {  
 var web3version = web3.version; //web3.version.api  
 //web3@0.20.x  
 //var shelloContract = web3.eth.contract([{"constant":true,"inputs":[],"name":"sayHello","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"pure","type":"function"}]);  
 //var hello = shelloContract.at("0x0d3C29cAD3c40497699e13CC34bA099E1fe426Ba");  
 //var hello=hello.sayHello.call();  
 //document.getElementById('sayhello').innerText = "The message from blockchain: " + hello;  
 //document.getElementById('web3ver').innerText = 'web3 version 0.20.x: '+web3version.api;  
 //document.getElementById('web3coinbase').innerText = 'coinbase: ' + web3.eth.coinbase;  
  
 //web3@1.2.x - can not instanciate a contract  
 var shelloContract = new web3.eth.Contract([{"constant":true,"inputs":[],"name":"sayHello","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"pure","type":"function"}],  
 "0x0d3C29cAD3c40497699e13CC34bA099E1fe426Ba");  
 shelloContract.methods.sayHello().call().then(function(str) {  
 document.getElementById('sayhello').innerText = "The message from blockchain: " + str;  
 });  
 document.getElementById('web3ver').innerText = '\*web3 version: '+web3version;  
 web3.eth.getCoinbase().then(function(coin) {  
 document.getElementById('web3coinbase').innerText = 'coinbase: ' + coin;  
 });  
 }  
</script>  
</head>  
<body>  
 <h1>Hello</h1>  
 <button type="button" onClick="displayHello();">Display Hello</button>  
 <div></div>  
 <div id="sayhello"></div>  
 <div id="web3ver"></div>  
 <div id="web3coinbase"></div>  
</body>  
</html>

이 파일을 로컬 호스트에서 띄어보자. 웹브라우저에서 다음 주소를 입력해본다.

http://localhost:8045/scripts/hello.html



# 4. Greeter 디앱

앞서 Hello 컨트랙은 함수 하나만으로 구성되었다. 이를 조금 수정하여, **데이터를 저장하는 컨트랙**을 개발해보자. 아직 초보니까 단순한 문자열을 저장하기로 한다. 기억해보면, 앞에서는 "Hello World"를 반환했을 뿐이다. 저장한 후 반환하는 것이 뭐 다를까 생각하지 말고, 앞서 배운 절차를 숙련하는 느낌으로 도전해보자.

Solidity 언어에 대해서는 다음 장부터 설명한다. 그러니 여기서 디앱에 사용되는 코드가 이해가 안된다고 걱정하지 말자. 나중에 배우게 된다. 여기서는 단순하게 따라해보도록 한다.

우리가 저장하는 문자열을 생각해보자. Solidity의 string 자료형은 동적으로 크기가 할당된다. 따라서 문자열의 길이가 가변적일 때 사용하면 유용하다. 반면에 일정한 크기로 제한한다면 bytes4(4 바이트까지 저장하는 자료형), bytes8(8 바이트까지 저장할 수 있는 자료형) 등을 사용하는 편이 효율적이다. 비슷하게 최대 32바이트를 사용한다면 bytes32를 사용하고 크기를 알 수 없다면 string 자료형을 선택한다.

단 데이터를 저장하면, 그 크기에 따라 gas가 더 필요하다는 것을 주의해야 한다. 256비트 당 20,000 gas가 필요하다.

## 4.1 단계 1. 컨트랙 개발

### REMIX에서 프로그램 작성

REMIX 사용법은 자세히 다루지 않고 있다. 직관적이므로 둘러보면 자연스럽게 배울 수 있다.

맨 좌측 '파일 탐색기(File Explorer)' 메뉴를 선택하고 '문서' 모양의 버튼을 눌러 빈 프로그램을 하나 연다. 프로그램의 이름을 정하면 그 이름을 파일명으로 쓰도록 하자. src 디렉토리에 Greeter.sol이라고 명명하자 (아래 파일명 참조)

[파일명: src\Greeter.sol]  
pragma solidity ^0.8.0;  
  
contract Greeter {  
 string greeting;  
  
 constructor() public {  
 greeting = 'Hello';  
 }  
  
 function setGreeting(string memory \_greeting) public {  
 greeting = \_greeting;  
 }  
  
 function greet() view public returns (string memory) {  
 return greeting;  
 }  
}

프로그램은 매우 단순하다. 코드를 설명해보자.

* 줄6 ~ 8: 생성자(constructor)는 컨트랙의 객체를 만드는 기능이다. 컨트랙의 명칭은 적지 않아도 된다. 특이하게 배포 시점에 실행되고 그 상태가 유지된다. 생성자 내에서 'Hello' 문자열을 기본으로 저장하고 있다.
* 줄10 ~ 16: 매우 단순한 기능으로, 문자열에 대해 set, get 기능을 구현한다. greet()은 블록체인에서 읽어오는 함수, setGreeting()은 블록체인의 값을 변경한다. setGreeting()함수는 블록체인에 기록된다.

다음으로 컴파일 해야 한다. 더 진행하기 전, REMIX에서 컴파일, 배포, 테스트까지 해보는 것이 좋다. 컴파일-오류 발견-재컴파일-배포-테스트-수정-재컴파일 이런 흔히 겪는 개발자의 고통스런 반복을 상당히 줄일 수 있기 때문이다.

* 컴파일은 맨 좌측 메뉴에서 찾을 수 있다. 컴파일하고 오류가 발생하면 알려준다. 물론 오류가 없으면 다음 단계로 이동할 수 있다.
* 다음은 컴파일 메뉴 바로 아래에 위치한 배포 버튼을 클릭하면 된다. 배포하고 나면 바로 테스트가 가능하다. 우리가 만든 함수들이 주황색, 하늘색 버튼이 만들어져 있다. Greeter의 경우 2개의 버튼이 생성되어 있다는 것을 발견할 것이다. 얼마나 편리한가!
* 테스트는 주황색 버튼 옆 칸 (set함수이므로 인자가 필요하다)에 문자열을 입력하면 된다 (한글도 가능하다). 그러고 나서 하늘색 버튼을 클릭하면 방금 입력한 문자열이 출력된다.

이렇게 하고 나서는 '다 끝났네..' 하면서, 더 이상 읽지 않고 다음 단계를 건너뛰려고 하지 말자. 침착하자. 설마 버튼 클릭하면서 블록체인 프로그래밍을 배우려는 생각은 아닐 것이다. 곧 설명하려는 컴파일, 배포, 사용 단계에서는 자바스크립트로 코딩하고 있고, 웹디앱을 만들 때 반드시 필요하게 된다.

## 4.2 단계 2. 컴파일

REMIX에서도 컴파일할 수 있지만, 여기서는 solc로 컴파일 해보자.

우선 스위치에 적어서 abi를 구해보자.

pjt\_dir> solc-windows.exe --abi src\Greeter.sol  
  
======= src/Greeter.sol:Greeter =======  
Contract JSON ABI  
[{"inputs":[],"stateMutability":"nonpayable","type":"constructor"},{"inputs":[],"name":"greet","outputs":[{"internalType":"string","name":"","type":"string"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"string","name":"\_greeting","type":"string"}],"name":"setGreeting","outputs":[],"stateMutability":"nonpayable","type":"function"}]

이번에는 바이트 코드를 출력해보자. 길어서 중간 생략한다.

pjt\_dir> solc-windows.exe --bin src\Greeter.sol  
   
======= src/Greeter.sol:Greeter =======  
Binary:  
60806040523480156100...0006010033

참고로 gas를 산정해보자. gas가 무한 infinite로 산정되고 있다. 이 경우는 산정할 수 없다는 표현이 더 적절하다. 프로그램의 string은 배열이고 그 크기를 알 수 없기 때문에 그렇다.

실제 동적배열 또는 반복문을 적절하지 못하게 사용하면 gas가 무한으로 산정될 수 있고 따라서 실행이 되지 않을 수 있다는 점에 유의해야 한다.

pjt\_dir> solc-windows.exe --gas src\Greeter.sol  
   
======= src/Greeter.sol:Greeter =======  
Gas estimation:  
construction:  
 infinite + 156800 = infinite  
external:  
 greet(): infinite  
 setGreeting(string): infinite

## 4.3 단계 3. 컨트랙 배포

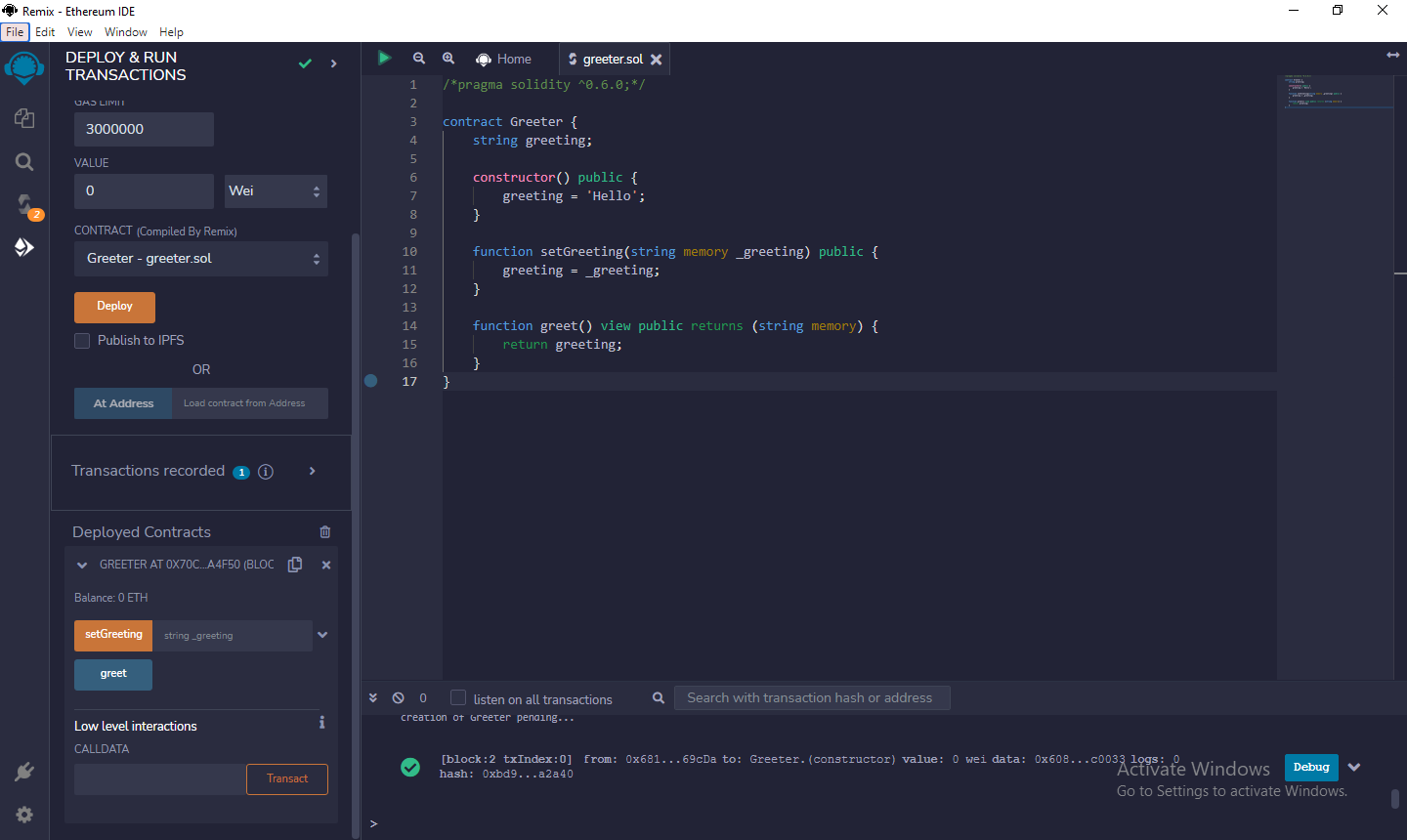
앞서 획득한 ABI, 바이트 코드로 컨트랙 객체를 생성하고 배포를 요청한다. 기억하자. geth@8445에 배포하고 있다. 배포하기 전 지급 계정이 해제되어야 하고, 마이닝을 해야 하고, 비로서 컨트랙이 블록체인에 배포된다. 배포가 완성되면 주소 값이 생성된다.

### REMIX

REMIX에서의 배포는 매우 간편하다. Deploy 버튼을 누르기만 하면 된다. 배포하기 전, 옵션 중 하나를 선택해야 하는 것은 잊지 말자.

* REMIX VM: 현재 사용 중인 REMIX의 가상 환경에 배포하는 것으로, 테스트할 때 유용하다. 우리는 보통 이 방법을 사용하고 있다.
* Injected Provider: MetaMask를 사용할 경우에 선택한다. MetaMask는 14장에서 배우게 된다.
* External Http Provider: 현재 로컬호스트에 떠 있는 원격접근 RPC에 연결할 때, 우리는 http://localhost:8445에 접속한다. 연결하면 자신의 계정이 'ACCOUNT' 박스에 보일 것이다. 거래에 사인할 때 이 계정이 사용된다. 주황색Deploy'버튼을 누르면, geth@8445에 배포하게 된다. 자신의 백그라운드를 확인하면 거기에 뜬다. 당연히 마이닝은 해주어야 한다. 그러고 나면, 우리가 원하는 컨트랙 주소가 주어진다.

어떤 방법을 선택하든 매우 간편하게 클릭으로 해결된다. 이렇게 배포하고 나면 아래 REMIX 화면의 아래 쯤에 있는 Deployed Contracts 밑에 Greeter 컨트랙을 발견하게 될 것이다. 이를 선택하여 실행해 볼 수 있다. 버튼의 색에 따라 함수를 구분할 수 있게 해 놓았다.



png

* **주황색은 transaction**, 이 경우 setGreeting() 함수에 0x로 시작하는 16진수 문자열을 입력하자.
* **하늘색은 view, pure 함수**로 블록체인에 기록되지 않는 함수이다. greet() 함수가 해당되는데, 버튼을 누르면 설정된 인사가 출력된다.

### web3.js

REMIX로 하면 매우 간편하다는 것을 경험하였는데, 굳이 코딩해서 배포하고 싶지 않을 수 있다. 익숙해지면, 코딩 방식이 효율적이고 오류에 대처할 수 있는 능력이 향상될 수 있다.

web3 1.20으로 배포를 해보자. 3째 줄에서 web3.currentProvider는 현재 설정을 그대로 쓴다는 뜻이다 (14장에서 배우게 된다). 그렇지 않으면 6째 줄에서 외부의 설정을 사용하게 된다.

15번째 줄의 from: 부분에 본인의 어카운트 번호를 입력하고 코드를 실행시킨다. geth@8445 또는 ganache@8345에 배포하느냐에 따라 6번째 줄의 URL의 포트 번호를 다르게 지정한다. 여기서는 geth@8445에 배포하는 것을 설명한다.

[파일명: src\greeterDeploy.js]  
var Web3=require('web3');  
var web3;  
if (typeof web3 !== 'undefined') {  
 web3 = new Web3(web3.currentProvider);  
} else {  
 web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8445"));  
}  
var \_abiArray=[{"constant":false,"inputs":[{"name":"\_greeting","type":"string"}],"name":"setGreeting","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"greet","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}];  
var \_bin="60806040523480156100...생략...216b550029";  
var \_contract = new web3.eth.Contract(\_abiArray);  
//web3.personal.unlockAccount(web3.eth.accounts[0],'password'); 비밀번호 해제는 콘솔에서 하는 편이 좋다  
\_contract  
 .deploy({data:"0x"+\_bin})  
 .send({from: "0xfA279EEA36550b831ce5734475F67e7d6eC4d607", gas: 364124, gasPrice: '1000000000'})  
 .then(function(newContractInstance){  
 console.log(newContractInstance.options.address) // instance with the new contract address  
 });

이제 배포를 해보자. 터미널 화면에서 node를 이용해 greeterDeploy.js를 실행시킨다.

pjt\_dir> node src\greeterDeploy.js

배포할 때 주의할 점을 살펴보자.

배포를 geth@8445에 하려면, geth 콘솔에서 몇 가지 추가 작업을 해야 한다. 다음과 같이 "geth attach http://localhost:8445" 콘솔창을 열자.

pjt\_dir> geth attach http://localhost:8445

우선 배포할 컨트랙이 큐에 대기하고 있는지 확인한다. 마이닝을 켜놓지 않아서 대기 상태로 있을 것이다. 다음과 같이 "txpool.inspect" 명령을 입력해보자. 현재 상태에 따라 출력은 다를 수 있으니, 자신과 다르다고 걱정할 필요는 없다. 배포 명령이 대기에 들어와 있으면 된다.

geth> txpool.inspect  
{  
 pending: {  
 0x8E6A037be185a70C101cDE94079a0d244Cc30FbA: {  
 0: "contract creation: 0 wei + 4700000 gas × 1000000000 wei",  
 1: "contract creation: 0 wei + 364124 gas × 1000000000 wei",  
 2: "contract creation: 0 wei + 364124 gas × 1000000000 wei",  
 3: "contract creation: 0 wei + 364124 gas × 1000000000 wei"  
 }  
 },  
 queued: {}  
}

대기하는 배포 명령을 완성하려면, 마이닝이 필요하다. 지금 열려 있는 geth 명령행에 다음을 입력한다. 마이닝을 시작하고 .start(), 거래 한 건의 마이닝이 진행될 동안 단말을 대기하도록 .sleepBlocks(1) 한다. 그리고 마이닝이 끝나면 .stop() 중단한다.

geth> miner.start(1);admin.sleepBlocks(1);miner.stop();  
null  
> txpool.inspect  
{  
 pending: {}, --> 마이닝을 하고나면 당연히 대기거래가 없어진다  
 queued: {}  
}  
>

마이닝을 하는 시간은 상황에 따라 편차가 있다. 컴퓨터 속도와 현재 실행되고 있는 프로그램 등에 따라 얼마나 걸릴지 모르는데, 기다리다 보면 마이닝이 끝난 후 null이라는 결과가 나타난다. 그러면, greeterDeploy.js를 실행시킨 창에 컨트랙 주소가 출력된다.

pjt\_dir> node src\greeterDeploy.js  
0xE4E640c0BF81Bbe45342589bd0c318D837ebDf2A ---> 이 주소는 복사해서 사용단계에서 코드의 (X)에 붙여넣는다.

이러면서 geth@8445를 띄어 놓은 단말을 관찰하자. 화면이 바쁘게 로그를 출력하고 있을텐데 다음 메시지를 발견할 수 있다.

* Submitted contract creation 컨트랙 생성을 요청한 후 해시를 발행하고,
* Commit new mining work 마이닝이 되었다는 로그 메시지를 관찰할 수 있다.

다시 한 번 강조하면, ganache@8345에 배포하는 경우 이런 절차가 필요없다.

## 4.4 단계 4. 사용

위에서 얻은 컨트랙 주소는 꼭 필요하다. 앞서 만든 컨트랙을 이 주소가 없으면 호출할 수 없다.

객체를 생성할 경우 ABI, 바이트 코드가 필요하다.

함수를 호출할 경우, call() 또는 send() 어느 방식을 적용할지 판단한다.

* 값을 읽는 경우 call()을 사용하고, gas비용이 발생하지 않는다.
* greeter.methods.greet().call()
* 값을 변경하는 경우, 함수를 직접 호출하거나 send()을 사용한다. gas비용이 발생하고, 적게 설정하면 오류가 난다. 반환 값이 없다. options에는 from, gasPrice, gas, value를 넣어준다.

myContract.methods.myMethod([param1[, param2[, ...]]]).send(options[, callback])

자 그럼 setGreeting() 함수를 호출해보자.

greeter.methods.setGreeting("Hello World").send({from:web3.eth.accounts[0],gas:1000});

web3 1.20으로 배포를 해보자.

[파일명: src/greeterUse.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8445"));  
var \_abiArray=[{"constant":false,"inputs":[{"name":"\_greeting","type":"string"}],"name":"setGreeting","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"greet","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}];  
var greeter = new web3.eth.Contract(\_abiArray,"0xE4E640c0BF81Bbe45342589bd0c318D837ebDf2A");  
greeter.methods.greet().call().then(function(value) {console.log(value);});  
greeter.methods.setGreeting("Hello SMU").send({from:"0xfA279EEA36550b831ce5734475F67e7d6eC4d607",gas:100000});  
greeter.methods.greet().call().then(function(value) {console.log(value);});

위 함수를 1회 실행하면, 비동기적 특성으로 setGreeting() 함수가 반영되지 않는다. 반영되지 않은 상태로 처음에는 "Hello"가 출력된다. 2회 실행시키면, 비로서 "Hello SMU"로 문자열을 변경한 결과가 아래와 같이 출력된다.

pjt\_dir> node src/greeterUse.js  
  
Hello SMU  
Hello SMU

# 5. 간단한 계수기

계수기는 언제나 현재의 계수를 가지고 있다. 0부터 시작해서 하나씩 증가하거나 감소하며, **현재의 계수로부터 증감**한다는 것이다.

블록체인은 **상태 기계(state machine)**이라서, 객체가 생성되면서 초기화되지 않고 현재 값을 기억하고 있다. **프로그램을 호출한 횟수**가 좋은 예가 될 수 있다. 여러 사람이 호출했다고 하더라도 총 횟수를 기억할 수 있다. 현재 값을 저장하고 있고, 그 값이 누구에게나 공개된다는 블록체인의 특징을 보여주고 있다.

## 5.1 단계 1. 컨트랙 개발

컨트랙은 Counter라고 명명하고, 계수를 증감하고 읽는 함수를 넣어보자.

코드는 REMIX에서 작성하자. 다음 단계에서 컴파일, 배포를 하겠지만, REMIX에서 앞서 그랬던 것과 같이 테스트까지 하는 편이 좋다. 그렇지 않으면 개발-컴파일-배포-오류-수정 절차가 반복되기 때문이다.

[파일명: src\Counter.sol]  
pragma solidity ^0.8.0;  
contract Counter {  
 uint256 counter = 0;  
 function add() public {  
 counter++;  
 }  
 function subtract() public {  
 counter--;  
 }  
 function getCounter() public view returns (uint256) {  
 return counter;  
 }  
}

* 줄L4 ~ 9: add(), subtract() 함수는 계수를 1씩 증감하고 있다.
* 줄10 ~ 12: getCounter()는 계수를 읽어내고 있다.

## 5.2 단계 2. 컴파일

컴파일해서 abi, bin, gas를 구한다.

pjt\_dir> solc-windows.exe --abi --bin --gas src\Counter.sol  
   
======= src/Counter.sol:Counter =======  
Gas estimation:  
construction:  
 5141 + 89400 = 94541  
external:  
 add(): 21082  
 getCounter(): 1159  
 subtract(): 21104  
  
Binary:  
60806040526000805534...생략...736f6c63430008010033 ---> (A) 아래 (X)에 복사-붙여넣기  
  
Contract JSON ABI  
[{"inputs":[],"name":"add","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"getCounter","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[],"name":"subtract","outputs":[],"stateMutability":"nonpayable","type":"function"}] ---> (B) 아래 (Y)에 복사-붙여넣기

## 5.3 단계 3. 배포

web3.js를 사용한 배포 프로그램이다.

코드의 .send({from:에 자신의 계정 번호를 적고 실행시킨다. 기억하자. ganache@8345에서 출력한 10개 가운데 하나를 복사-붙여넣기 하고 있다.

[파일명: src\counterDeploy.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
var \_abiArray=[{"inputs":[],"name":"add","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"getCounter","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[],"name":"subtract","outputs":[],"stateMutability":"nonpayable","type":"function"}]; <--- (Y) 위 (B)를 복사-붙여넣기  
  
var \_bin=60806040526000805534...생략...736f6c63430008010033; <--- (X) 위 (A)를 복사-붙여넣기  
  
var \_contract = new web3.eth.Contract(\_abiArray);  
\_contract  
 .deploy({data:"0x"+\_bin})  
 .send({from: "0x0A815B7818A8e6BC27B430e41Edc8FC455F658c2", gas: 364124, gasPrice: '1000000000'})  
 .then(function(newContractInstance){  
 console.log(newContractInstance.options.address) // instance with the new contract address  
 });

위 프로그램을 실행하면 컨트랙 주소가 성공적으로 출력된다.

pjt\_dir> node src\counterDeploy.js  
  
0xb24ab776373e53fFeb9B7298209E195853D2fc8e ---> (C) 아래 (Z)에 복사-붙여넣기

## 5.4 단계 4. 사용

counterDeploy.js를 실행시킨 후에 나온 주소값을 addr 변수에 저장하고, "counter.methods.add().send({from:"의 값을 역시 본인 것으로 수정한 후 실행한다.

[파일명: src/counterUse.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var abi =[{"constant":false,"inputs":[],"name":"add","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":false,"inputs":[],"name":"subtract","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"getCounter","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"}];  
var addr = "0xb24ab776373e53fFeb9B7298209E195853D2fc8e"; <--- (Z) 위 (C)를 복사-붙여넣기  
var counter = new web3.eth.Contract(abi,addr);  
counter.methods.getCounter().call().then(function(str) {console.log(str);});  
//counter.methods.subtract().send({from:"0x0A815B7818A8e6BC27B430e41Edc8FC455F658c2",gas:100000});  
counter.methods.add().send({from:"0x0A815B7818A8e6BC27B430e41Edc8FC455F658c2",gas:100000});  
counter.methods.add().send({from:"0x0A815B7818A8e6BC27B430e41Edc8FC455F658c2",gas:100000});  
counter.methods.getCounter().call().then(function(str) {console.log(str);});

geth 개인망에서 add(), subtract() 함수는 호출하고, Pending Transactions을 처리하고 나면 숫자가 증감한 것을 알 수 있다.

add()는 counter에 1을 더하는, subtract()은 1을 빼는 상태변경 이므로 마이닝을 해야 그 결과가 반영된다. 그러고 나면 getCounter()에서 알 수 있다.

주목할 점은 수가 계속 증감한다는 점이다. 세션을 다시 열어도 전의 counter 값은 남아 있다.

pjt\_dir> node src/counterUse.js  
  
4  
4

# 실습문제: 타이머 디앱 - 파일에서 ABI, 바이트 코드 읽는 방식

블록체인의 타이머를 만들어 보자.

블록체인에서 객체는 한 번만 생성되고, 공유된다. 따라서 블록체인의 타이머는 자신만 사용하는 것이 아니라, 다른 누군가 사용해도 된다. 공유 타이머를 만들려고 하면, 매핑을 적용해서 나의 시간, 너의 시간을 다르게 설정할 수 있다. 여기서는 자신만 쓰는 간단한 타이머를 만들기로 하자.

지금까지 컴파일하고 abi, 바이트 코드를 매번 복사하는 것이 불편할 수 밖에 없다. 여기서는 컴파일하고 생성되는 abi, 바이트 코드를 파일로 저장하도록 하자. 프로그램이 조금 복잡해지겠지만, 작성된 파일에서 읽어내면, 복사-붙여넣기를 하지 않아도 되기때문에 간편해진다.

또한 geth@8445에 배포하고, API를 사용하여 보자.

## 단계 1: 컨트랙 개발

코드는 REMIX에서 작성하자. 코드를 작성하고 난 후, 그냥 끝내지 말고 배포까지 하고 나서 기능 버튼이 만들어 지면 테스트도 해보자.

[파일명: src\Timer.sol]  
pragma solidity ^0.8.0;  
contract Timer {  
 uint256 startTime;  
 function start() public {  
 startTime=block.timestamp;  
 }  
 function timePassed() public view returns(uint256) {  
 return block.timestamp-startTime;  
 }  
 function getNow() view public returns(uint) {  
 return block.timestamp;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 1 | Pragma 프로그램 버전. 맨 앞의 ^ (caret)은 메이저버전으로 시작하는 최근 버전의 컴파일러를 사용. 따라서 0.8 이상 ~ 0.9 미만도 문제 없이 사용할 수 있다. |
| 4 | 함수의 가시성을 public으로 설정하여, 사용권한을 제한하지 않는다 (가시성은 나중에 배우게 된다) |
| 5 | now는 현재 시간, 시스템에서 제공하는 전역변수. now는 버전0.7부터 더 이상 지원되지 않는다. block.timestamp를 사용해야 하며, 요청이 발생하는 시점의 timestamp이다. |
| 7 | 시간측정 함수이고 public view로 설정하여, 누구나 조회할 수 있도록 한다 (view는 나중에 배운다) |
| 10 | 현재시간을 조회하는 함수이다. |

## 단계 2: 컴파일

컨트랙 주소를 복사-붙여넣기를 하지 않기 위해, 이번에는 컴파일하고 출력되는 ABI, 바이트 코드를 JSON 파일로 내보내보자.

combined-json을 사용하면 abi, 바이트코드를 하나씩 생성하지 않고, 한꺼번에 컴파일한 JSON을 Timer.js 파일에 저장한다. 파일의 확장자가 .js라는 것은 자바스크립트를 의미한다.

--optimize 옵션은 컨트랙이 200회 정도 실행될 것이라고 예상하고, 생성하는 바이너리를 최적화하게 된다.

주의! 운영체제 별로 디렉토리 구분자가 다르므로 주의해야 한다. - 윈도우는 역슬래쉬 src\Timer.json, - 리눅스는 그냥 슬래쉬 src/Timer.json라고 적는다.

이런 원칙에도 불구하고 컴파일 결과에서 보면 리눅스 스타일로 적혀 있다 (아래 결과를 보면 그렇다). 아래 컴파일할 경우에도 슬래쉬, 역슬래쉬 모두 작동된다. 주의! --combined-json abi, bin 이렇게 공백이 포함되면 안된다.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/Timer.sol > src/Timer.json

Timer.json에 저장이 잘 되었는지 출력해보자. 리눅스에서는 cat src/Timer.json이다. 윈도우에서는 type src\Timer.json (src/Timer.json로 타이핑하면 슬래쉬의 문법오류)이다.

출력 결과는 이해하기 쉽도록 편집하였다. JSON 형식으로 키와 값이 사상(mapping)되어 저장되었다. 컨트랙의 명칭, abi, bin을 유의해서 보아야, 나중에 배포하면서 이 파일에서 읽어올 수 있게 된다.

pjt\_dir> type src\Timer.json  
  
{"contracts":  
 {  
 "src/Timer.sol:Timer":{  
 "abi":"[{\"inputs\":[],...생략...}]",  
 "bin":"60806040523480156100...생략...736f6c63430008010033"  
 }  
 },  
"version":"0.8.1+commit.df193b15.Windows.msvc"}

#### 파일에서 abi, 바이트 코드 조회

위 파일에서 abi, 바이트 코드를 읽는 프로그램을 작성해보자.

[파일명: src/TimerImportTest.js]  
줄1 const \_abiBinJson = require('./Timer.json'); // 파일 가져오기  
줄2 contractName=Object.keys(\_abiBinJson.contracts); // ['src/Timer.sol:Timer'] 이 부분을 읽어온다  
줄3 console.log("- contract name: ", contractName);  
줄4 \_abi=\_abiBinJson.contracts[contractName].abi;  
줄5 \_abiArray=JSON.parse(JSON.stringify(\_abi));  
줄6 \_bin=\_abiBinJson.contracts[contractName].bin;  
줄7 console.log("- ABI: ", \_abi);  
줄8 console.log("- ABIArray: ", \_abiArray);  
줄9 console.log("- Bytecode: ", \_bin);

* 줄1 JSON파일은 바로 require(JSON파일명) 함수로 읽으면 편하다. 자바스크립트 파일의 require(자바스크립트파일명)와 달리 export 명령어를 사용할 필요가 없다. require에서 읽는 파일 경로는 **현재 파일의 상대적 경로**를 지정한다. src폴더 아래에 있어서 src.json라고 써야 한다고 생각할 수 있으나 현재 파일에서 보면 같은 디렉토리에 있는 것이다. 같은 폴더에 파일이 있더라도 앞에서 보인 것처럼 "./"를 붙여줘야 한다.
* 줄2 자바스크립트 문법으로 **객체의 키**를 읽어오려면, Object.keys(키) 함수에 인자 '키'를 입력하면, 그에 해당하는 값을 얻을 수 있다. 컨트랙 명칭은 'src/Timer.sol:Timer'로 작명된다. 이 부분은 <context>:<prefix>=<target> 문법에 따라 생성이 된다.

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 설명 | 선택 | 예 |
| context | 읽을 파일 | 선택적 | src/Timer.sol |
| prefix |  |  | Timer |
| target | 없으면 prefix와 동일 | 선택적 |  |

* 줄4 JSON 자료를 점연산자로 접근하고, .abi 필드를 읽고 있다. 줄6은 같은 방식으로 .bin을 읽는다.
* 줄5 stringify() 함수는 자바스크립트 오브젝트를 문자열로 전환한다. 연쇄적으로 문자열을 JSON 자료로 변환하기 위해 .parse() 함수를 적용한다.

이 프로그램을 실행하면, abi, bin 정보를 출력한다. 잘 읽었는지 확인해보자.

pjt\_dir> node src/TimerImportTest.js  
  
 - contract name: [ 'src/Timer.sol:Timer' ]  
 - ABI: [{"inputs":[],"name":"getNow","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[],"name":"start","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"timePassed","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"}]  
 - ABIArray: [{"inputs":[],"name":"getNow","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[],"name":"start","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"timePassed","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"}]  
 - Bytecode: 60806040523480156100...생략...736f6c63430006010033

### 노드에서 한 줄씩 코딩해보기

JSON을 생성하고 그 파일을 읽어오는 과정이 이해하기 어려우면 한 줄씩 상호작용해보자. Node에서 한 줄씩 abi, bin을 읽어오는 연습을 해보자.

pjt\_dir> node  
node> var \_abiBinJson=require('./src/Timer.json')  
undefined  
> Object.keys(\_abiBinJson.contracts)  
[ 'src/Timer.sol:Timer' ]  
> 이후에도 한 줄씩 계속 입력하고 결과를 출력해보자.

## 단계 3: 컨트랙 배포

컴파일하고 abi, bin를 파일에 저장하였고, 그 파일에서 읽어서 배포해 보자.

Node에서는 앞서 하나씩 생성한 Timer.json 파일을 읽어서 배포해보자.

주의: \* require('./Timer.json')에 호출 시점의 상대 디렉토리 src/Timer.json를 적는다. \* 바이트 코드에는 16진수 표현 "0x"를 붙인다.

|  |  |
| --- | --- |
| 줄 | 설명 |
| 1 | web3 라이브러를 가져온다. |
| 2 | 컴파일하고 생성된 abi, bin이 저장된 JSON 파일(Timer.json)을 읽는다. |
| 3 | HttpProvider를 ganache@8345로 설정한다. |
| 5 | contract명에 path prefix인 디렉토리 구분자 '/'가 포함되어 있다. Object.keys로 읽은 key를 contract명으로 사용한다. |
| 10 | binary data는 '0x'가 빠져 있다. 블록체인에 배포할 때 0x를 붙여서 16진수 문자열이라고 명시해 준다. **gas**를 임의로 정해준다. 적으면 문제가 되지만, 쓰고 남으면 반환된다. |
| 12 | deploy() 함수는 async로 선언되었다. 중간 await new로 객체 생성을 하고 있다. |

[파일명: src\TimerDeployAbiBinFromFile.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./Timer.json'); //importing a javascript file  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src/Timer.sol:Timer']  
console.log("- contract name: ", contractName);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
\_abiArray=JSON.parse(\_abi); // 오류가 발생하면 JSON.stringfy(\_abi)를 사용할 것  
\_bin=\_abiBinJson.contracts[contractName].bin;  
  
console.log("- ABI: " + \_abiArray);  
console.log("- Bytecode: " + \_bin);  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: "0x"+\_bin})  
 .send({from: accounts[0], gas: 364124}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

파일을 실행해보자.

pjt\_dir> node src\TimerDeployAbiBinFromFile.js  
  
- contract name: [ 'src/Timer.sol:Timer' ]  
- ABI: [object Object],[object Object],[object Object] Object 형식을 문자열로 출력하려면 JSON.stringify()  
- Bytecode: 60806040523480156100...생략...736f6c63430006010033   
Deploying the contract from 0xEb670169A5fcD2550035588A2ba3b45509703ABd  
hash: 0xbd010c6ed017b05579abd42f94d332ece6b4ce21c83eb881d46094fb3572eed4  
---> The contract deployed to: 0x5dB5E509ff13A760345C0B58591E2a5DA6e1B2DE

* 컨트랙 명이 올바르게 출력되고 있고,
* ABI는 출력하면 Javaascript Object가 포함되어 있다. 문자열로 출력하려면 JSON.stringify(obj) 함수를 적용한다.
* 마지막 줄에 배포된 컨트랙의 주소를 출력하고 있다.

혹시 코드를 실행하면서 JSON.parse()에서 오류가 발생할 수 있다. Solidity JSON 파일을 생성하는 방법이 버전마다 차이가 있는 것 같다. 어떤 버전으로 컴파일 하느냐에 따라 JSON.parse(\_abi)형태로 실행될 수도 있고, 오류가 발생하는 경우도 있다. 오류가 발생하면, JSON.parse(JSON.stringfy(\_abi))형태로 사용해본다.

## 단계 4: 사용

JSON에서 파일을 읽어서 ABI를 가져온다. 바이트코드는 API를 호출할 때 불필요하다. 사용할 때 지정하는 주소는 배포 후 얻은 주소값으로 변경한다. 앞에서 언급한 것처럼 JSON.parse(\_abi)에서 오류가 발생하면, JSON.parse(JSON.stringfy(\_abi))형태로 사용해본다.

우리가 호출한 getNow(), timePased()는 **call 함수**이다. 반면에 start()는 **send 함수**이다.

[파일명: src\TimerUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./Timer.json'); //importing a javascript file  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src/Timer.sol:Timer']  
//console.log("- contract name: ", contractName); //or console.log(contractName[0]);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));   
\_abiArray=JSON.parse(\_abi); // 오류가 발생하면 JSON.stringfy(\_abi)를 사용할 것  
//\_bin=\_abiBinJson.contracts[contractName].bin;  
  
// TimerDeployAbiBinFromFile.js를 실행시킨 후 나온 주소값으로 변경  
var timer = new web3.eth.Contract(\_abiArray,"0x5dB5E509ff13A760345C0B58591E2a5DA6e1B2DE");   
  
async function doIt() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Call from: " + accounts[0]);  
 timer.methods.getNow().call().then(function(value) {console.log("Now: ", value);});  
 await timer.methods.start().send({from:accounts[0],gas:100000});  
 //await 4000;  
 timer.methods.timePassed().call().then(function(value) {console.log("Passed: ", value);});  
}  
  
doIt()

일정 시간이 지난 후 몇 번 반복시켜 지나간 시간이 변동하는지 확인해 보자.

지난 시간은 타임스탬프 signed 32-bit 형식으로 출력된다. 이 형식을 우리가 사용하는 시간 형식으로 변환해야 한다 (참조: https://www.epochconverter.com/ 에서 변환기능을 제공한다)

**타임스탬프 timestamp**

유닉스에서 사용하는 타임스탬프이고 Long Integer 형식이다. 1970년 1월 1일 00:00:00 기준으로 몇 초나 지났는지 계산한다. 하루는 86400초, 60초 x 24시간 x 60분 기준이다. 윤초는 계산에서 제외해야 한다.

REMIX에서 테스트하면, 올바르게 작동하지만 여기서는 Passed가 0으로 출력되고 있다. 비동기적 특성으로 인한 것으로 판단하며, 일정시간을 기다리고 출력해서 확인해보자 (await 4000).

pjt\_dir> node src\TimerUse.js  
  
Call from: 0xEb670169A5fcD2550035588A2ba3b45509703ABd  
Now: 1648957889  
Passed: 0

# 연습문제

1. 컨트랙을 블록체인에 올려서 그 API를 사용하기 위해 해야 할 절차를 순서대로 배열하시오.
   1. 컨트랙을 코딩한다.
   2. 컨트랙을 컴파일 한다.
   3. 컨트랙 주소를 획득한다.
   4. ABI, 바이트코드를 생성한다.
   5. ABI와 컨트랙주소를 입력해서 객체를 생성한다.
   6. 컨트랙의 API를 호출한다.
2. 컨트랙을 컴파일 한 후 획득해서 다음 배포단계에서 넘겨주어야 하는 것은 무엇인가? 모두 고르시오.
3. ABI (2) 바이트 코드 (3) gas (4) 계정 주소 (5) 컨트랙 주소
4. 컴파일 한 후 획득해서 다음 사용단계에서 넘겨주어야 하는 것은 무엇인가? 위 보기에서 고르시오.
5. REMIX는 컴파일만 가능하다. OX로 답하시오.
6. 컴파일한 후 획득하는 ABI의 형식은 무엇인가?
7. 문자열 (2) 자바스크립트 JSON (3) 바이트 코드
8. 컨트랙을 배포하기 위해서 비용이 발생하는가? 힌다면, 비용 계산은 어떻게 하는가?
9. gas와 gas price를 임의로 자신이 결정할 수 있는가?
10. 계정에서 지출이 가능하도록 해제하는 것이 어느 경우 필요한가?
11. ganache (2) geth
12. 계정해제를 해주지 않는 경우, 발생하는 오류는 무엇인가?
13. 발생한 거래를 마이닝해주어야 하는 망은?
14. ganache (2) geth 사설망 (3) geth 공중망
15. REMIX에서 배포할 수 없는 대상은?
    1. REMIX 자체의 가상머신
    2. 웹지갑 (예: MetaMask)
    3. 로컬의 RPC (예: geth@8445)
    4. 로컬의 운영체제
16. call()과 send() 함수의 차이를 식별하시오.
17. 책에서 정의한 마지막 '사용' 단계에서:
    1. 투표 컨트랙의 vote() 함수를 호출하는 코드를 작성하시오. 이 함수는 투표 기능이다.
    2. 투표 컨트랙의 getProposalNum() 함수를 호출하는 코드를 작성하시오. 이 함수는 투표안 번호를 읽는 기능이다.
18. REMIX에서 버튼 기능이 활성화되면 색으로 식별할 수 있다. 주황색은 어떤 종류의 기능에 주어지는 색인지 설명하시오. 푸른색은 어떤 종류의 기능에 주어지는 색인지 설명하시오.
19. 웹디앱은 중앙처리 방식인지 아닌지 선택하고, 이유를 설명하시오.
20. 웹디앱에서 web3 라이브러리는 require('web3')로 읽어온다. OX로 답하시오.
21. 솔리더티 버전을 0.8 가운데 가장 큰 것으로 사용하는 것으로 정하는 문법을 작성하시오.
22. 전역변수 설정하는 자바스크립트 코드이다. 결과를 적으시오.

var x = 11;  
function hoist() {  
 x = 22;  
}  
hoist();  
console.log(x);

1. Hello 컨트랙을 수정해서 sayHello() 함수를 호출하면 "Hello, Snowman"이 출력되도록 하세요. Solidity로 Hello.sol을 수정하고, 컴파일하고, web3.js를 사용하여 node로 프로그램해서 배포한 후, sayHello() 출력하시오.
2. greeter의 greeting의 데이터타입을 string에서 bytes32로 변경하시오.
   1. 노드에서 배포한 후, 인사하는 문자열을 설정하고 setGreeting("Hello World!"), greet()을 출력하시오.
   2. string에서 bytes32로 변경하면 gas의 차이가 있는지 확인하시오. string을 byte32로 변경하면 글자 수가 제한되므로, gas비가 산정되는지 확인하송.
   3. "Hello SMU"와 같은 String을 인자로 사용하면, 오류가 발생하는지 확인하시오. "Hello SMU"에 해당하는 16진수 "0x48656c6c6f20534d55"를 넘겨주면, 오류가 없어지는지 확인하시오.
3. 7을 곱하는 컨트랙을 구현하시오. 8을 인자로 넘기면, 7을 곱해서 56을 출력하면 된다. 컨트랙 명은 Multiply7, 함수는 function multiply(uint input) public pure returns (uint)를 구현하시오. 컨트랙은 Solidity로, 클라이언트는 노드로 프로그램하시오.
4. 계수기와 Timer 컨트랙
   1. Counter 컨트랙을 구현하세요. add(), subtract(), getCount() 함수를 구현하고, 여기에 호출하는 시점을 출력하기 위해 다음 함수를 추가하세요. 강의자료의 Counter와 Timer를 참조하세요.
   * function setTimeCalled() internal 호출시점을 설정  
     function getTimeCalled() view public returns(uint256) 호출시점을 조회

REMIX에서 각 함수를 실행해보고, 우하단 창에 출력되는 성공로그를 복사해서 붙여넣기 하세요. 아래는 샘플입니다.

[vm]from: 0x5B3...eddC4to: Counter.add() 0x0fC...9A836value: 0 weidata: 0x4f2...be91flogs: 0hash: 0x6a7...a14d6  
transact to Counter.subtract pending ...   
[vm]from: 0x5B3...eddC4to: Counter.subtract() 0x0fC...9A836value: 0 weidata: 0x6de...ebae3logs: 0hash: 0x5aa...6e62a  
call to Counter.getCount  
CALL [call]from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4to: Counter.getCount()data: 0xa87...d942c  
call to Counter.getTimeCalled  
CALL [call]from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4to: Counter.getTimeCalled()data: 0xc45...7a0a4

* 1. 로컬에서 컴파일하세요. 파일로 abi, 바이트코드를 작성하세요.
  2. 파일로부터 abi, 바이트코드를 읽어서 배포하세요. 배포부터 geth 8445로 연결하여 node로 하세요 (ganache 8345로 하면 감점).
* (4-1) 배포된 주소로 부터 객체를 생성하고, 바로 계수와 호출시점을 출력하세요.
* (4-2) 계수를 1회 증가시키세요. 증가한 후, 증가된 계수와 호출시점을 출력하세요.
* (4-3) 위 (4-2)를 1회 반복하세요.

1. 거듭제곱 컨트랙을 구현하시오. 아래 (2) ~ (6)은 노드로 프로그램하세요.
   1. 2의 거듭제곱을 계산하는 컨트랙을 프로그램하시오. 예를 들어, 8을 입력하면 256, 32는 4294967296을 출력하게 된다. 컨트랙은 Math.sol로 저장하고, 함수는 powerOf2로 명명해서 프로그램 작성한다. power를 계산하려면, 반복문으로 하지 않는다. \*\* 연산자로 계산한다. 컴파일해서 abi, gas, bin을 화면에 출력한다.
   2. ganache@8345 배포, 파일에서 ABI, BIN을 읽으시오.
   3. ganache@8345에서 8과 32를 입력해서 결과 출력하시오.
   4. 웹페이지 powerOf2.html을 작성 (32를 입력하고, 결과를 화면에 출력)
   5. geth에서 배포하고 사용. geth@8446, geth@8445 가운데 선택.
2. Counter 콘트랙의 계수가 홀수인지, 짝수인지 판별하는 함수 isEven()을 추가하세요. geth@8445에서 실행하세요.

* Counter를 0부터 3까지 증가 (REMIX 우측 하단 함수실행 로그 붙여넣기)
* counter가 짝수일 때, isEven()을 실행해서 결과를 출력 (timestamp 포함)

1. Counter가 4이면 알려주는 컨트랙을 구현하시오.

REMIX에서 배포하고, 아래를 노드로 프로그램하세요 \* Counter를 0부터 3까지 증가 \* counter가 4일 때, isEven()을 실행해서 결과를 출력

1. Counter의 다른 세션에서의 값을 확인해보시오. 노드로 배포하고, 아래를 프로그램하세요.

* counter를 조회하고, 위 REMIX에서 배포한 콘트랙의 counter의 수와 같은지 확인.
* Counter를 0부터 3까지 증가
* 노드 세션을 새로 열어, Counter를 하나 더 증가시키려고 할 때 0부터 시작하는지 또는 전 세션에서 끝낸 counter의 수가 남아 있는지 출력
* counter가 4일 때, isEven()을 실행해서 결과를 출력