# Chapter 6. 웹3 인터페이스

탈중앙 Web3를 구현하기 위해 web3.js를 설치하고, 웹3 페이지를 만들어 보자.

# 1. 자바스크립트 웹3

블록체인에 스마트 컨트랙을 만든다고 하자. 그렇다면 웹 클라이언트에서 블록체인과 상호작용할 수 있는 자바스크립트 web3.js 라이브러리가 필요하다.

web3.js는 그 명칭에서 대략 무엇인지 느낌이 오겠지만, 웹에서 이더리움 노드에 **HTTP, WebSocket**로 연결해서, 통신하기 위한 **자바스크립트 라이브러리**이다.

분산앱 dApp을 웹에서 구현한 것을 웹디앱이라고 한다. 웹디앱에서는 블록체인에 업로드된 스마트 컨트랙의 API를 자바스크립트 Web3.js에서 호출하면서 그 결과를 화면단에서 받아서 처리하게 된다.

------------------- ----------------  
 | 웹디앱 - web3.js | --- http, WebSocket --- | 이더리움 노드 |  
 ------------------- ----------------

자바스크립트 언어는 **비동기** 프로그래밍을 지원하는데, 어떤 작업을 실행한 후 완성될 때까지 대기하지 않고 다른 작업을 하는 것이다.

동기적 프로그램에서는 함수를 호출하면, 그 작업이 끝날 때까지 무조건 기다려야 한다. 그 작업이 완성되고 나서야 다음 작업을 실행할 수 있다. 웹 사용자에게는 이러한 대기가 몹시 불편하기 마련이다. 비동기 프로그래밍 방식에서는 함수를 실행시키는 작업을 시작한 후, 그 함수가 실행되고 있는 것과 관계없이 다음 작업을 수행할 수 있다.

블록체인은 그 특성상 비동기 프로그래밍이 자연스러워 보인다. 거래가 발생한 후 바로 인증이 되는 것이 아니다. **마이닝을 통해 작업 증명을 해야 하고, 일정 시간 후 거래가 완성**이 되는 방식이다. 따라서 그 시간 동안 프로그램의 실행을 막아놓고 마이닝이 끝나기를 기다리는 동기방식은 바람직하지 않다. 어떤 작업을 실행한 후, 인증되기까지 기다렸다가 다른 작업을 시작해야 한다면, 그런 동기적 방법은 적합하지 않다.

예를 들어, 블록체인으로 투표를 했다고 하자. 곧 이어 득표수를 조회해도 반영되지 않을 수 있다. **바로 변경된 값을 반영하기 위해 마이닝을 하려면 시간이 걸릴 수 있기** 때문이다. 마이닝을 해야 완성되므로 비동기 방식이 자연스럽다.

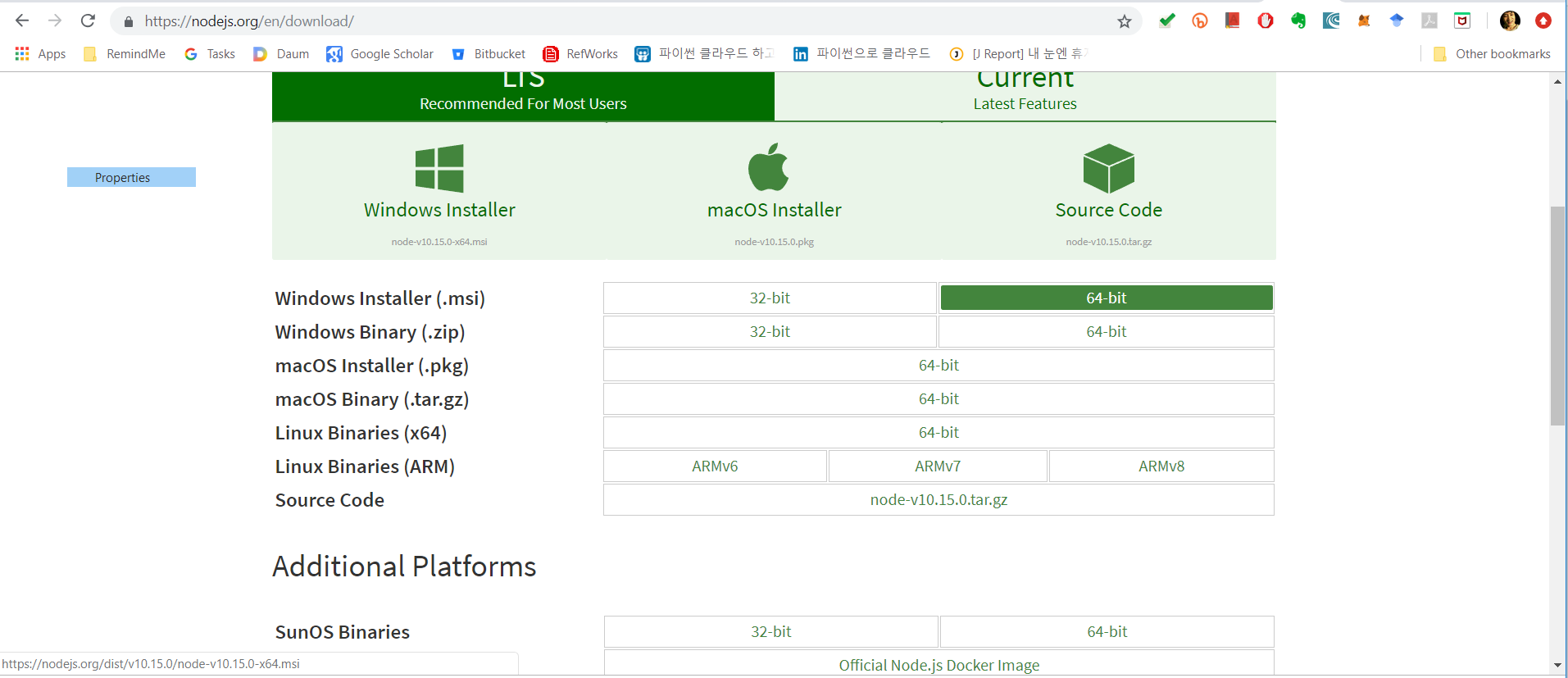
## 1.1 node, npm 설치

### node 설치

자신의 운영체제에 맞는 node.js를 다운로드한다 (https://nodejs.org) 윈도우에 설치하려면 64 또는 32비트 중 자신의 컴퓨터에 맞는 설치파일을 다운로드해서 설치한다.

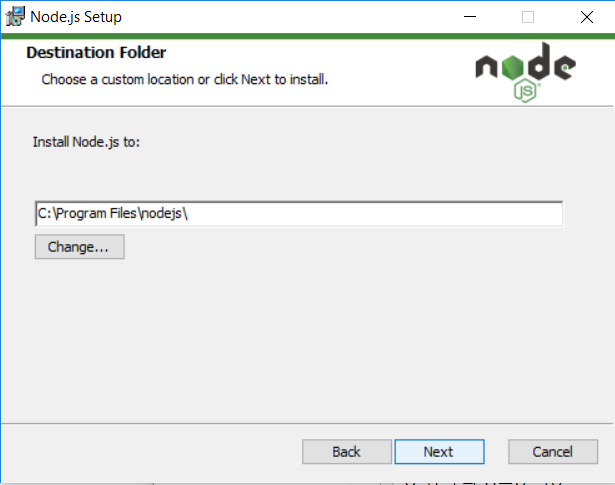
리눅스나 맥os를 사용하면, 단말에서 명령어를 입력하여 설치할 수 있다.

* 리눅스에서는 apt install node.js,
* 맥은 brew install node.js,
* 윈도우에서는 아래와 같이 msi 파일을 내려받아 설치한다.



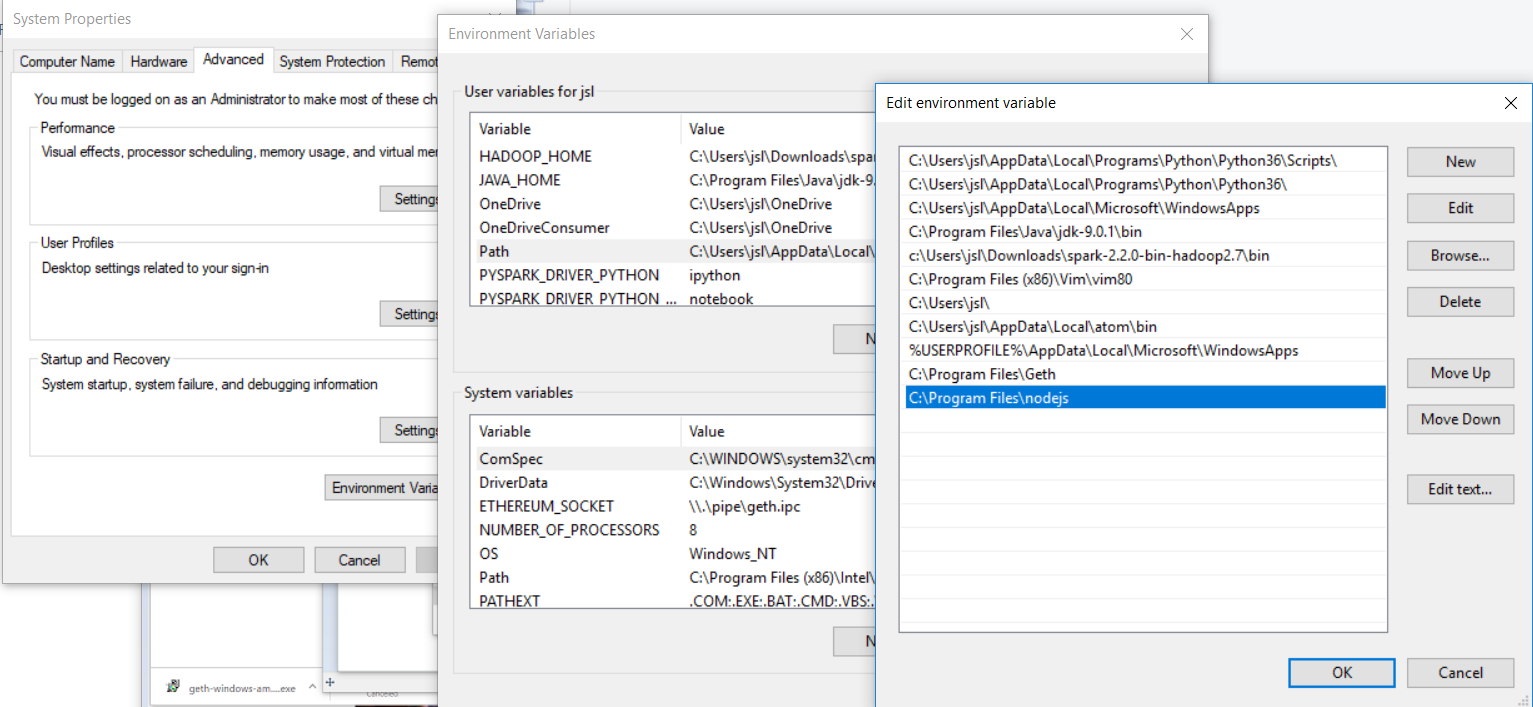
alt text

윈도우 설치 디렉토리는 C:\Program Files\node.js로 하자.



alt text

node.js의 경로를 설정한다. node.js를 설치한 경로를 등록하여, 명령창에서 실행이 바로 가능하도록 한다.



alt text

### npm

**web3.js를 설치하기 위해서는 npm이 필요**하다.

npm은 노드 패키지 관리자 Node Package Manager로서, 자바스크립트 패키지들을 설치할 수 있도록 해주는 일종의 앱스토어 같은 저장소 역할을 한다. npm은 노드 패키지의 저장소를 검색하고 설치하거나 관리에 필요한 명령어를 지원한다.

윈도우에서 **npm은 node.js와 함께 이미 설치**되어 있다. 명령창에서 where npm하면 알 수 있다.

리눅스는 apt install npm으로 설치, which npm으로 설치여부를 확인한다.

### package.json 생성

Node.js 프로젝트를 시작할 때 필요한 기본 설정을 만들어 보자. 명령창을 열고, 프로젝트 디렉토리로 이동한 후 npm init 명령을 입력한다. 이 명령어를 실행하지 않아도 패키지를 설치할 수 있기는 하지만, package.json 파일을 생성하기 위해 필요하다.

pjt\_dir> npm init

이 명령어를 입력하면 생성하려는 패키지 관련하여 package name, version, author 등의 정보를 입력하게끔 되어 있고 (입력하지 않고 그냥 를 해도 된다), 그러면 package.json을 생성하게 된다. package.json은 패키지명, 버전, 설명 등 관련 내용을 담고 있다.

npm을 사용하다 보면, package-lock.json이 생성되어 있곤 한다. 이 파일은 npm이 (1) node\_modules 또는 (2) package.json을 수정할 경우, 당시 의존성을 기준으로 자동으로 생성된다.

더 알아보기: package-lock.json

개발자들이 동일한 node\_module 트리를 생성해서 동일한 의존성에 따른 설치를 보장하는 장치이다. package.json에는 버전 범위가 넓게 "^1.2.3 또는 1.2.\*" 으로 적혀있지만, package-lock.json 에는 정확한 버전명이 적혀있다. 예를 들어, package.json의 ^1.2.3에 따라 의존성도 결정되었다면, npm install이 동일한 버전을 선택하여 설치한다는 보장이 없다. npm update는 package-lock.json에서, npm install은 package.json에서 설치한다.

## 1.2 web3 설치

### 버전 알아보기

명령창에서 npm view web3@버전 명령어로 버전 정보를 알아보자. --json 스위치를 붙여도 된다.

pjt\_dir> npm view web3  
  
web3@1.9.0 | LGPL-3.0 | deps: 7 | versions: 194  
Ethereum JavaScript API  
...

web3 버전 1.0의 전과 후로 문법이 크게 변경되었다. 버전은 0.20.x버전으로 머물다가, 바로 1.0으로 상향되었다. 문법이 크게 변경되어서인지 오랫동안 1.0.0 beta로 50회 넘게 버전 업이 되었다가, 2019년 중반이 되어서야 1.2.0이 발표되었다. 2023년 3월 1.9.0까지 나와 있다.

### 설치

npm으로 web3.js를 **프로젝트 디렉토리** C:\Users\jsl\Code\20171111에서 설치해보자. 필요한 라이브러리를 설치하는 명령어는 npm install이고, 버전을 명시하지 않으면 최신을 설치한다. 설치 디렉토리 아래 node\_modules이 생성된다.

pjt\_dir> npm install web3

web3 1.0 이전 버전은 web3.eth.accounts하면 계정을 바로 조회하기 때문에 비동기를 하거나 말거나 신경쓰지 않아도 된다. 이런 프로그래밍의 간편성때문에 버전 1.0 이상을 사용하기 주저한다면, 이미 설치되었던 1.0을 제거하고, 하위 버전을 명시하여 npm install <package>@<version>으로 한다.

다음과 같이 @^0.20 버전을 명시하고, 캐럿기호(^)를 적용해서 최신 버전으로 설치할 수 있다.

pjt\_dir> npm install -g web3@^0.20

설치는 전역으로 설치하거나 특정 사용자 권한으로 설치할 수 있다. 전역으로 설치할 때는 -g 스위치를 넣어주고, 이를 생략하면 사용자 권한으로 설치하게 된다. package.json에 기록된 패키지의 의존성이 갱신되어야 하는데 --save 옵션으로 해준다. --save는 생략해도 기본 옵션으로 처리된다.

더 알아보기: **gyp 오류**

gyp는 C 또는 C++ 언어로 씌여진 node.js로 만든 모듈을 컴파일하는 명령어 도구이다. 오류의 원인이 하나가 아니라서 몇 가지 시도를 해봐야 한다.

1. 버전이 높은 경우, 오류가 발생할 수 있다. Truffle이 npm < 7인 경우 작동한다고 한다 (Ganache는 Truffle의 일부) 그래서 npm 버전을 낮추도록 한다 (node 10.x.x까지 npm 6이 설치된다.)

npm install -g npm@6

1. 버전에 따라 windows-build-tools를 설치가 필요하다는 오류 메시지가 뜨면. 다음과 같이 설치하자.

npm install --global --production windows-build-tools (관리자 권한으로)  
npm install -g node-gyp  
node-gyp configure --msvs\_version=2015  
npm install web3@^1.0.0-beta.36 # 필요한 버전을 설치하면 된다.

1. python 오류라고 하면 버전을 설정할 필요가 있다. npm은 이 python 버전 정보를 node-gyp를 실행할 떄 넘겨준다.

npm install --python=python2.7 (설치된 버전을 적어준다)  
또는  
npm config set python /path/to/executable/python2.7 (절대경로를 적어준다)

### 설치 확인

라이브러리가 설치 되었는지 확인하기 위해서 npm list하면 패키지명, 버전을 출력한다. 명령창의 프로젝트 디렉토리에서 하면 자신이 설치한 목록을 출력한다.

pjt\_dir> npm list --depth=0 (글로벌을 보려면 -g를 넣는다 npm -g list --depth=0)  
201711111@1.0.0 C:\Users\jsl\Code\201711111)  
+-- @openzeppelin/contracts@4.5.0  
+-- ganache@7.0.3 설치되어 있는 ganache, 버전정보를 출력  
`-- web3@1.2.6 설치되어 있는 web3, 버전정보를 출력

더 알아보기: npm **ERR! extraneous**

패키지가 설치되었지만 package.json에 등록이 되지 않았다는 의미이다. npm init 초기화를 하였는지 확인하고, 당분간 무시하고 사용해도 된다.

### web3 테스트 실행

네트워크에 연결하려면 geth가 실행되고 있어야 한다. 여기에서는 버전만 확인해 본다. 대소문자 web3, Web3에 주의한다.

아래 >는 node.js 프롬프트이다. geth 콘솔프롬프트와 생김새가 동일하다. 어느 프롬프트에서 하는지 잘 분별해서 하기로 하자. 식별을 용이하게 하기 위해 앞에 node>라고 적었다.

pjt\_dir> node  
node> var Web3=require('web3');  
> Web3.version  
'1.2.6'

Web3.modules.라고 적고 을 누르면, 제공되는 API가 출력된다. 버전에 따라 조금 차이가 있겠지만, 뒷 부분에 모듈 Eth, Personal, Net, Shh 등이 출력되고 있다.

node> Web3.modules.<TAB>  
Web3.modules.\_\_defineGetter\_\_ Web3.modules.\_\_defineSetter\_\_ Web3.modules.\_\_lookupGetter\_\_ Web3.modules.\_\_lookupSetter\_\_  
Web3.modules.\_\_proto\_\_ Web3.modules.constructor Web3.modules.hasOwnProperty Web3.modules.isPrototypeOf  
Web3.modules.propertyIsEnumerable Web3.modules.toLocaleString Web3.modules.toString Web3.modules.valueOf  
  
Web3.modules.Bzz Web3.modules.Eth Web3.modules.Net Web3.modules.Personal  
Web3.modules.Shh

다음 화면은 web3.js를 연결하여 몇 가지 명령어를 시연하고 있다.



alt text

## 1.3 ganache 설치

지금까지 nodejs, npm, web3.js를 설치했다. 이번에는 ganache를 설치하자.

뭔 설치를 또 한다고? 그냥 geth를 쓰면 안되냐고? 물론 된다! 그러나 ganache를 사용하는 것이 편리하고 효율적이다.

ganache는 마이닝을 할 필요가 없어서 편리하다. 그래서 geth 대신 사용하자는 것이다. 뿐만 아니라, 계정을 만들지 않아도 되고 rpc, webSocket가 기본 설정되어 있어 보다 편리하다.

ganache는 이더리움 geth의 테스트용 TestRPC, 즉 geth를 대신해서 테스트하는 용도로 만들어졌다. 버전 7.0부터는 ganache-core와 ganache-cli가 ganache로 병합되었다. 그러나 ganache-cli 별칭은 그대로 존재한다.

pjt\_dir> npm install ganache

설치하고 나서, ganache를 띄워보자. 간단하게 포트만 8345로 설정하자. 기본 8545는 geth에서 사용하므로 다른 포트를 사용한다.

아래 부분에 계정 10개, 잔고를 무려 100 Ether를 채워주고 있다. 계정을 만들거나 충전할 필요 없이, 나중에 gas를 지급하거나, 입출금에 사용하면 된다. 주의! ganache를 띄울 때마다 계정은 모두 삭제되고 새로 만들어 제공된다.

pjt\_dir> node\_modules\.bin\ganache -p 8345  
Ganache CLI v6.3.0 (ganache-core: 2.4.0)  
  
Available Accounts  
==================  
(0) 0xdc3f9fd9b1205bb854e5a8bb287816bf3e9fbdf6 (~100 ETH)  
(1) 0x6d814bcd9060d1998c6d43fe01f919c16e0cf667 (~100 ETH)  
(2) 0x6ee93ca4837d1ab240a572b9b0003e28c91985f0 (~100 ETH)  
...생략...  
(9) 0xfc4aa1584398e0eb77a8a4154fed71761ba5534e (~100 ETH)  
...생략...  
Listening on 127.0.0.1:8345

## 문제 6-1: ganache 접속

### 설정하고 ganache 띄우기

이전에 geth 사설망을 개설했었다. 이번에는 geth 대신 ganache를 설정하고 실행해보자.

ganache에서는 기본으로 설정되어 있기 때문에, 복잡하게 보이는 http, ws 등의 설정은 하지 않아도 된다. 대폭 줄여서 하고, geth와 같이 파일에 저장하여 일괄 실행하자. 다음과 같이 IP 주소를 적고, 포트를 8345로 해서 접속한다. 앞으로 이 환경은 ganache@8345, 앞의 geth@8445와 다르게 명명하기로 하자.

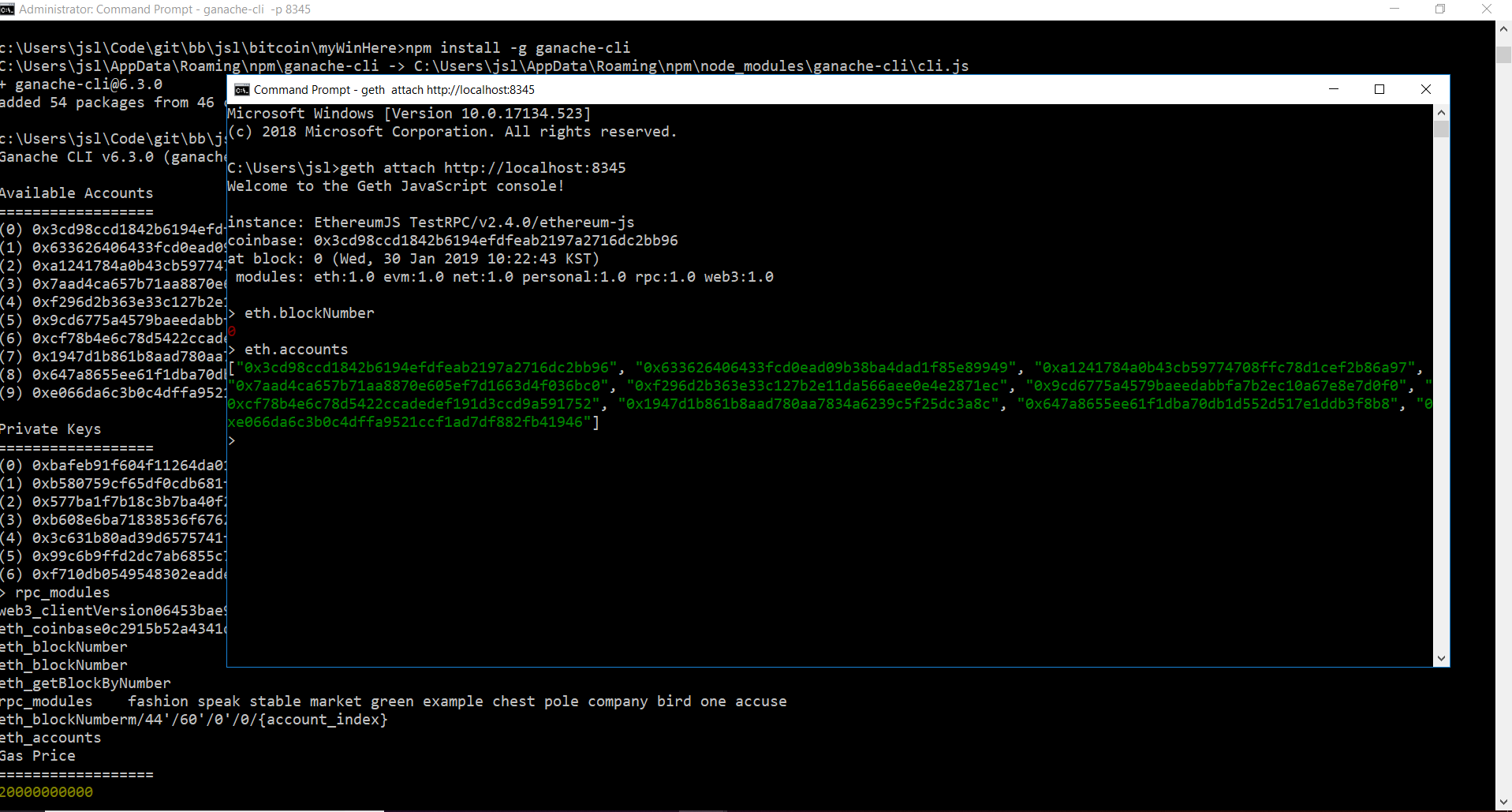
ganache-cli 또는 ganache로 적어 주어도 된다. 사용자 권한을 전역(global)으로 설치했다면, 앞의 호출 경로를 적지 않아도 된다. 파일명은 운영체제에 따라 윈도우에서는 배치 파일 확장자인 ".bat"를, 리눅스나 맥에서는 쉘 확장자 ".sh"를 적어서 저장한다.

[파일: \_ganacheNow.bat 또는 \_ganacheNow.sh]  
node\_modules\.bin\ganache-cli --host "127.0.0.1" --port "8345"

### geth attach

그리고 명령창을 하나 더 열어 접속을 한다.

앞서 사설망 연결했던 것을 기억해서, geth attach http://localhost:8345 로 접속한다 (또는 http://127.0.0.1:8345). 명령어는 앞서 geth와 비교해서 차이가 없다.



alt text

그러면 백그라운드 ganache에 접속을 하며 프롬프트가 출력된다. ganache 프롬프트에서는 geth에서 배웠던 명령어를 그대로 적으면 된다.

pjt\_dir> geth attach http://localhost:8345  
Welcome to the Geth JavaScript console!  
  
instance: Ganache/v7.0.3/EthereumJS TestRPC/v7.0.3/ethereum-js  
coinbase: 0x0000000000000000000000000000000000000000  
at block: 0 (Thu, 29 Sep 2022 14:16:15 KST)  
 modules: eth:1.0 evm:1.0 net:1.0 personal:1.0 rpc:1.0 web3:1.0  
  
>

주의하자. 프롬프트 >를 보면 geth와 동일하다. 분명하게 말하면 ganache에 접속된 geth 콘솔의 프롬프트이라서, 내용적으로는 서로 완전히 다르다. geth 콘솔의 프롬프트는 geth>라고 적어주고 있다.

여기에서 계정을 출력해보자. 아래와 같이 명령어를 실행하고 나면, 백그라운드의 ganache 로그를 관찰하자. 최초 접속을 하거나 그 이후 어떤 명령을 실행하게 되면, 반응하는 로그가 출력이 된다.

geth> eth.accounts (1)  
["0x83e790081f4117befe83d35bcdec772e066f675d", "0x4c598aa8afea968bfd45ded00879085e6f862c3b", ...생략..., "0x61ee7299a12b0874cafb2281149143b836c42122"]  
  
> web3.eth.accounts (2)  
["0x83e790081f4117befe83d35bcdec772e066f675d", "0x4c598aa8afea968bfd45ded00879085e6f862c3b", ...생략..., "0x61ee7299a12b0874cafb2281149143b836c42122"]  
>

위 명령어(1)로 하거나, 명령어(2)로 해도 문제가 없다. 명령어를 잘 살펴보면, (2)는 web3로 시작한다. web3에서 제공하는 API를 사용하고 있다. 웹에서 인터페이스하기 위해서는 web3 API를 사용해야 한다.

그래서 node가 필요하다. 자바스크립트 문법으로 코딩해야 웹에 바로 적용할 수 있기 때문이다.

# 2. 웹에서 블록체인에 연결

## 2.1 node 시작하기

앞서 node가 설치되었는지 확인하기 위해 열어 놓은 창이 있다면 그것을 사용하자. 또는 명령창을 하나 더 열어서 node를 시작한다.

좀 복잡하지만, 명령창을 여러 개 열어 놓고 작업하고 있다. 다른 프로그래밍 언어에서는 통합 개발 도구를 사용하면 그럴 필요가 없을 수 있다. 블록체인 개발에는 geth 뿐만 아니라 자바스크립트나 solidity 등 여러가지 프로그래밍 언어가 필요해서, 불가피하게 명령창 콘솔에서 하고 있으니, 잘 분별해서 사용해야 한다.

앞서 web3.js를 설치한 **프로젝트 디렉토리에서 node를 실행**한다. web3가 설치된 node\_modules에서 인식하여 호출하기 때문이다.

pjt\_dir> node  
node> var Web3=require('web3')  
undefined

## 2.2 네트워크 연결

앞서 이더리움 노드에는 HTTP, WebSocket 또는 IPC를 통해 접속할 수 있다고 설명하였다.

여기에서는 이 가운데 node에서 ganache의 HttpProvider()에 연결하고, 이를 위해 Web3 라이브러를 적용하자. 명령창에서 node를 실행하고, 노드 프롬프트 >에 다음과 같이 'web3'라이브러리를 호출하고, HTTP에 연결하자.

이 때 백그라운드에 ganache가 떠 있어야 응답을 한다.

node> var Web3=require('web3');  
> var web3=new Web3('http://localhost:8345'); // 고정IP를 사용한다면 'localhost' 대신 그 자리에 적어준다  
또는  
> var web3=new Web3(new Web3.providers.HttpProvider('http://localhost:8445'));

## 2.2 web3.eth 모듈

다음과 같이 을 누르면, web3.eth의 API가 출력된다. 앞서 geth에서 배웠던 eth 모듈의 함수와 크게 다르지 않다. 즉, 동일한 API가 web3를 통해서 제공되고 있을 뿐이다.

node> web3.eth.<TAB>

### account 조회

먼저 계정을 출력해보자. 어떤 거래이든 지급계정이 필요하다. 계정을 읽어서 from 필드에 넣어야 그 계정에서 출금 결제가 되기 때문이다.

geth에서는 다음과 같이 eth.accounts로 출력하면 된다.

geth> eth.accounts  
["0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0",...생략..., "0x6f94a3529222ad7b67305c30928df0fd2d70966d"]

그러나 주의하자. 지금은 node에서 하고 있다. node와 geth 콘솔의 프롬프트가 서로 동일해 헛갈릴 수 있으니, 자신이 어떤 환경에서 하고 있는지 분명히 알고 있어야 한다.

버전 1.0 이후, 주소는 eth.accounts[0] 이런 식으로 배열 인덱스로 읽어올 수 없다.

node> web3.eth.accounts[0]  
undefined  
>

node에서 web3를 사용할 때는, 버전1.0부터는 보통 함수명에 get이 붙고, 자바스크립트 비동기 처리 방식의 문법을 따르고 있다.

node> web3.eth.getAccounts()  
Promise {  
 <pending>,  
 ...생략}

Web3 1.0 이상에서는 함수의 callback에 Promises를 지원하고 있다. then(console.log)과 같이 결과를 출력한다.

특정 인덱스를 출력하려면 다음과 같이 한다.

node> web3.eth.getAccounts().then(ac => { console.log(ac[0]); })  
Promise {  
 <pending>,  
 ...생략...  
> 0x83e790081F4117bEfe83d35bcDEC772e066F675D

### coinbase, blocknumber, 잔고, node 관련 정보 읽기

이번에는 coinbase, blocknumber, balance, nodeInfo를 출력해보자. ganache에서는 coinbase를 정하지 않고 있어 0으로 출력한다. ganache --miner.coinbase 스위치를 적어서 정해줄 수도 있다. 하지만 ganache는 매번 새로 실행할 때마다 계정이 새로 주어지므로 주의한다.

node> web3.eth.getCoinbase().then(console.log);  
0x0000000000000000000000000000000000000000 coinbase가 정해져 있지 않으면 0으로 출력한다.  
> web3.eth.getBlockNumber().then(console.log);  
0  
> web3.eth.getBalance('0xf2a4f09c903a0a7b3450d9d16bbca14dea36aee1').then(console.log);  
0  
> web3.eth.getNodeInfo().then(console.log);  
Ganache/v7.0.3/EthereumJS TestRPC/v7.0.3/ethereum-js  
>

명령어를 파일에 저장하고 일괄 실행해 보자. 물론 위에서 한 것과 같이 IP와 계정 주소는 자신의 설정에 맞게 변경해야 한다.

[파일명: src/web3ethCommands.js]  
var Web3 = require('web3');  
var web3 = new Web3('http://localhost:8345');  
//web3.eth.getChainId().then(console.log);  
web3.eth.getAccounts(console.log);  
web3.eth.getCoinbase().then(console.log);  
web3.eth.getBlockNumber().then(console.log);  
web3.eth.getBalance('0xf2a4f09c903a0a7b3450d9d16bbca14dea36aee1').then(console.log);  
web3.eth.getNodeInfo().then(console.log);

결과의 출력이 의도했던 순서가 아닐 수 있다는 점에 주의하자. 비동기적 특성때문에 그렇다. 곧 설명을 하게 된다.

pjt\_dir> node src/web3ethCommands.js  
  
0x8078e6bc8e02e5853d3191f9b921c5aea8d7f631 10개의 계정 중 coinbase를 출력한다.  
null [ '0x8078e6Bc8e02e5853D3191f9b921C5aEA8D7F631',  
 '0x2371f0A50b4E63ED38063Ca4aE42E0Ad43965330',  
 ...생략... 모두 10개가 출력된다  
 '0xCE40E2396fafA297aDC4107c5968513f58e4C1FD' ]  
11 현재의 블록 번호를 출력한다.  
0 인식할 수 없는 계정의 잔고는 0으로 출력한다.   
EthereumJS TestRPC/v2.10.2/ethereum-js 노드정보를 출력한다.

### personal 모듈

web3.eth 아래 하위 personal 모듈이 제공된다. 다음과 같이 을 누르면 볼 수 있는데, 계정관련 web3.eth.personal.lockAccount, web3.eth.personal.unlockAccount 거래관련 web3.eth.personal.sendTransaction, web3.eth.personal.signTransaction 등의 함수를 볼 수 있다.

node> web3.eth.personal.<TAB>  
....생략...  
web3.eth.personal.currentProvider web3.eth.personal.defaultAccount web3.eth.personal.defaultBlock  
web3.eth.personal.ecRecover web3.eth.personal.extend web3.eth.personal.getAccounts  
web3.eth.personal.givenProvider web3.eth.personal.importRawKey web3.eth.personal.lockAccount  
web3.eth.personal.net web3.eth.personal.newAccount web3.eth.personal.providers  
web3.eth.personal.sendTransaction web3.eth.personal.setProvider web3.eth.personal.sign  
web3.eth.personal.signTransaction web3.eth.personal.unlockAccount

## 2.3 비동기 함수의 이해: getAccounts() 사례

### account 저장

계정 0번째는 coinbase, 즉 지급 계정이다. 계정 주소는 거래마다 입력해야 하는 정보이므로, 하드 코딩하기 보다는 변수에 저장해 놓기 위해 ac = web3.eth.getAccounts() 형태로 사용하면 편리하다.

node> var ac = web3.eth.getAccounts().then(ac => { console.log(ac[0]); })  
undefined  
> console.log(ac)  
Promise {  
 undefined,  
 ...  
 }  
undefined 저장이 되지 않았다는 의미이다.  
>

.getAccounts() 함수를 보면, 계정 주소를 받아오고, 왼쪽 변수인 ac에 저장하고 있다. 실행 결과는 ac 변수에 저장된 값이어야 할 것 같은데, 출력 내용은 undefined이다. 왜 그럴까? getAccounts() 함수가 비동기적으로 실행되기 때문에 결과값이 바로 저장되지 않기 때문이다.

잘 이해가 가지 않는가? 아직 getAccounts() 함수에 있는 코드의 실행이 끝나지 않았고, 그래서 대기해야 하고, 그 정보는 함수 호출의 끝을 대기하는 것이 마무리되어야 저장된다는 것이다.

전역변수에는 안전하게 저장될 수 있을까? 아래 프로그램의 줄4, 5의 문법이 좀 까다롭기 때문에 설명을 해보자.

* 줄4의 전역변수 my=[] 배열을 선언한다.
* 줄5의 web3.eth.getAccounts(function(err, res) { for (x in res) { my.push(res[x]) } } )는 조금 복잡하다. 반복문으로 my 배열에 하나씩 계정 주소를 저장하는 작업을 하고 있다.
  + callback함수 function(err, res) 의 인자는 err, res를 적고 있다. 이 인자는 어디서 튀어나온 것일까? 하나씩 보자. Error First, 즉 첫 자리에는 발생 오류가 차지하고, 함수 getAccounts()의 실제 결과는 res에 저장이 된다.
  + 그 res가 반복문에 쓰이고 있다. 배열에 저장하는 push()함수로 하나씩 저장하고 있다.

이제 명령문을 파일에 저장하고 일괄 실행해 보자.

[파일명: src/web3accounts.js]  
줄1 var Web3 = require('web3');  
줄2 var web3 = new Web3('http://localhost:8345');  
줄3 web3.eth.getAccounts(function(err,res) { console.log("(1) accounts: " + res); } );  
줄4 my=[]  
줄5 web3.eth.getAccounts(function(err, res) { for (x in res) { my.push(res[x]) } } );  
줄6 console.log("(2) accounts from array: " + my[0]);

저장한 파일을 프로젝트 디렉토리에서 실행해보자. 계좌주소의 출력이 길어서 중략하고 있다.

pjt\_dir> node src/web3accounts.js  
  
(2) accounts from array: undefined //전역변수에 저장하더라도 출력이 실패한다  
(1) accounts: 0x8078e...65330,0xCC16c...29065,0x3b84F...225Ea,0x22078...F0D71,0xdee20...f6C08,0xB2075...10A27,0x85fD4...0E73D,0x108c2...973A0,0xCE40E...4C1FD

결과를 보면, **출력의 순서**가 바뀌었다. (2)번이 (1)번보다 먼저 출력되었다.

비동기 명령문은 실행 순서가 앞에 있더라도, 먼저 실행된다는 보장이 없다. 명령문 (2)는 undefined가 출력 되었는데, 출력하려는 배열의 데이터를 push로 저장하는 것이 나중에 실행되어, 배열의 요소가 없기 때문에 나타나는 현상이다. 즉 실행이 마칠 때까지 기다리지 않고 되돌아오는 **비동기적** 특성때문이다.

### 비동기 함수의 Promise

위 프로그램의 줄6의 출력은 (2) accounts from array: undefined이다. 실행되었어도 왜 전역변수 account0이 undefined일까?

일반적인 프로그래밍이라면 이해할 수 없는 결과이다. 저장되지 않은 이유는 앞에서 설명한 것처럼 비동기적 특성때문에 Promise가 반환되기 때문이다.

Promise는 결과 값이 지연되는 경우의 약속 코드이고, then() 함수로 성공인 경우와 실패인 경우에 대해 약속 코드의 결과 값을 출력할 수 있다.

**자바스크립트 Promise**

비동기 함수를 처리하는 패턴으로, **결과 값을 지금이 아니고, 약속을 하여 처리에 필요한 시간이 경과한 후 반환**하게 된다. 약속은 아직 완료되지 않은 실행을 의미하며, fulfilled, rejected, pending 또는 settled (fulfilled 혹은 rejected)이 될 수 있다. Promise가 생성되면 pending 상태가 되고, Promise 내에서 비동기 작업이 성공하면 fullfilled 실패하면 rejected 상태가 된다. Promise 객체를 생성할 때 콜백 함수를 전달할 수 있다. Promise에 전달되는 콜백 함수의 인자는 Promise 객체에서 실행하는 작업이 성공했을 때 호출할 함수를 나타내는 myResolve와 실패했을 때 호출하는 함수를 나타내는 myReject이다. Promise 객체의 콜백 함수에서 myResolve()함수가 호출되면 then() 함수를 이용해서 결과를 확인할 수 있고, 실패해서 myReject가 실행되면, catch() 함수를 이용해서 확인할 수 있다.

아래 코드를 보자.

* doPromise 변수에는 Promise에 전달된 콜백 함수의 인자인 myResolve나 myReject가 저장된다. 앞에서 설명한 것처럼 Promise에서 실행하는 작업이 성공하면 Promise에 전달되는 콜백 함수 내부에서 myResolve() 함수를 호출해야 한다. 만약 실패하면 myReject() 함수를 호출한다.
* 이 값은 doPromise의 연결 함수 then() 함수의 인자 my로 넘겨가게 된다. err인 경우에 catch() 함수에서 처리한다.

[파일명: src/promiseThen.js]  
var doPromise = new Promise(function(myResolve, myReject) {  
 // 여기서 비동기 작업을 실행  
 myResolve(123); //when successful, returns 123  
 //myReject("Rejected"); //when error  
});  
  
doPromise.then(function(my){  
 console.log("RETURNED: "+my);  
 }).catch(function(err) {  
 console.log("ERROR: "+err);  
 });

그리고 실행해 보자. 앞에서 보인 코드에서는 실행 작업이 성공했다고 가정하고 myResolve(123)을 호출하였다. myResolve()함수가 호출되었으므로, doPromise의 then 함수가 호출된다. myResolve() 함수를 호출하면서 전달하는 값은 then() 함수에 전달되는 콜백 함수의 매개변수 my에 전달되고 "RETURNED: 123"이 출력된다. 실패한다고 가정하고 myResolve(123)를 주석문으로 바꾸고 myReject("Rejected")를 실행하고 어떤 결과가 나오는지 살펴보도록 한다.

pjt\_dir> node src/promiseThen.js  
  
RETURNED: 123

promise를 3으로 결과 값을 만들고 then() 함수로 보낸다. 연이어 10을 곱해서 다음 then()으로 보낸다. 결과가 하나씩 순서대로 출력되고 있다.

[파일명: src/promiseThen1.js]  
new Promise(function(resolve, reject) {  
 resolve(3)  
}).then(function(res) {  
 console.log("3 is expected..."+res); //3  
 return res \* 10;  
}).then(function(res) {  
 console.log("30 is expected..."+res); //30 as multiplied by 10 on 2 lines above  
 return res \* 10;  
}).then(function(res) {  
 console.log("300 is expected..."+res); //300  
});

일괄 실행해보자.

pjt\_dir> node src/promiseThen1.js  
  
3 is expected...3  
30 is expected...30  
300 is expected...300

### 비동기 함수의 async/await

비동기 함수는 Promise를 반환한다. 비동기적으로 실행되므로 바로 값을 반환하지 않고 일단 약속만 돌려주는 것이다. Promise가 성공적으로 완료되면 결과 값이 반환되고, 그렇지 못하는 경우에는 예외 또는 오류가 발생하고 반환된다.

**await**는 **async 함수에서 비동기 함수의 실행을 일반 함수처럼 실행시키는** 기능이다. await는 async함수에서 비동기 함수를 실행시키면 그 함수가 완전히 실행이 끝날 때까지 async함수 내부의 다른 코드의 실행을 중단한다. Promise가 실행되고 난 후에 그 결과 값(fullfilled(성공)인 경우 결과 값, rejected(실패)이면 거절된 값) 을 받게 된다. Promise가 실행된 후, await 다음에 있는 코드를 순서대로 실행시킨다. **await는 async 함수 내부에서만** 사용할 수 있고, 전역으로는 쓰일 수 없다.

async/await를 사용하면, await가 결과가 나올 때까지 대기상태로 있기 때문에 **.then을 사용하지 않아도 된다**.

[파일명: src/asyncAwait.js]  
async function hello() {  
 await 1000;  
 return "hello";  
}  
hello().then(console.log);

실행해보자. await를 1000ms만큼 대기 후, hello가 출력된다.

pjt\_dir> node src/asyncAwait.js  
  
hello

### 비동기적 getAccounts 실행

먼 길을 돌아왔다. 이제 계정 정보는 함수에서 구해서 사용하면 된다. 우선은 하드코딩해서 사용하고, 나중에 다음과 같이 함수에서 계정 정보를 받아와서 프로그래밍하자.

[파일명: src/web3account0.js]  
줄01 var Web3 = require('web3');  
줄02 var web3 = new Web3('http://localhost:8345');  
줄03 var account0;  
줄04 async function getAccount0() {  
줄05 accounts = await web3.eth.getAccounts(); # 비동기 함수인 getAccounts()가 끝날 때까지 기다림  
줄06 account0=accounts[0];  
줄07 console.log("local account0: " + account0);  
줄08 }  
줄09 getAccount0()  
줄10 console.log("global account0: " + account0);

저장된 파일을 실행한다.

pjt\_dir> node src/web3account0.js  
  
global account0: undefined  
local account0: 0xbdC3f3ddBd068E0e739b926aaA527BF1D1Eb7EF0

출력을 보면 전역변수는 출력이 실패하고, 로컬에서는 성공하고 있다.

* 줄3 전역변수 account0을 선언하고, 나중에 계정 주소 0을 저장한다. 물론 저장은 실패하게 된다.
* 줄4 getAccount0() 함수를 async로 선언해서 비동기 실행을 가능하게 한다.
* 줄5 await를 적용하여 getAccount0()함수를 호출한다. 이렇게 하는 것은 비동기적 실행을 하고, 좌측의 전역 변수인 account0에 저장한다.
* 줄7 await을 통해 getAccounts() 함수가 종료될 때까지 기다리므로, account0를 출력하면 계정 정보가 제대로 보인다. async 함수에서 계정을 조회하고 사용해야 한다는 것을 알 수 있다. 나중에 그렇게 사용하게 된다.
* 줄10은 저장된 계정 주소를 출력하고 있다. 그러나 아래 출력은 실패하고 있고, 저장되지 않은 것을 볼 수 있다.

## 2.3 web3.utils 모듈

몇 가지 web3.utils 모듈의 함수를 호출해보자.

* 줄3 randomHex()는 32바이트 무작위 hex를 생성한다. 64자리 문자열이 출력된다.
* 줄4 keccak256()는 입력의 keccak256을 계산한다.
* 줄5 isAddress()는 입력이 이더리움 주소인지 확인한다.
* 줄6 utf8ToHex()는 utf8 한글을 hex로 변환해서 출력하고 있다.
* 줄7 toWei() Wei로 변환한다.

[파일명: src/web3utils.js]  
줄1 var Web3 = require('web3');  
줄2 var web3 = new Web3('http://localhosst:8345');  
줄3 console.log("random hex: ", web3.utils.randomHex(32)); //generate random 32 byte hex  
줄4 console.log("234 hash: ", web3.utils.keccak256('234')); //calculate keccak256 for '234'  
줄5 console.log("is address?: ",web3.utils.isAddress('0xf2a4f09c903a0a7b3450d9d16bbca14dea36aee1')); //check if the input is a valid address  
줄6 console.log("안녕 in Hex: "+web3.utils.utf8ToHex('안녕'));  
줄7 console.log("wei of 1 ether: "+web3.utils.toWei('1', 'ether'));

명령창에서 유니코드를 지원한다면 console.log() 함수의 한글은 깨지지 않고 출력된다.

pjt\_dir> node src\web3utils.js  
random hex: 0x7cf571f949a30e1dd290428bfde58d69aab57c320be8b4048e3539b24402a0ec  
234 hash: 0xc1912fee45d61c87cc5ea59dae311904cd86b84fee17cc96966216f811ce6a79  
is address?: true  
안녕 in Hex: 0xec9588eb8595  
wei of 1 ether: 1000000000000000000

# 3. Web3 페이지

## HTML 작성

간단한 Web3 페이지를 만들어보자.

잠깐 생각해 보자. 지금까지 우리가 알고 있는 웹은 서버가 필요하다. 웹 페이지를 보는 것 부터 서버에 접속해서 웹페이지의 내용을 받아와서 브라우저가 화면에 보여준다. 웹페이지에서 버튼을 누르거나 클릭을 하거나 서비스를 요청하고, 그 요청을 받아 처리하는 것도 모두 서버가 필요하다.

여기서 만드는 것은 웹서버를 요구하는 일반적인 웹 페이지가 아니라는 점을 명심하자. 블록체인에 결과를 받아오는, 그러니까 블록체인이 서버가 되는 웹페이지를 만든다. 이런 페이지를 Web3로 만든다.

프로젝트 디렉토리 아래 scripts 폴더를 만든다. html을 편집기를 이용해서, 파일을 scripts폴더에 저장한다.

[파일명: scripts/simpleCoinbase.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');  
 var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
 function displayCoinbase() {  
 var web3version = web3.version; //web3.version.api  
 //web3@0.20.x  
 //document.getElementById('web3ver').innerText = 'web3 version 0.20.x: '+web3version.api;  
 //document.getElementById('web3coinbase').innerText = 'coinbase: ' + web3.eth.coinbase;  
 //web3@1.2.x  
 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="displayCoinbase();">Display Coinbase</button>  
 <div></div>  
 <div id="web3ver"></div>  
 <div id="web3coinbase"></div>  
</body>  
</html>

## 서버 올리기

HTML 코딩이 되었으면 웹서버를 띄운다.

잠깐! 블록체인은 서버가 필요없는 serverless 환경이라고 강조하지 않았던가? 맞다. 서버가 필요없다. 그렇다면 웹서버는 왜 필요한가? 웹페이지를 띄우기 위해 필요하다. 띄우고 나서, 웹페이지에서의 호출하는 서비스는 블록체인에서 제공된다.

포트번호는 의도적으로 8045을 사용한다. 8000번은 누구나 쉽게 접근가능하고, Python 서버는 가벼운 대신 보안을 통제할 수 없으므로 잠깐 사용하는 동안이라도, 매우 희박한 확률이겠지만, 문제가 될 수 있다.

명령창을 열어, Python 3.x 버전에서는 python3 -m http.server 8045 --bind xxx.xxx.xxx.xxx(윈도우에서는 python3 대신 python을 입력)로 웹서버를 띄운다.

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 -

## Web3 페이지 열기

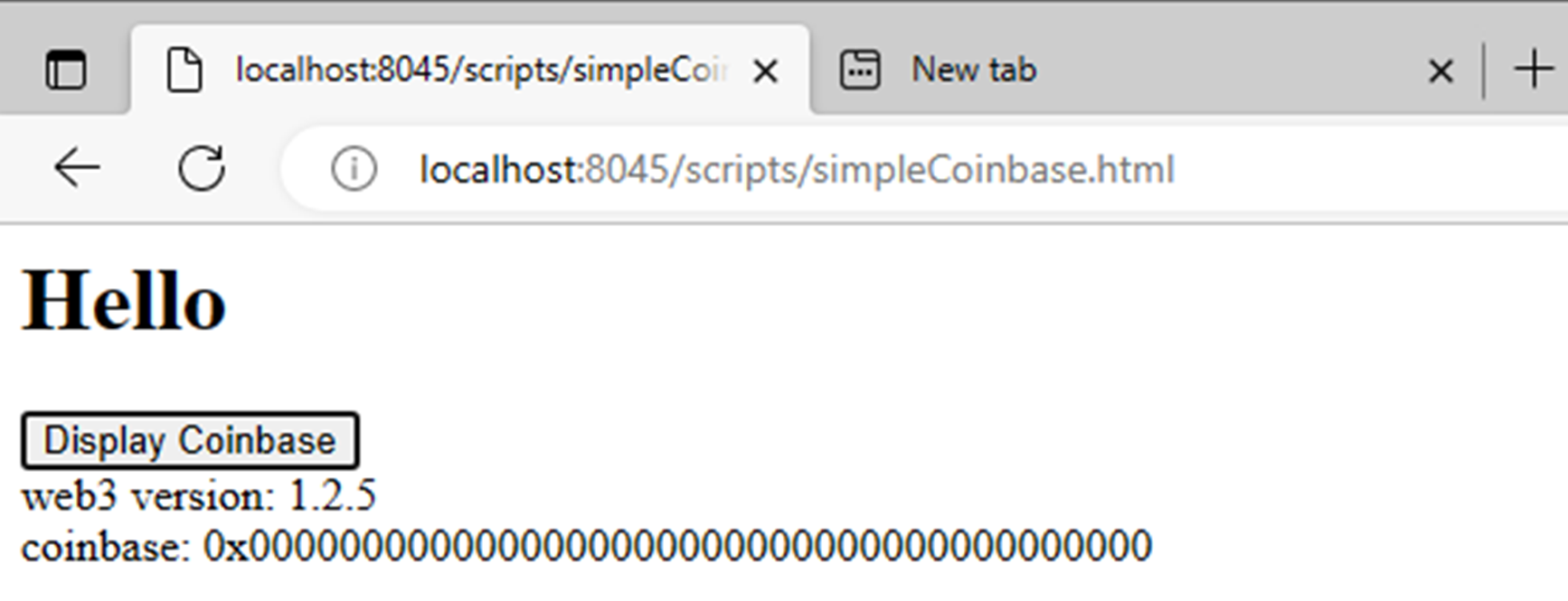
다음으로 웹브라우저에서 URL을 넣어서 띄운다.

url은 호스트명, 포트번호, 그 다음에 파일경로는 웹서버를 띄운 프로젝트 디렉토리를 기준으로 http://<url>:<port>/<경로/파일명> 적어준다.

브라우저 주소창에 로컬호스트, 포트 8045, 프로젝트기준으로 scripts/simpleCoinbase.html을 적고 해보자.

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

다음 화면은 제작한 웹페이지를 띄워 coinbase 등을 출력하고 있다.



## 연습문제

1. web3.js는 http, websocket을 모두 지원한다. OX로 답하시오.
2. web3.js는 버전1.0 이후 비동기 프로그램을 지원하지 않는다. OX로 답하시오.
3. node단말에서 web3의 버전을 출력하는 프로그램을 작성하시오.
4. ganache는 테스트 환경으로 마이닝을 하지 않는다. OX로 답하시오.
5. 노드단말에서 http로 연결하는 프로그램을 작성하시오.
6. 노드단말에서 블록번호를 출력하시오.
7. 노드단말에서 async/await를 적용하여 첫째 주소를 출력하는 프로그램을 작성하시오.
8. 노드단말에서 async/await를 적용하여 첫째 주소의 잔고를 출력하는 프로그램을 작성하시오.
9. 웹서버를 띄우기 위해 간단한 파이썬의 서버를 적용해도 된다. OX로 답하시오.
10. coinbase를 출력하는 웹페이지를 작성하시오.
11. 다른 환경의 계정 간 이체

* ganache@8345를 백그라운드로 띄우고, 노드 단말에서 coinbase주소를 출력하시오.
* 위 문제를 geth@8445를 띄우고 실행해보세요. geth에서의 coinbase를 출력하시오.
* ganache@8345의 coinbase에서 geth@8445의 coinbase로 1 ether를 송금하시오. 송금 성공이면 잔고변화를 출력하고, 실패이면 그 이유를 간단히 설명하시오.

1. 환전 계산

* ganache를 연결하고, 현재 coinbase의 잔고를 ether로 출력하시오.
* 현재 한화 시세를 인터넷에서 구하시오 (프로그램으로 하지 않는다). 한화로 환산하여 출력하는 코드를 작성하시오.

1. coinbase를 account0에서 account1로 변경하고, 출력하는 코드를 작성하시오.
2. ganache@8345를 백그라운드로 띄우고, node에서 다음의 실행 결과를 출력하시오.
   1. 주소0, 주소1 출력
   2. 주소0, 주소1의 잔고 출력
   3. 주소0에서 주소1로 1111 wei 전송 (callback 함수에서 전송메시지 출력)
   4. 해시 출력 ((3)의 callback으로 출력하지 말고, 전송 완료 후 출력)
   5. gas와 gas 가격을 곱해서 출력
   6. nonce 출력
   7. 주소0과 주소1의 잔고 변화 출력 ((2)로부터 증감이 얼마나 되었는지)
   8. 잔고의 합계를 더하는 과정 출력
   9. 잔고의 합계 출력

출력 예시

(1) ac0: 0xEb670...03ABd ac1: 0xF724C...9aDCE  
(2) bal Of ac0: 999997530038074448552 ac1: 1000000000000000019998  
(3) callback - sending ac0 -> ac1  
(4) transactionHash: 0x4fe17...35c6d  
(5) gas costs: 180000000000000  
(6) nonce: 39  
(7) bal diff1: ... bal diff2: ...  
(8) adding balance  
sum: 0 adding 0 bal: 999997488038074447441  
sum: 999997488038074400000 adding 1 bal: 1000000000000000021109  
sum: 1.9999974880380744e+21 adding 2 bal: 1000000000000000000000  
sum: 2.9999974880380746e+21 adding 3 bal: 1000000000000000000000  
sum: 3.9999974880380746e+21 adding 4 bal: 1000000000000000000000  
sum: 4.999997488038075e+21 adding 5 bal: 1000000000000000000000  
sum: 5.999997488038075e+21 adding 6 bal: 1000000000000000000000  
sum: 6.999997488038075e+21 adding 7 bal: 1000000000000000000000  
sum: 7.999997488038075e+21 adding 8 bal: 1000000000000000000000  
sum: 8.999997488038075e+21 adding 9 bal: 1000000000000000000000  
(9) balance total: 9.999997488038075e+21