

DECENTRALIZED APPLICATION (DAPP) ECOSYSTEM

DECENTRALIZED APP COMPONENTS

- ❖ an **Ethereum-based network**

- mainnets (live nets)
- testnets
- personal nets

- ❖ one or more **smart contracts**

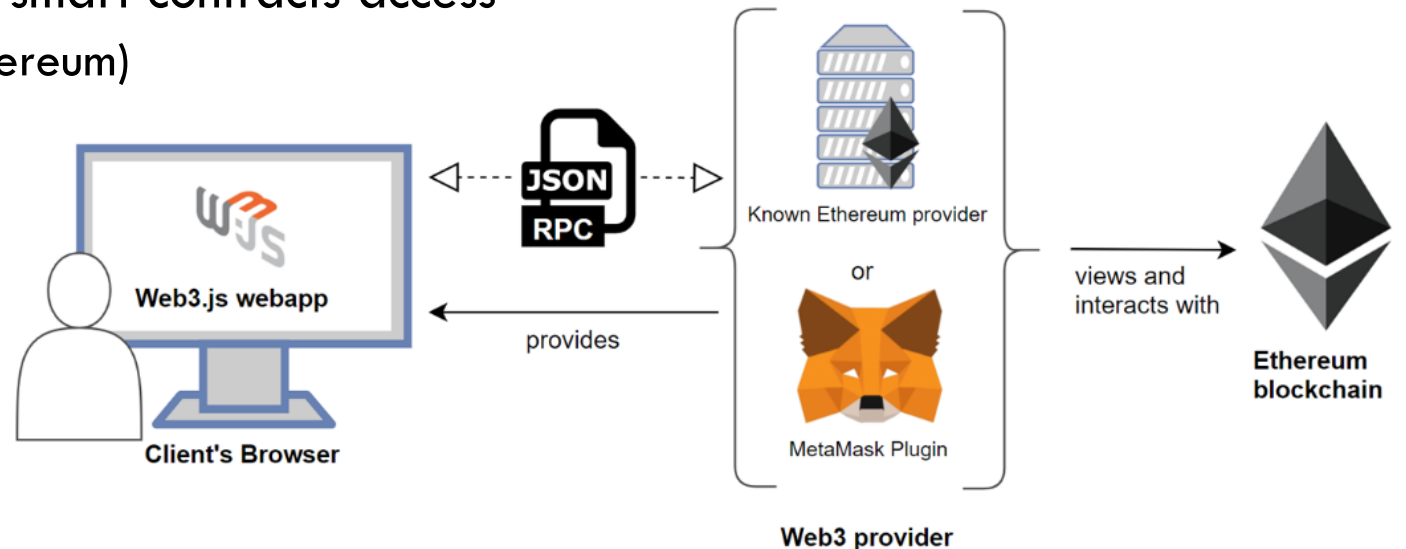
- ❖ a digital **wallet** to manage EOA accounts and sign transactions

- ❖ a **Web3 Provider** for blockchain and smart contracts access

- Ethereum clients (e.g. Geth or OpenEthereum)
- remote clients (e.g. Metamask)
- cloud-based clients (e.g. Infura)

- ❖ a **front-end application**

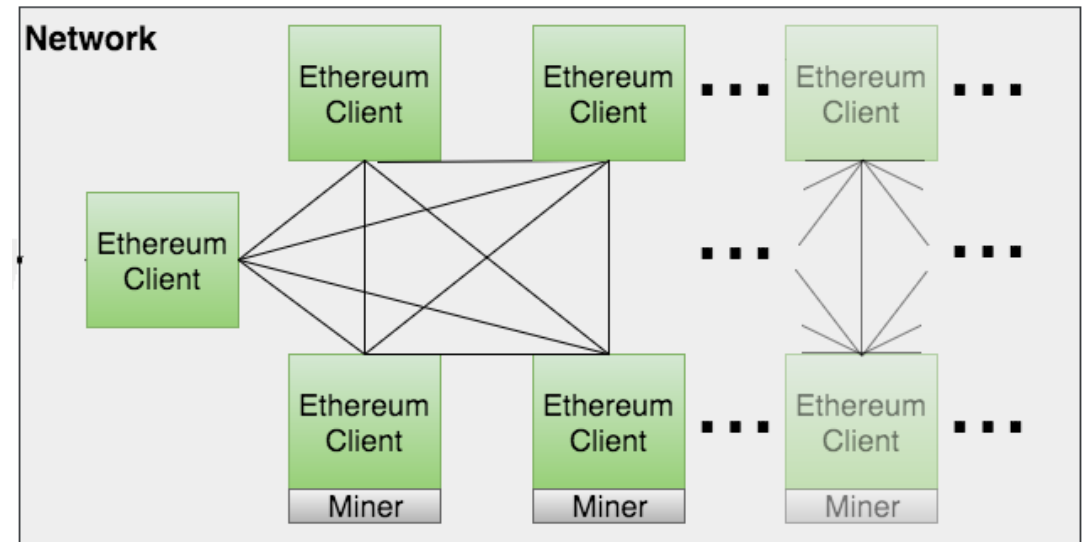
- web app
- mobile app
- back-end app



ETHEREUM NETWORKS

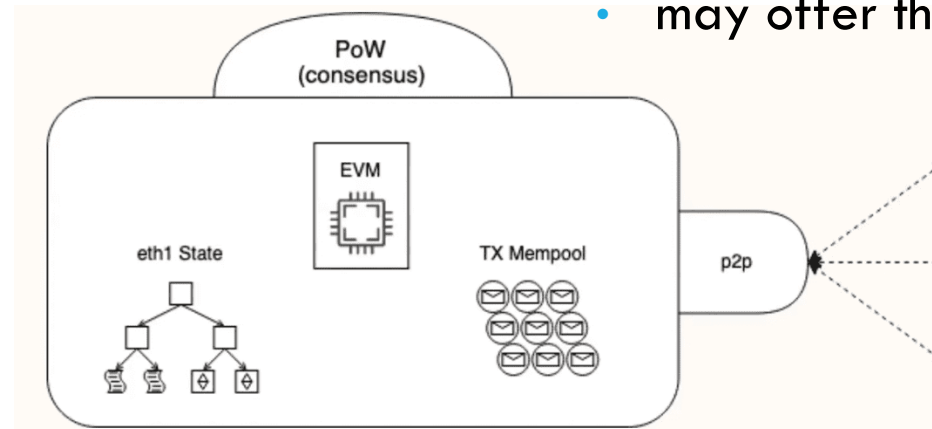
There exists a variety of **Ethereum-based** networks, either **public** or **private**, which may or may not interoperate with each other:

- **mainnets** (e.g. Ethereum, Energy Web Chain)
- **testnets** (e.g. Goerli, Sepolia, Volta)
- **personal nets** (e.g. Ganache, Remix JVM, Truffle Develop)



ETHEREUM CLIENTS

- Ethereum clients are **software applications** implementing the **open-source Ethereum specification** (yellow paper) that communicate over the P2P network with other Ethereum clients
- Ethereum clients can also provide **Web3 Provider** and **wallet functionalities** to allow blockchain and smart contracts access from Web2
- DApp developers **do not need to run a client** with **full-node** features
- DApps can access the blockchain through **remote** or **client-based clients**, which:
 - are **able to connect** to full nodes of existing main, test or private Ethereum-based networks
 - **do not store** a local copy of the blockchain
 - **do not validate** transactions and blocks
 - may **create and broadcast** transactions
 - may offer the **functionality of a wallet**



Simplified diagram of an Ethereum full-node client

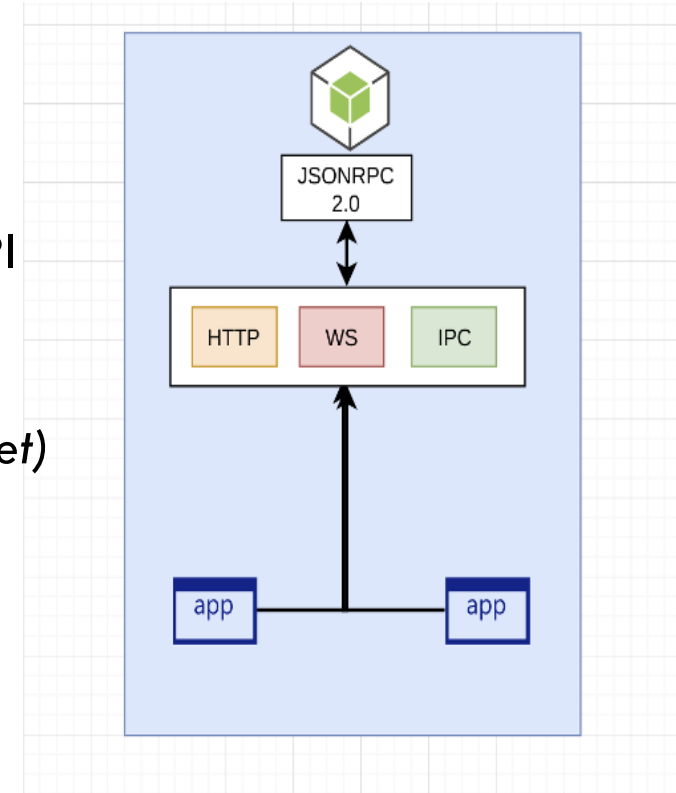
JSON-RPC INTERFACE

Ethereum clients - with Web3 Provider proxy capabilities - offer an **API** in the form of Remote Procedure Call (**RPC**) commands, which allow developers to use the **Ethereum client** as an access **gateway to the Ethereum blockchain**

The RPC commands are encoded as JavaScript Object Notation (JSON). The API is therefore referred as the **JSON-RPC API**

Usually, the **RPC interface** is offered as an **HTTP service** on **localhost** port **8545**, but it can be done through other communication protocols (*IPC, WebSocket*)

Native access to the JSON-RPC API can be made by means of a generic command-line **HTTP client** (e.g. *Curl*) by sending the **request message** to the Ethereum client and receiving the **response message** from it



// Request

```
curl -X POST -H "Content-Type" : "application/json" --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
http://localhost:8545
```

// Response

```
{"id":1, "jsonrpc":"2.0", "result":"Mist/v0.9.3/darwin/go1.4.1"}
```

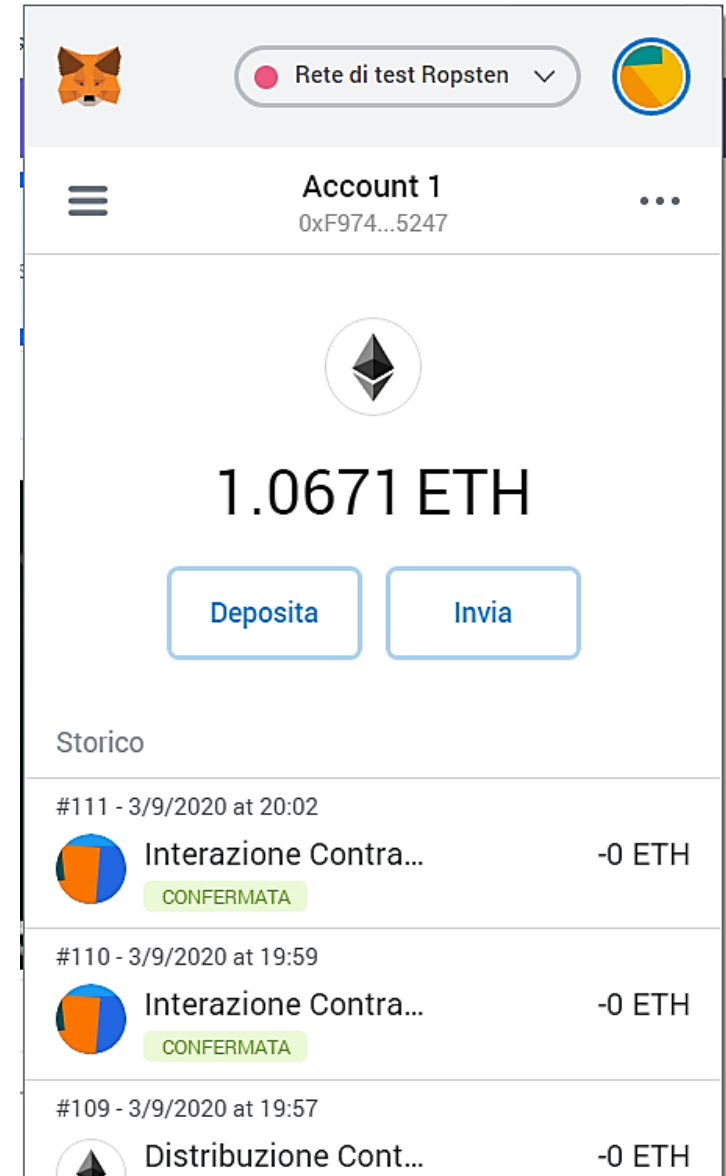
WALLETS

A **wallet** is a specialized SW functionality capable of **managing Externally Owned Accounts (EOA)**:

- store private keys
- retrieve account balances from the blockchain
- **do not hold cryptocurrencies**
- sign and inject transactions into the network

Ethereum clients of any kind may **include wallet functionalities**

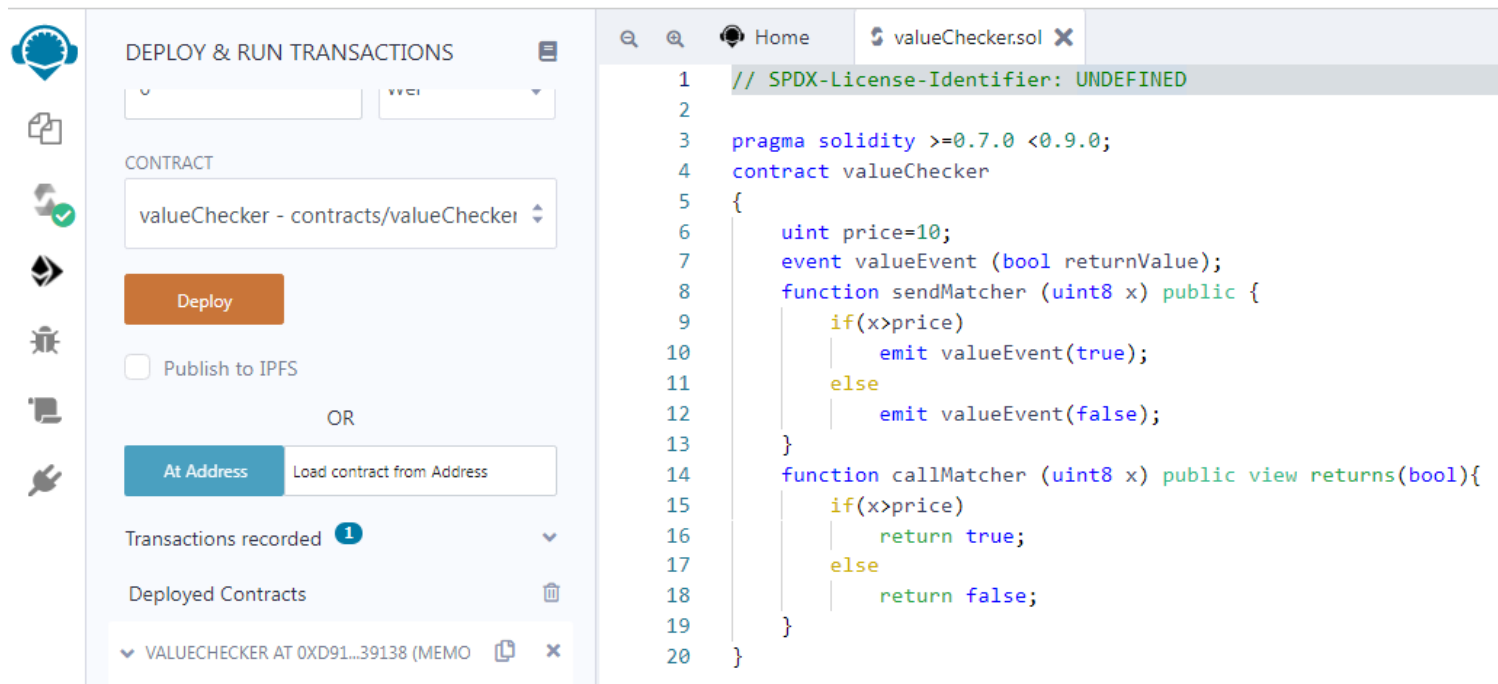
There exist **browser plug-in wallets**, which connect with the browser-based DApp UI (e.g. *Metamask*)



SMART CONTRACTS

Immutable EVM software modules:

- ✓ **written** in *Solidity* (or other high-level languages)
- ✓ **compiled** into *EVM bytecode* before being **deployed** over the blockchain
- ✓ **saved** in the *ephemeral storage area* associated with the newly created *contract's account*
- ✓ **executed** by the EVM on a *function call* to the smart contract's account address



The screenshot displays a web interface for deploying and running transactions. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel shows a dropdown menu for the contract, currently set to 'valueChecker - contracts/valueChecker'. Below this is a 'Deploy' button and a checkbox for 'Publish to IPFS'. The 'At Address' section is active, showing a text input field for the contract address and a 'Load contract from Address' button. The bottom of the panel shows 'Transactions recorded' (1) and 'Deployed Contracts' (VALUECHECKER AT 0XD91...39138 (MEMO)).

On the right, the Solidity code editor shows the following code:

```
1 // SPDX-License-Identifier: UNDEFINED
2
3 pragma solidity >=0.7.0 <0.9.0;
4 contract valueChecker
5 {
6     uint price=10;
7     event valueEvent (bool returnValue);
8     function sendMatcher (uint8 x) public {
9         if(x>price)
10            emit valueEvent(true);
11        else
12            emit valueEvent(false);
13    }
14    function callMatcher (uint8 x) public view returns(bool){
15        if(x>price)
16            return true;
17        else
18            return false;
19    }
20 }
```

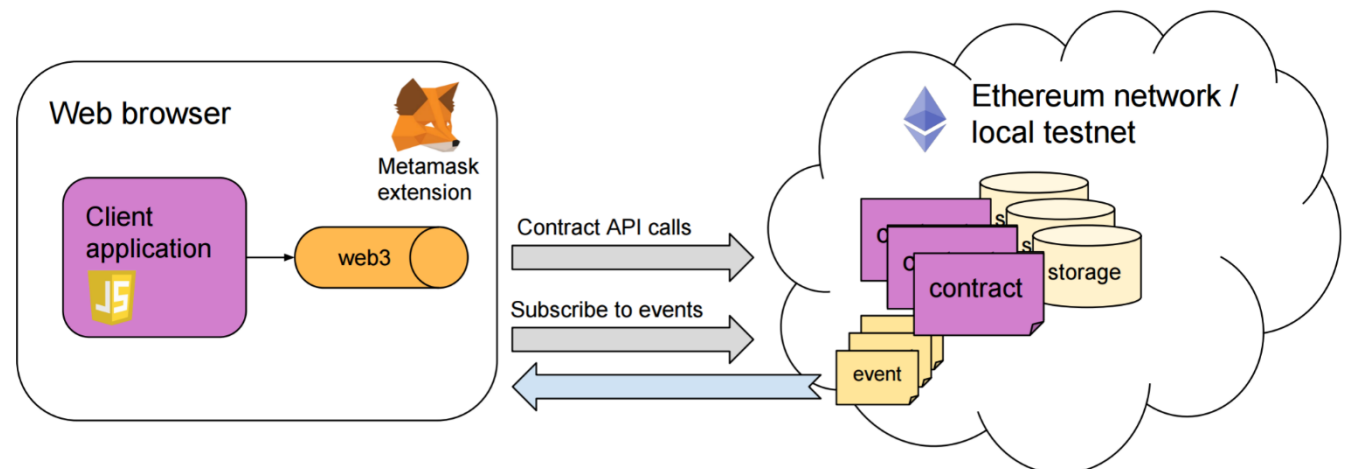
FRONT-END

The DApp's front-end programmatically **interfaces with smart contracts**:

- **interacts with a wallet software** to get transactions signed and EOA accounts managed
- **connects to any Ethereum client** (full, remote, cloud-based) which exposes the Web3 JSON-RPC API
- **reads from/writes to deployed contracts** or **reads from blockchain data structures** (i.e. log entries) preferably by means of *high-level libraries (e.g.: web3.js, ethers.js, web3j)*

A **front-end application** can be **written** in:

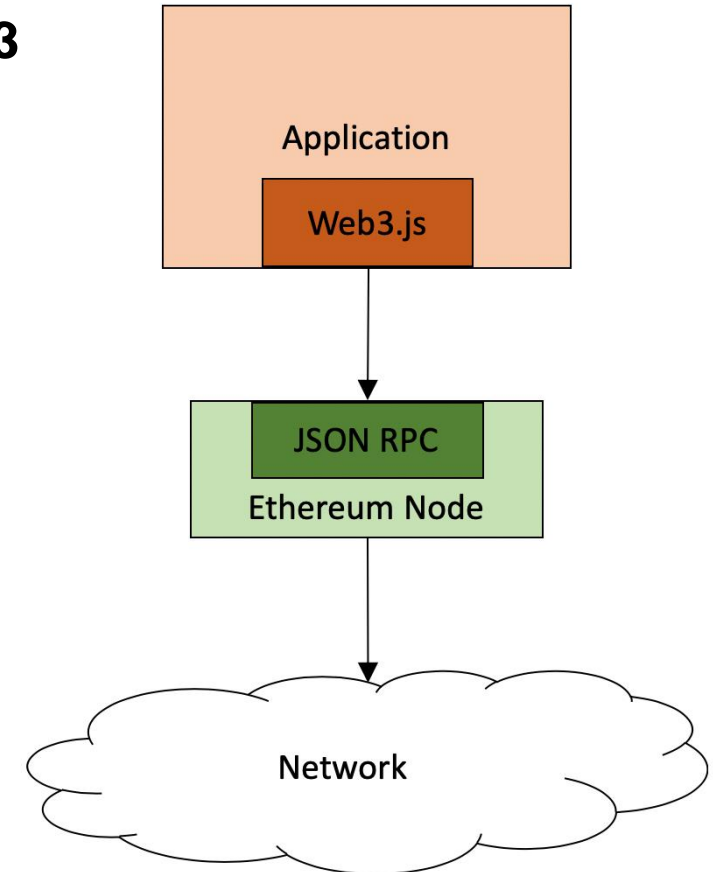
- **JavaScript/TypeScript** as a **browser application** → Apache webapp, Node.js webapp ...
- **Java** as a stand-alone application, microservice, mobile app ...
- **Kotlin** as a mobile app



TOOLS, TESTNET AND FRAMEWORKS



- **web3.js** is a convenience **JavaScript library providing a Web3 object** that can be used by the Dapp's front-end to interact with an Ethereum client
- **web3.js** works by **exposing high-level methods and objects** which hide the JSON-RPC API encoding
- **web3.js** is extensively used by IDE tools such as **Truffle** and **Remix**



(*) a more recent alternative to web3.js is **ethers.js**

METAMASK

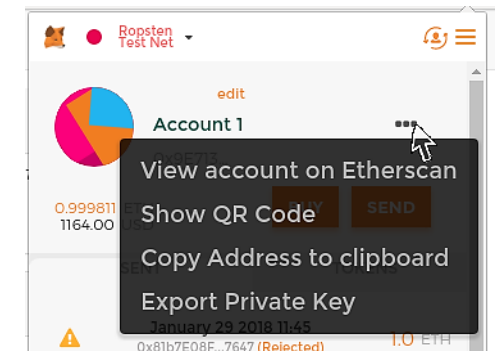
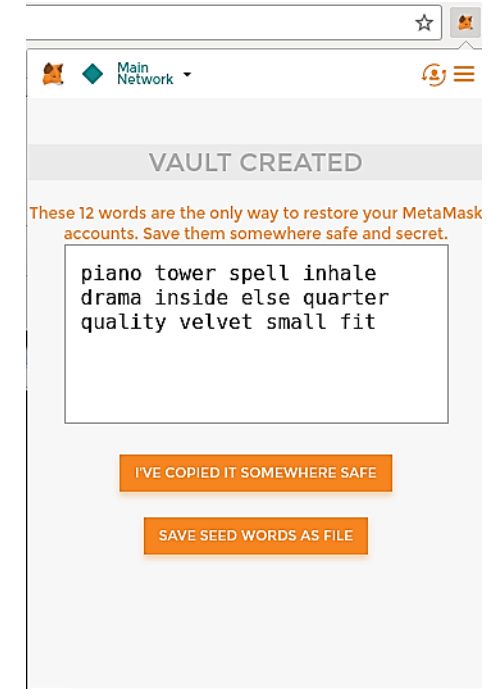


MetaMask is a **wallet software** which also provides **remote client functionalities**, since it acts as a **Web3 Provider** with respect to development tools and apps and as an **Ethereum gateway** to different network types:

- Live: Ethereum, Energy Web Chain
- Test: Goerli, Sepolia, Volta
- Localhost 8545: connects to a local node (full or test node)
- Custom RPC: connects to an Ethereum node with a compatible RPC interface

MetaMask **works as a browser extension** and, once installed, it will ask for a password and show a 12-word phrase useful to retrieve the wallet access (in case of lost password or account import from a different device)

MetaMask injects a **window.ethereum object** into the developer's browser window to let the **website interact with the MetaMask functionalities**





Remix is a **Browser IDE** which allows the following operations on Ethereum Smart Contracts :

- development
- compiling
- debugging
- testing
- deployment
- interaction

Remix can **connect to** Smart Contracts deployed on different **test** or **live environments**:

- **JavaScript VM** (test environment internal to Remix)
- **Injected Web3** (live or test environments through MetaMask)
- **Testnet Provider** (e.g., Ganache, Hardhat)
- **External Http Provider** (live or test environments through Web3 provider network endpoints, i.e. `https://<ip-address>:8545`)

A screenshot of the Remix IDE's Solidity Compiler settings panel. The panel has a vertical toolbar on the left with icons for file operations, compilation, and other functions. The main area is titled "SOLIDITY COMPILER" and contains several sections: "COMPILER" with a dropdown menu set to "0.6.12+commit.27d51765"; "LANGUAGE" with a dropdown menu set to "Solidity"; "EVM VERSION" with a dropdown menu set to "compiler default"; and "COMPILER CONFIGURATION" with three checkboxes: "Auto compile" (unchecked), "Enable optimization" (unchecked) with a dropdown set to "200", and "Hide warnings" (unchecked). At the bottom, there is a dark blue button with a refresh icon and the text "Compile 1_Storage.sol", and an orange button with the text "No Contract Compiled Yet".



- **Goerli** is the Ethereum Testnet (net ID 5) running the same consensus protocol (PoS) as the Ethereum Mainnet and provides easy funding of Test Ethers by means of network-available faucet contracts
- **Goerli** storage can be explored by means of a dedicated scanner tool, which is able to show transactions, blocks and contracts
<https://goerli.etherscan.io>

The screenshot shows the Etherscan interface for the Goerli Testnet Network. The page title is "Transaction Details" and it has tabs for "Overview", "Internal Txns", "Logs (3)", and "State". A red warning message states "[This is a Goerli Testnet transaction only]". The transaction details are as follows:

Transaction Hash:	0x0046f51053ce55831f622224f47650491e49c83abb168ea6f79ec283901e0df0
Status:	Success
Block:	8011461 (2 Block Confirmations)
Timestamp:	42 secs ago (Nov-24-2022 01:32:12 PM +UTC)
From:	0x8c6fdbca2070590d7f2ce5b2c1ed1c305d968e5d
Interacted With (To):	Contract 0xab17c7a02e67dadd170707d05fa22e7825438a60
ERC-20 Tokens Transferred:	From 0x8c6fdbca20705... To 0x000000000000... For 35 DPSPDoubleon (DOUBLO...)
Value:	0 Ether (\$0.00)
Transaction Fee:	0.00094363160698671 Ether (\$0.00)
Gas Price:	0.00000011776258667 Ether (11.776258667 Gwei)

SMART CONTRACT DEVELOPMENT AND TESTING

REMIX + METAMASK + GOERLI

<https://github.com/giorgion19/valuechecker-lab/>

This setting exploits **tools** that are **browser-based**, such as Remix and MetaMask

Remix interacts with the **Goerli** testnet via **MetaMask**, which also makes its Goerli accounts available to Remix

Through the Remix **IDE** it is possible to:

- **code, compile, debug, test** the developed smart contracts
- **deploy** them over the selected network (public, local, personal)
- **interact** with the deployed smart contracts through a built-in I/F
- **create ad-hoc front-end JS scripts**

The image shows two side-by-side screenshots. The left screenshot is from the Remix IDE, displaying the 'DEPLOY & RUN TRANSACTIONS' panel. It includes fields for 'ENVIRONMENT' (Injected Provider - MetaMask), 'ACCOUNT' (0xf28...3Dd06), 'GAS LIMIT' (3000000), 'VALUE' (0 Wei), and 'CONTRACT' (NO COMPILED CONTRACTS). The right screenshot is from the MetaMask wallet interface, showing the 'Rete di test Goerli' network, account 'Goerli #0' (0xf28...Dd06), and a balance of 1.2643 GoerliETH. It also features buttons for 'Compra', 'Invia', and 'Scambia', and a list of transactions including two 'Ricevi' entries.

DAPP SETTING AND LIVE TESTING

SETTING #1: APACHE WEBAPP + METAMASK + GOERLI

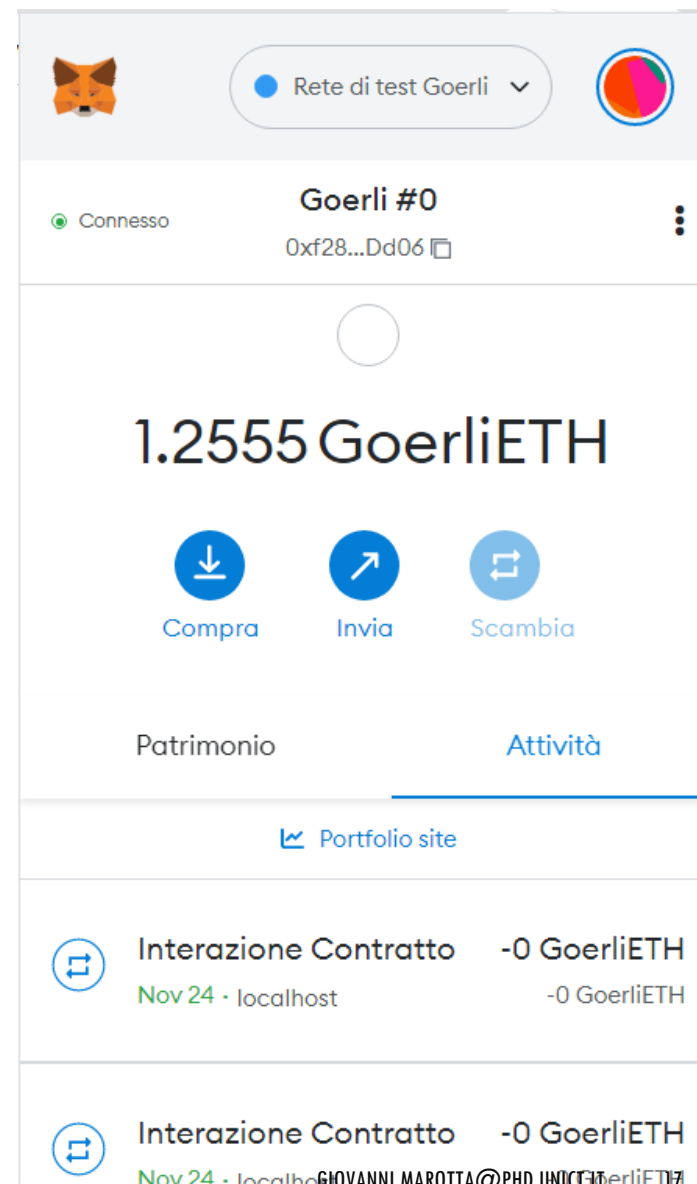
<https://github.com/giongion19/apache-metamask-goerli/>

This solution is made up of:

- i. a **HTML/JS front-end application**, deployed on an **Apache Server**
- ii. the **MetaMask** plug-in, which acts as both a **wallet software** and a **Web3 Provider** (*remote client*)
- iii. a **smart contract** deployed on the **Goerli** testnet

Requirements:

- a **Goerli EOA account** must be selected in Metamask to serve the application
- the web app connects to the **MetaMask plug-in** by means of the browser-injected **window.ethereum** object
- the **contract's address** must be known by the application beforehand (removable constraint)



SETTING #1: APACHE WEBAPP + METAMASK + GOERLI

<https://github.com/giongion19/apache-metamask-goerli/>

- The (HTML) index file includes the “**web3.js**“ library which allows the JavaScript application to **interact with the blockchain** via the MetaMask Web3 Provider
- The application script **resolves manually the dependencies** with the contract’s ABI and Ethereum address (dependencies can be resolved automatically with a dedicated DApp framework)
- The **called** smart contract’s **functions** can provide the **result value** through:
 - the **return value** of the called function (.call() web3 method in the app)
 - the **log** contained in the tx receipt generated by the emitted **event** (.send() web3 method in the app)

The screenshot shows a web browser at the URL localhost/apache-metamask-goerli/. The page title is 'Smart Contract: valueChecker'. It displays the web3.js library version as 1.3.4 and the deployed contract account as 0xf34dce9a680846c1f598e796e25c82b5794d6843. There is a blue button labeled 'Connect to MetaMask'. Below that, it shows an externally owned account: 0xf28f585656f80799d4f1727b8c585701bef3dd06. There are two input fields, both containing the number '2'. The first input field is followed by an orange button labeled 'Send transaction to contract'. Below this, it says 'Event's log generated by web3.js method .send(): false'. The second input field is followed by a green button labeled 'Call contract'. Below this, it says 'Function value returned by web3 method .call(): false'.