The Architecture of a Web 3.0 application

[Source](https://www.preethikasireddy.com/post/the-architecture-of-a-web-3-0-application)

# Web 2.0

* **Database:** store data
* **Backend code:** handle request from then frontend and interact with the database
* **Frontend code:** user interaction with the application.

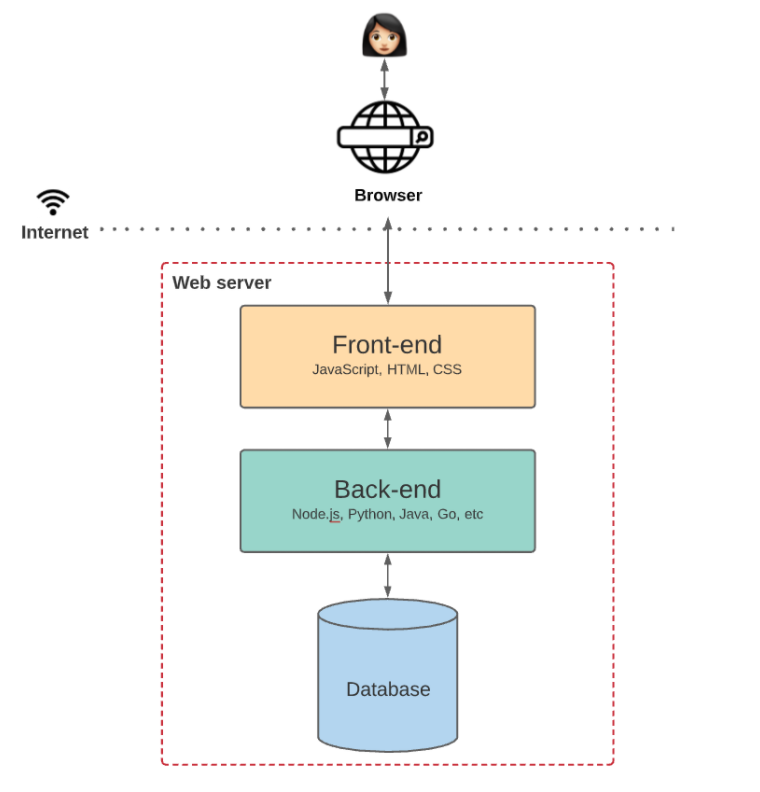


Figure a: Web 2.0 architecture

# Web 3.0

## Overall

* **No centralized database**.
* **No centralized backend codes.**
* **Decentralized state machine** maintained by anonymous nodes on the internet.
* State machine: EVM, Ethereum virtual machine.
* No single entity controls this decentralized state machine, it is collectively **maintained by everyone in the network**.
* Build a blockchain application by deploying code on this shared state machine.
* The frontend can be built on the web 2.0 solution and interact with the application logic in smart contracts.
* The Ethereum blockchain is often touted as a “world computer”: **deterministic state machine** maintained by a peer-to-peer network of nodes.
* Data can only be written to the Ethereum blockchain — you can **never update existing data**.
* **Smart contract** is a program that runs on the Ethereum blockchain and **defines the logic behind the state changes** happening on the blockchain.
* **EVM execute** logic in **smart contract compiled** and **process the state changes** that happen globally.

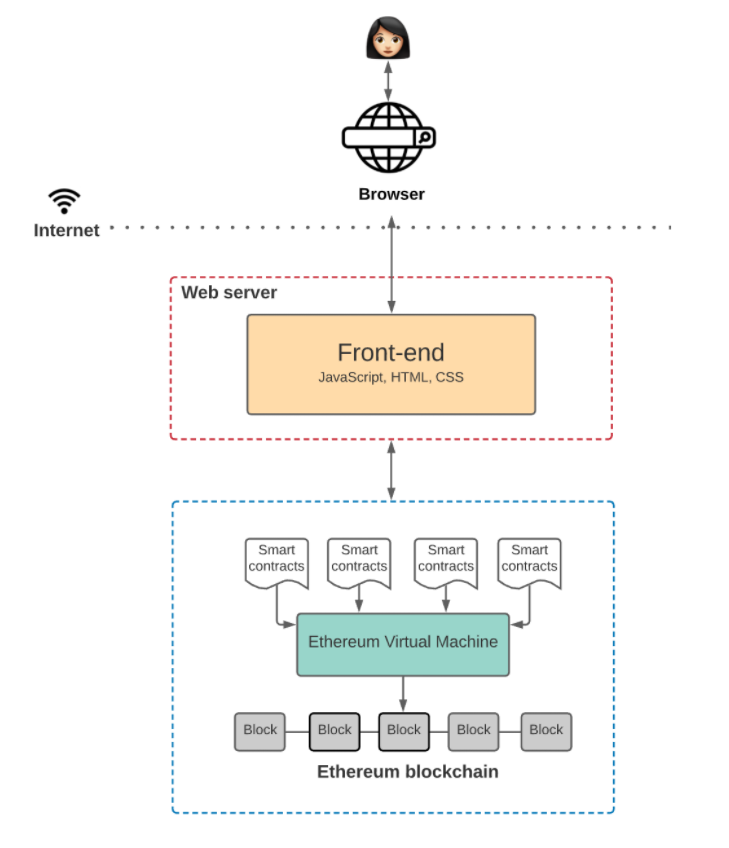


Figure b: Web 3.0 architecture overall

## Communication between Frontend and Smart Contracts

* **Nodes broadcast** request for a **transaction** to be executed.
* **Miners execute transaction** and propagate the resulting changing state to the network.
* 2 ways to broadcast new transaction:
  + Set up **your own node**
  + Use nodes provided by **third-party**
* **Providers (Ethereum client):** **nodes** that you connect with when you need to **interact with the blockchain**. As we see above, can be one of yourself or one of a third-party (like Infura, Alchemy or Quicknode).
* Provider implement a JSON-RPC specification: stateless, lightweight remote procedure call protocol that defines several data structures and the rules for their processing. It’s transport-agnostic (support multiple protocol, http, sockets, etc.).
* **Read** the state stored on the blockchain: **connect** to the blockchain through a **provider**.
* **Write** to the state: **sign transaction** using **private key** before submit to the blockchain.
* **Metamask**: tool to handle **key management**. Allow users to **sign transaction**. Metamask a connection to the nodes provided by Infra, so is both **provider and a signer**.

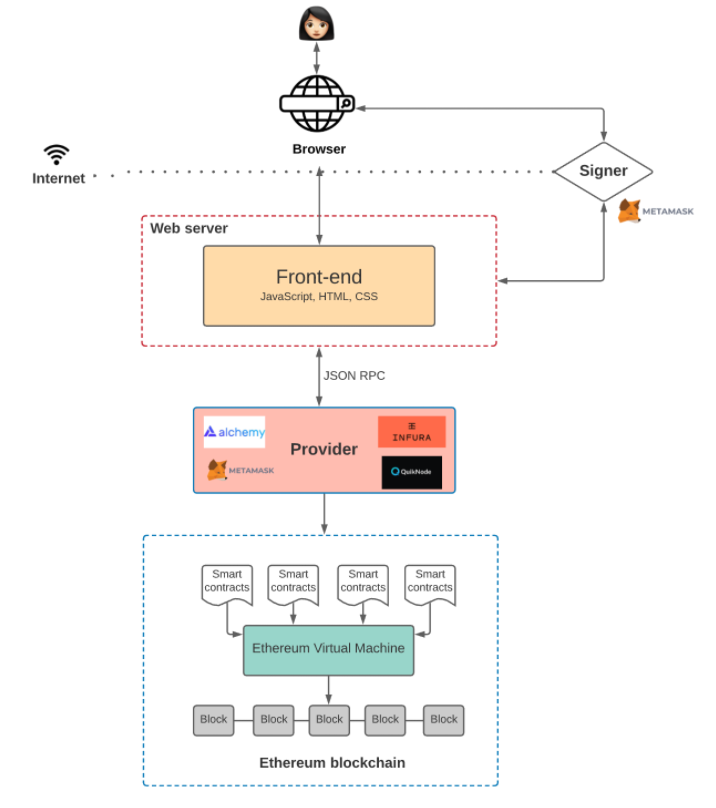


Figure c: Web 3.0 communication between frontend and blockchain

## Storage on the Blockchain

* With Ethereum, the user **pays** every time they **add** new **data** to the blockchain.
* **Storing** everything on the blockchain gets really **expensive**, really fast.
* One way to combat this is to use a **decentralized off-chain storage** solution, like IPFS or Swarm.
* IPFS is a distributed file system for storing and accessing data.
* Swarm’s incentive system is built-in and enforced through smart contracts on the Ethereum blockchain for storing and retrieving data.
* To build a truly decentralized app you might choose to host your frontend on a decentralized storage solution, like IPFS or Swarm.

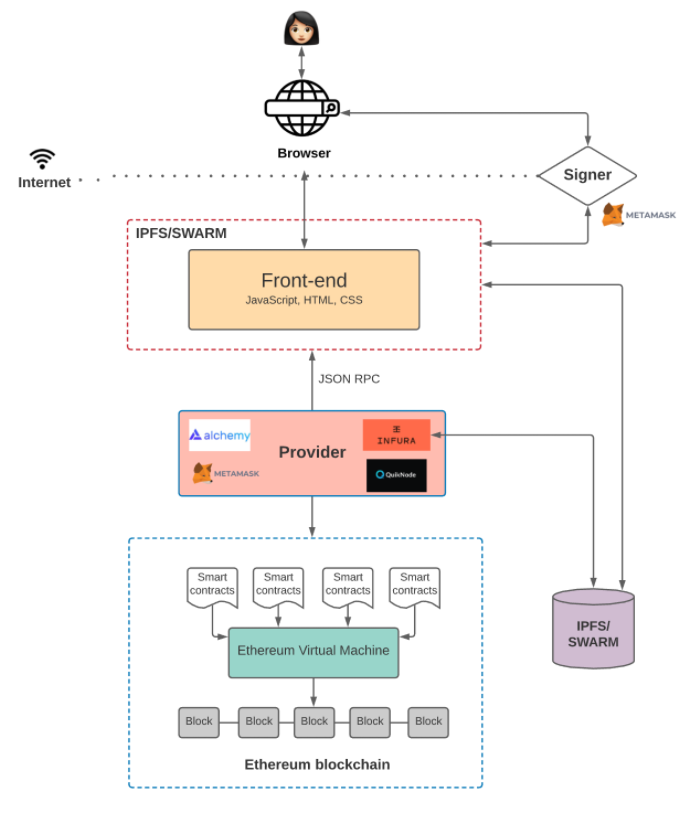


Figure d: Storing data on blockchain

## Querying the Blockain

### Smart contract events – Web3.js

* **Query and listen** for **smart contract events**.
* Can listen to specific events and specify a **callback** every time the **event** is **fired**.
* Has some limitations: if you deploy a smart contract and later realize you need an event emitted that you **didn’t originally include**: you’ll need to **redeploy** new smart contract with that event and data.
* Moreover, using callbacks to handle various **UI logic** gets very **complex** very quickly.

### The Graph

* **Off-chain indexing** solution that makes it **easier to query data** on the Ethereum blockchain.
* Define which **smart contracts** to **index**, which **events and function calls to listen** to, and how to **transform** incoming **events** into **entities** that your frontend logic can consume.
* Indexing blockchain data: **low latency**.

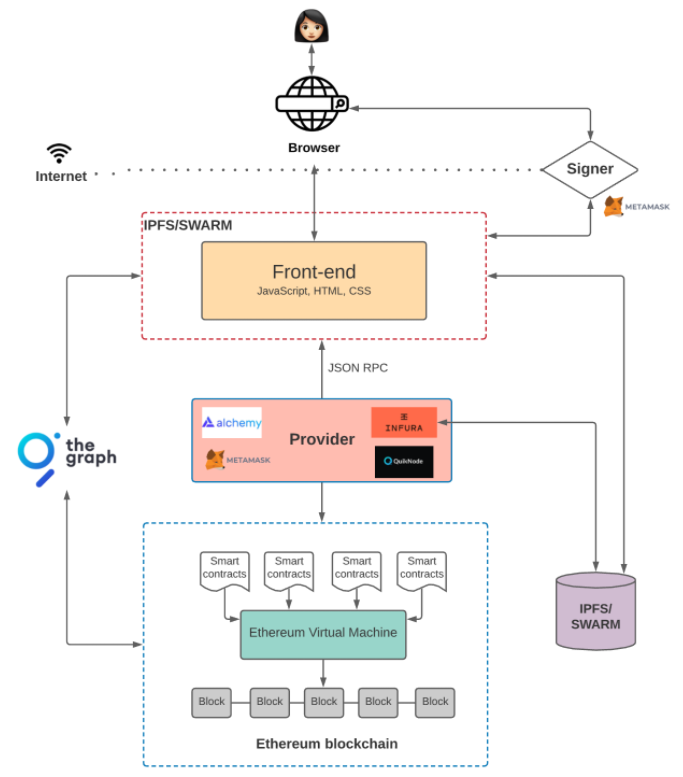


Figure e: Querying the blockchain

## Scaling Dapps

* Building a **Dapp on Ethereum**: **high gas fees** and bad UX.
* A **sidechain** is a secondary blockchain that **interfaces** with the **main chain**. Every so often, the sidechain submits an aggregation of its recent blocks back to the primary chain.
* Polygon, an L2 scaling solution: has “sidechains” that **process and execute transactions**.
* Other examples of L2 solutions are Optimistic Rollups and zkRollups. The idea here is similar: We **batch transactions off-chain** using a “**rollup**” smart contract and then **periodically commit** these transactions to the main chain.
* **L2 solutions** do **transaction execution** (i.e., the slow part) **off-chain**, with only the transaction data stored on-chain.
* **Don’t** have to **execute every** single **transaction on-chain**.
* **Transactions faster and cheaper** — and they can **still communicate** with the **main Ethereum blockchain** when necessary.

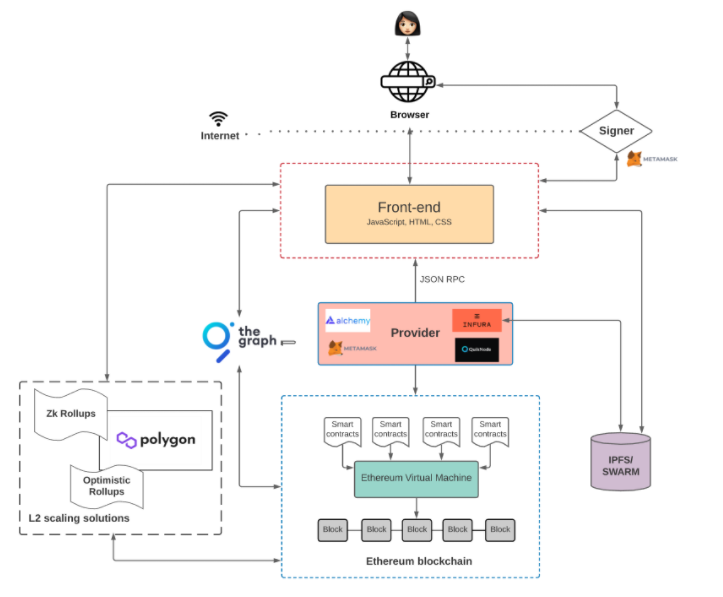


Figure f: Web 3.0 with layer 2 solutions