

Web3: Interacting with Ethereum





Grace Kull
Simon Guo

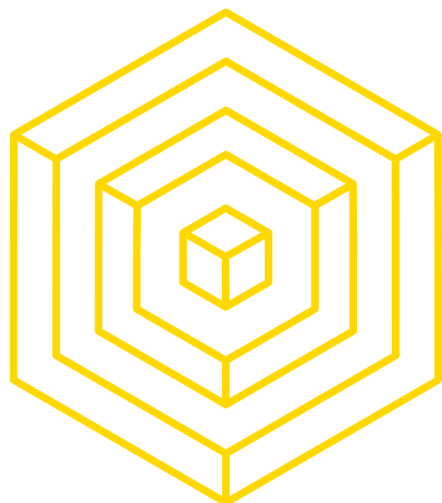


BLOCKCHAIN
AT BERKELEY



LECTURE OUTLINE

- 1  ETHEREUM ARCHITECTURE
- 2  WEB3.JS OVERVIEW
- 3  WEB3.JS FUNCTIONS
- 4  DEMO

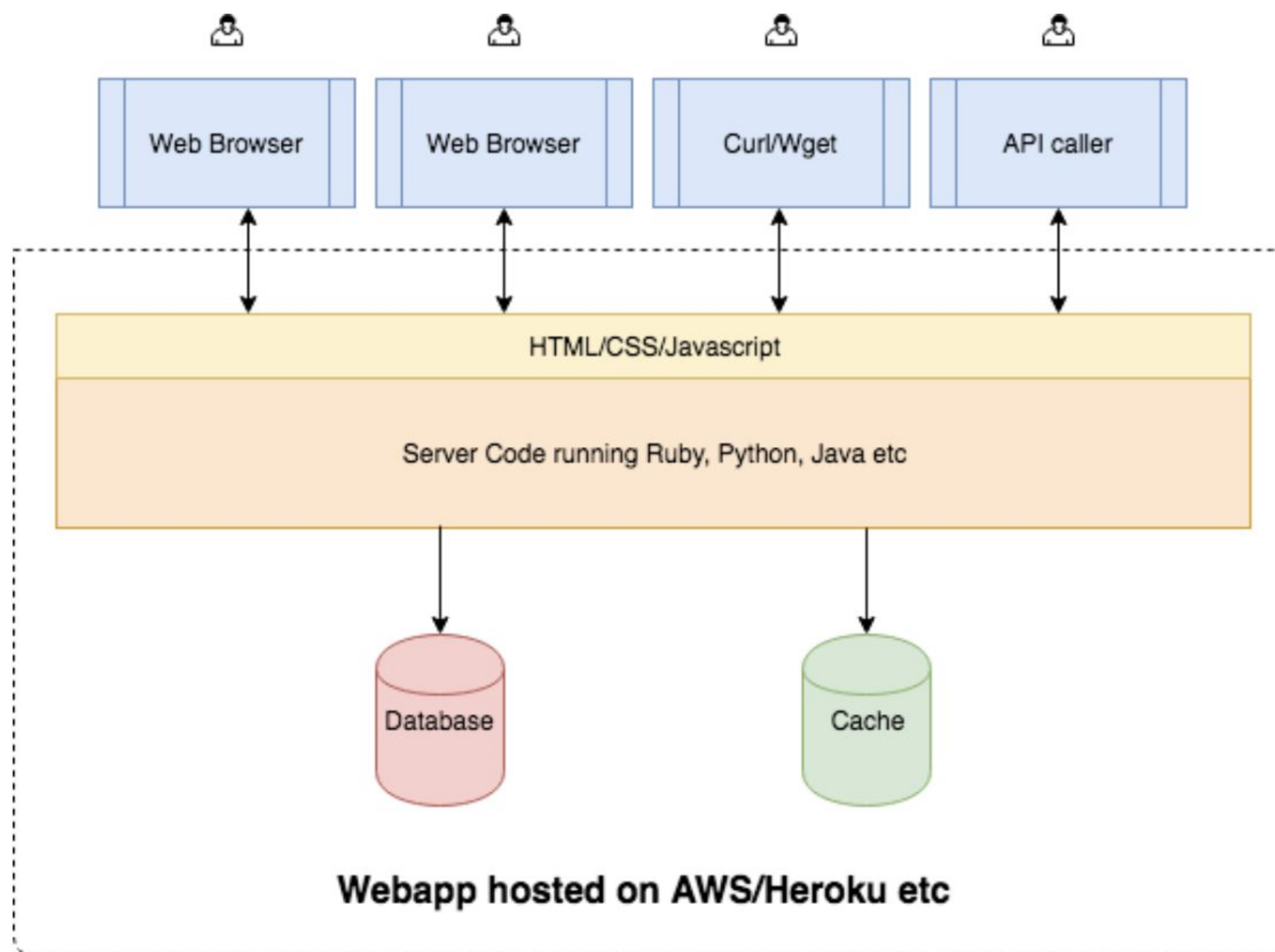


1 ETHEREUM ARCHITECTURE



STANDARD WEB

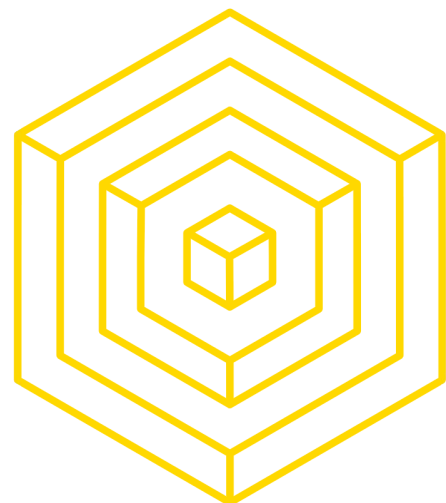
LOTS OF BOXES



What the user interacts with

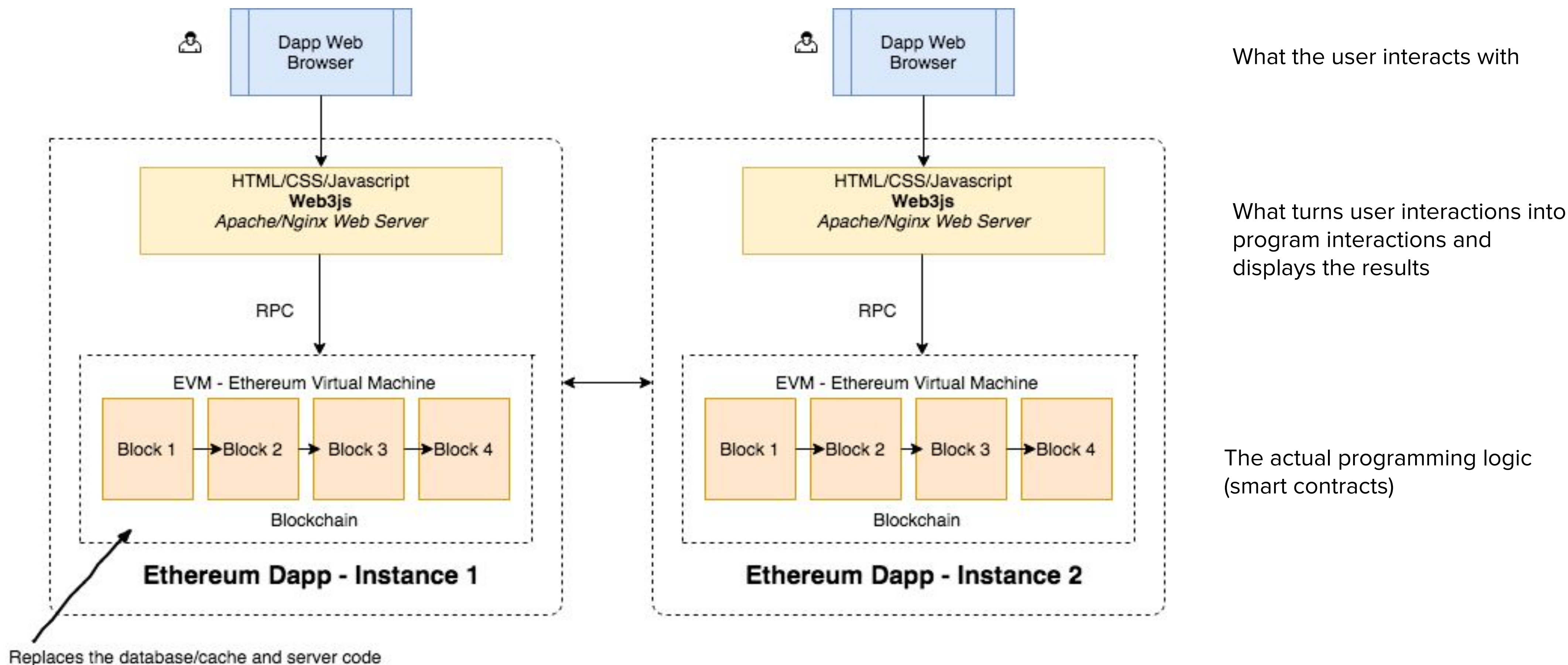
What turns user interactions into program interactions and displays the results

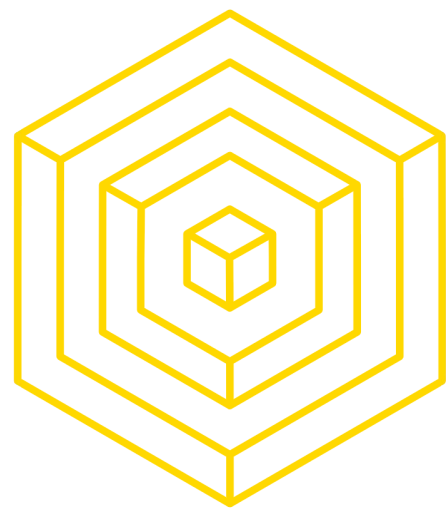
The actual programming logic



WHAT DOES ETHEREUM LOOK LIKE?

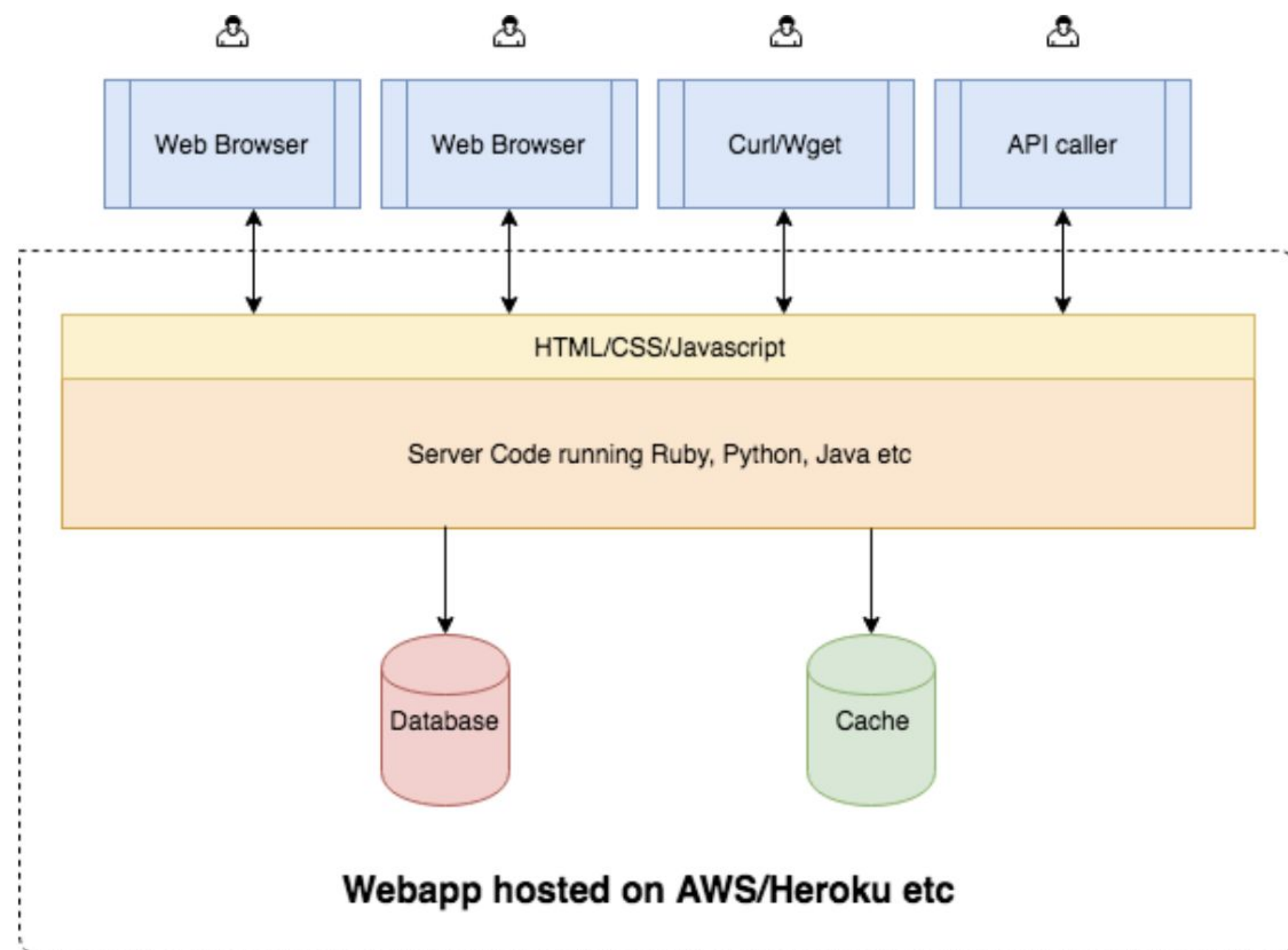
LOTS OF BOXES





WHAT ABOUT STANDARD WEB?

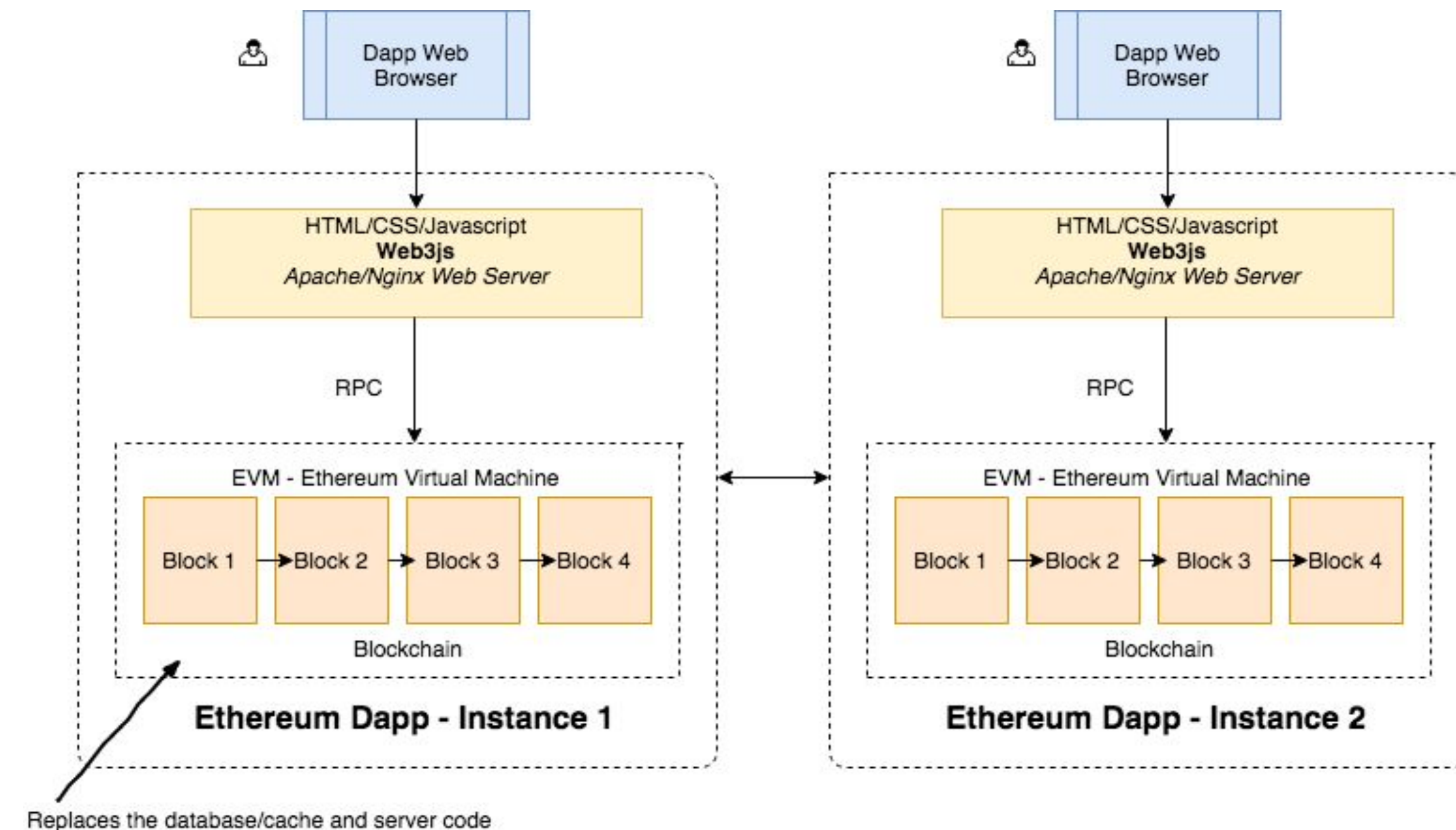
EVEN MORE BOXES



USER LEVEL

FRONT END

BACK END



Standard web infrastructure: central party hosts and serves content, everyone accesses the same instance

Decentralized web: no central party serving content. Everyone runs the programs they use on their own interconnected machines.



HAVE WE BEEN USING THIS?

KIND OF, BUT NOT REALLY

- Web3.js is the entrance to the Ethereum blockchain from the client side of a dApp.
- Communicates to nodes on network using RPC (Remote Procedure Calls)
- Web3 contains both the eth object and the shh object
 - Web3.eth - for Ethereum blockchain interaction
 - Web3.eth.abi
 - Web3.eth.accounts
 - Web3.eth.personal
 - Web4.shh - for Whisper interaction (communication between dApps)
 - Web3.bzz - for Swarm interaction (communication between nodes, 'server')



HIGH-LEVEL EXAMPLE

WHAT GOES ON UNDER THE HOOD

Let's say you want to use a dApp that allows you to vote for candidate A or B. Here's an idea of what happens when you click "A" and send the transaction:

1. Frontend receives input
2. Relevant inputs and functions (let's say `Vote("A")`) are passed into Web3 contract instance, matched against ABI
3. Transactions and events are fired
4. Web3 instance queries blockchain for updated data (`votes["A"] = 1`)
5. New data displayed on frontend



2 WEB3 OVERVIEW





WHAT IS WEB3.JS?

POKE POKE

web3

a collection of libraries that allows a user to **interact** with the Ethereum platform and smart contracts



DIFFERENT IMPLEMENTATIONS

WEB3 DOT...

Python **Web3.py**

Haskell **hs-web3**

Java **web3j**

Easy to make, most used languages now have some kind of implementation in progress, but we will mainly focus on:

JavaScript **web3.js**



WHAT CAN WEB3.JS DO?

EVERYTHING

- It is the official DApp API that is run on all of the Ethereum nodes
 - Interfaces with Ethereum nodes from the network using **JSON-RPC** calls
- Main Capabilities:
 - Interact with contract functions
 - Deploy contracts
 - Send data to contracts and initiate transactions
 - Query the blockchain for data
 - Includes logged events by any contract along with block data
- Can have our back-end on chain, and our interface off chain



WHAT IS RPC?

POKE POKE

RPC

Remote Procedure Call (RPC) is a protocol that one program can use to **request a service** from a program located in another computer in a network without having to understand network details.

[SOURCE](#)



JSON-RPC API

CONTEXT BEHIND ETHEREUM'S API

- JSON is a lightweight data-interchange format
- RPC is used to make remote function calls
- JSON-RPC is a stateless, lightweight remote procedure call (RPC) protocol.
 - Defines several data structures and the rules around their processing.
 - It is transport agnostic in that the concepts can be used within the same process, over sockets, over HTTP, or in many various message passing environments

Side note: Serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment).



ETHEREUM ABI

EXPOSING CONTRACT METHODS

ABI = Application Binary Interface

- An ABI is how you **call** functions in a contract and **get data back**
 - It determines how functions are called and in which binary format information should be passed from one program component to the next
- Why is it necessary?
 - You need a way to specify **which function** in the contract to invoke as well as guarantee to **what type of data is returned**
- Not part of the core Ethereum protocol, you can define your own ABI - but easier to comply with format provided by web3.js



ETHEREUM ABI

ANOTHER ANALOGY

API = Application **Programming** Interface

ABI = Application **Binary** Interface

- So therefore an ABI is an API at a lower level?
- Contract code is stored as bytecode in binary form on the blockchain under a specific address
 - You can access the binary data in the contract through the ABI
 - The ABI defines the structures and methods that you will use to interact with binary contract (just like an API)

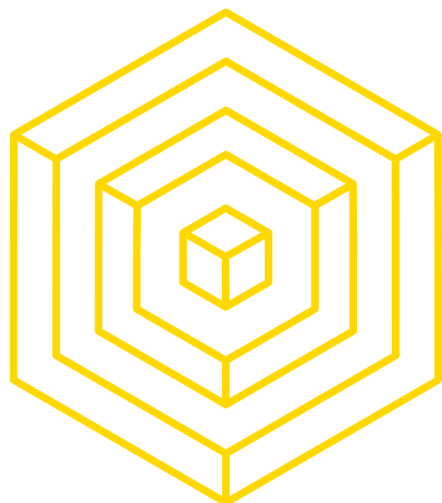


ETHEREUM ABI

THE LOVELY ABI FORMAT

```
contract Test {  
  function Test(){ b = 0x12345678901234567890123456789012; }  
  event Event(uint indexed a, bytes32 b);  
  event Event2(uint indexed a, bytes32 b);  
  function foo(uint a) { Event(a, b); }  
  bytes32 b;  
}
```

```
[{  
  "type": "event",  
  "inputs": [{"name": "a", "type": "uint256", "indexed": true}, {"name": "b", "type": "bytes32", "indexed": false}],  
  "name": "Event"  
}, {  
  "type": "event",  
  "inputs": [{"name": "a", "type": "uint256", "indexed": true}, {"name": "b", "type": "bytes32", "indexed": false}],  
  "name": "Event2"  
}, {  
  "type": "function",  
  "inputs": [{"name": "a", "type": "uint256"}],  
  "name": "foo",  
  "outputs": []  
}]
```



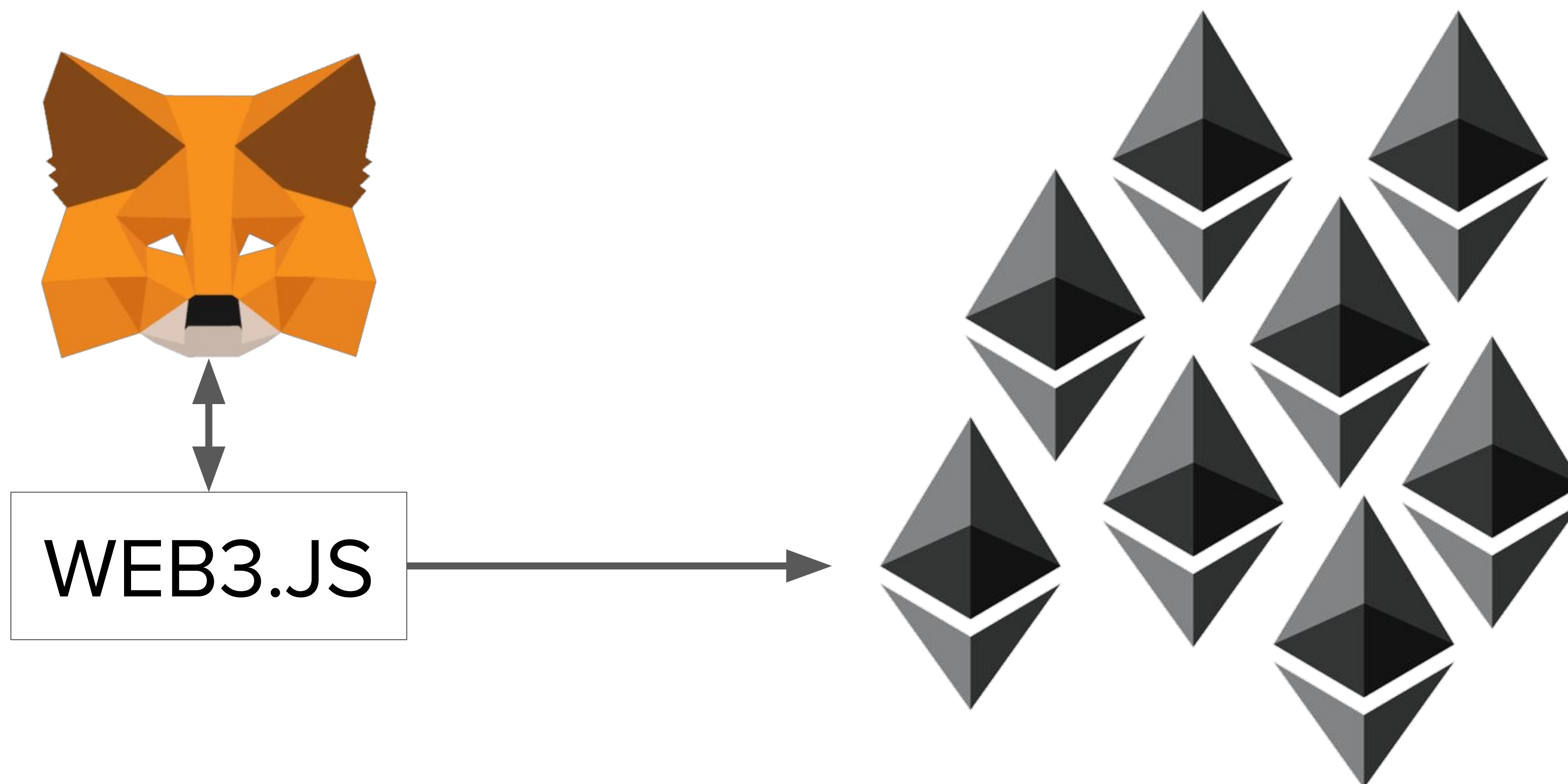
3 WEB3.JS USAGE

Sources for this section: [Source 1](#), [Source 2](#), [Source 3](#)



WEB3.JS: Connecting the Blockchain

NOW PUTTING IT ALL TOGETHER

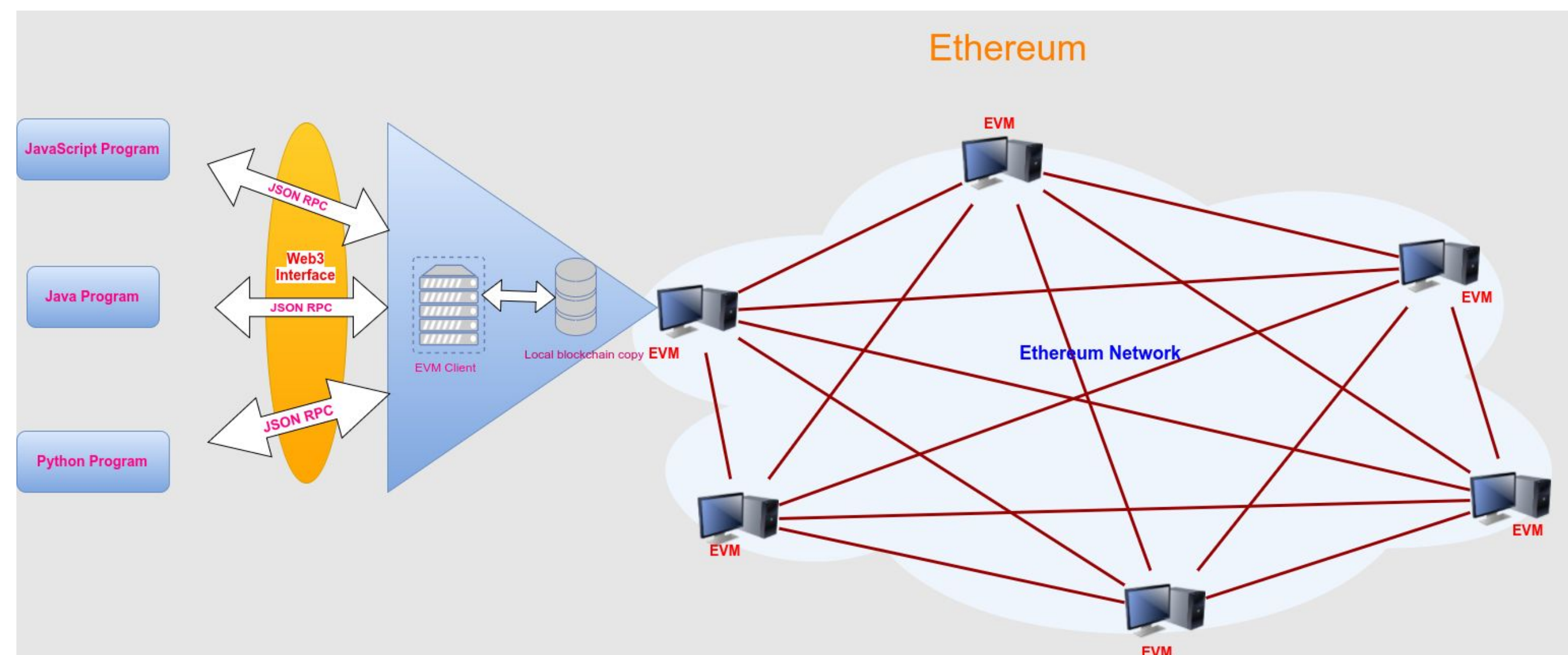




LET'S GET STARTED

READING AND WRITING TO THE CHAIN

- Web3.JS is our key to reading and writing the Ethereum blockchain
- Uses JSON RPC to talk to chain (Remote Procedural Call)
- Need to connect to an Ethereum node - can use:
 - Infura (most efficient)
 - Parity
 - Geth



*All of the following code can be done in your terminal or a javascript file in your text editor



FIRING UP WEB3.JS

HELLO WORLD!

- Let's start using web3.JS
- First fire a node up by typing “node”, then:

```
const Web3 = require('web3')
```

- Next, if you are using Infura:

```
const rpcURL = "https://mainnet.infura.io/YOUR_INFURA_API_KEY"
```

```
const web3 = new Web3(rpcURL)
```

- With this, the ethereum node is connected and the Web3.JS library is at your disposal

Side Note: You can use Geth and Parity also but that requires more time to sync up with the full chain



SIMPLE WEB3.JS CALL

ACCOUNTS

We can get the balance of an account with Web3.JS

To do this (With account representing the account public address):

```
const account = "0x90e63c3d53E0Ea496845b7a03ec7548B70014A91"  
web3.eth.getBalance(address, (err, wei) => {  
  balance = web3.utils.fromWei(wei, 'ether')  
})
```

The MetaMask extension exposes the web3 API by an **injected web3 object** which you can access via JavaScript and **does not support most synchronous** web3 API methods.

If user has installed MetaMask, to get their current account, do: `web3.eth.accounts[0]`



READ DATA FROM SMART CONTRACTS

GRABBING CONTRACTS

```
new web3.eth.Contract(jsonInterface, address, options)
```

`jsonInterface`: Contract ABI

`address`(optional): This address is necessary for transactions and call requests and can also be added later using `myContract.options.address = '0x1234...'`

`options`(optional): The options of the contract. Some are used as fallbacks for calls and transactions.

```
const myContract = new web3.eth.Contract([...], '0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe', {  
  from: '0x1234567890123456789012345678901234567891', // default from address  
  gasPrice: '20000000000' // default gas price in wei, 20 gwei in this case  
});
```



CONT...

GRABBING CONTRACT FEATURES

- Now that a variable has been assigned to the contract using Web3.JS, we can call any (public) function that the smart contract has within it
- This is the feature that makes Web3.JS so attractive to developers
- Suppose we have the variable “myContract” from the previous slide

```
const myContract = new web3.eth.Contract([...], '0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe', {  
  from: '0x1234567890123456789012345678901234567891', // default from address  
  gasPrice: '20000000000' // default gas price in wei, 20 gwei in this case  
});
```

- We can call any function from the contract stored within the const variable using: `myContract.methods.methodCall()`
Where the underlined portion is any method/function in the contract



SMART CONTRACT EVENTS

EVENTS, EVENTS, EVENTS

- Smart contracts may emit events that we want to catch so we can reflect the change within the UI or frontend
- We can listen to these events using the “getPastEvents()” function
- First let’s store our contract in a variable to interact with

```
const myContract = new web3.eth.Contract(abi, address)
```

Remember, ABI is the application binary interface of the contract, and address is the contract address

- Now let’s suppose we are emitting an event from our smart contract called “hasVoted”

*This is the same event from the voting smart contract demo



CONT...

EVENTS, EVENTS, EVENTS

- Now we can use `getPastEvents` to listen to all the times `hasVoted` was emitted

```
contract.getPastEvents(  
  'hasVoted',  
  {  
    fromBlock: 6321265,  
    toBlock: 'latest'  
  },  
  (err, events) => {  
    console.log(events) }  
)
```

- In place of 'hasVoted', we could put 'AllEvents' to listen to all events
- This code logs all instances that our event was emitted
- "fromBlock" and "toBlock" specify the block range to go back/listen to the event



Web3.JS SUMMARY

SO MUCH TO LEARN...

- So far, this has only given a glimpse of what Web3.JS can do
- These are some very important ways that current developers are using web3, especially frontend blockchain developers
- Web3.JS also can be used to create signed transactions from accounts with their private keys
- These transaction objects can be used to deploy smart contracts
- Example:

```
const txObject = {  
  nonce:    web3.utils.toHex(txCount),  
  gasLimit: web3.utils.toHex(1000000),  
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),  
  data: data  
}
```



EXTRA FUNCTION CALLS

CHECKING OUT BLOCKS

- get the latest block number
- get the average gas price currently for the network
- Calling a certain method in a instantiated contract

```
web3.eth.getBlockNumber().then(console.log)
```

```
web3.eth.getGasPrice().then((result) => {  
  console.log(web3.utils.fromWei(result, 'ether'))  
})
```

```
...  
contract.methods.transfer(account2, 1000)
```

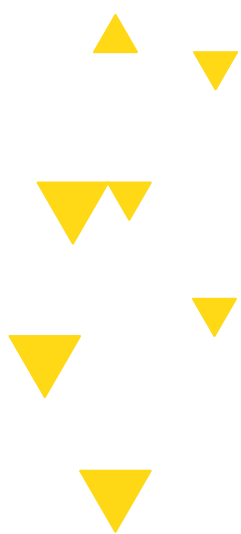
Find out more here on the official documentation of web3.js:

<https://web3js.readthedocs.io/en/v1.2.6/>



Attendance

<https://tinyurl.com/sp20-dev-decal-4>





4 DEMO + HOMEWORK WALKTHROUGH

