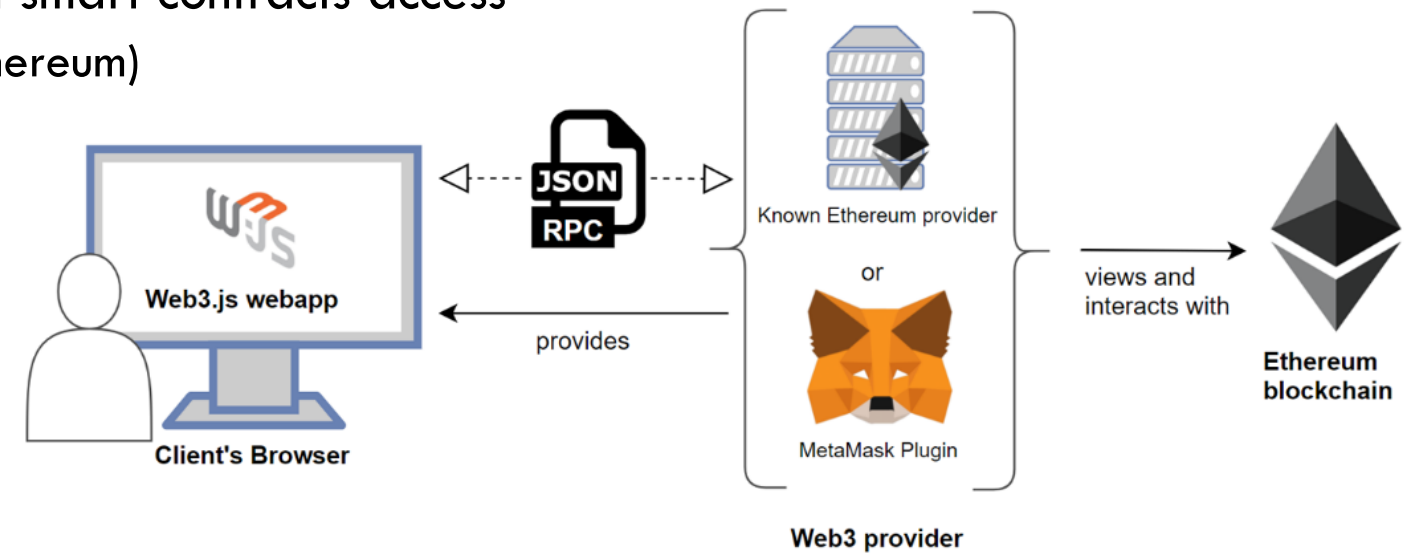
DECENTRALIZED APPLICATION (DAPP) ECOSYSTEM

DECENTRALIZED APP COMPONENTS

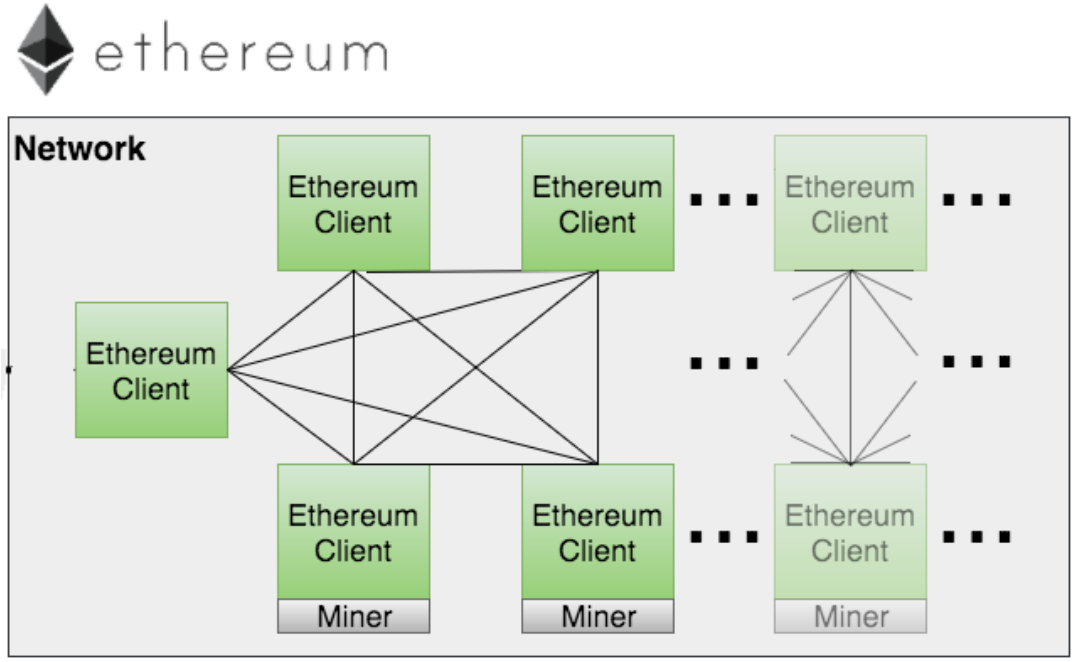
- an **Ethereum-based network**
  - mainnets (live nets)
  - testnets
  - personal nets
- one or more **smart contracts**
- a digital **wallet** to manage EOA accounts and sign transactions
- a **Web3 Provider** for blockchain and smart contracts access
  - Ethereum clients (e.g. Geth or OpenEthereum)
  - 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:

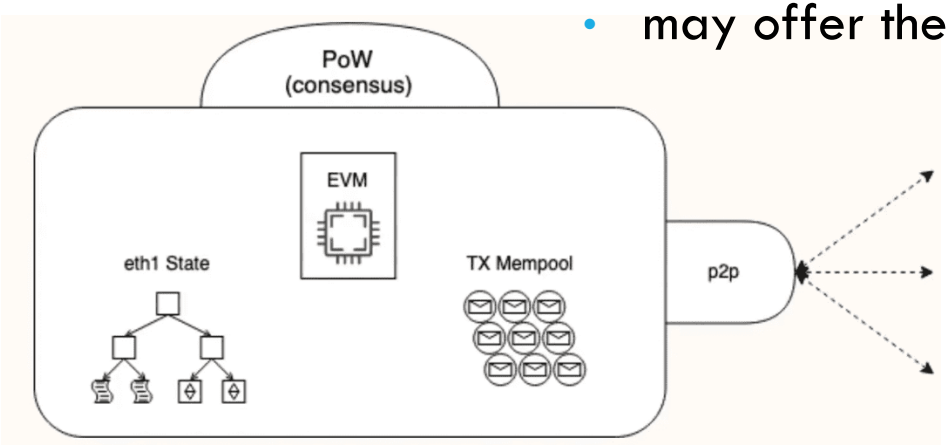
- **mainnets** (e.g. Ethereum, Energy Web Chain)
- **testnets** (e.g. Goërli, Sepolia, Volta)
- **personal nets** (e.g. Ganache, Remix JVM, Truffle Develop)



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

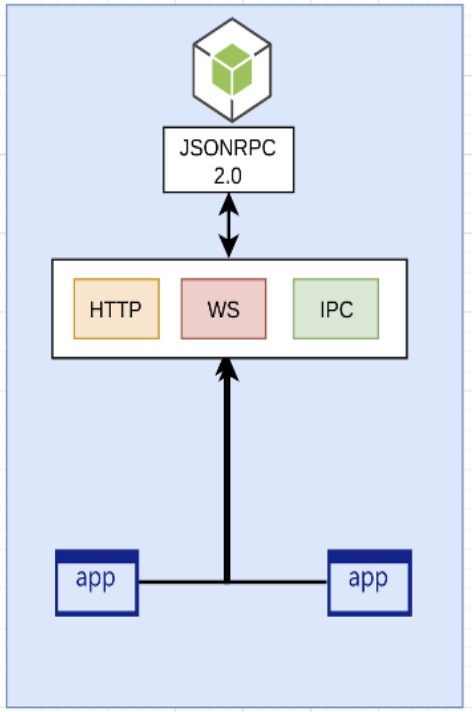
JSON-RPC INTERFACE

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**"}

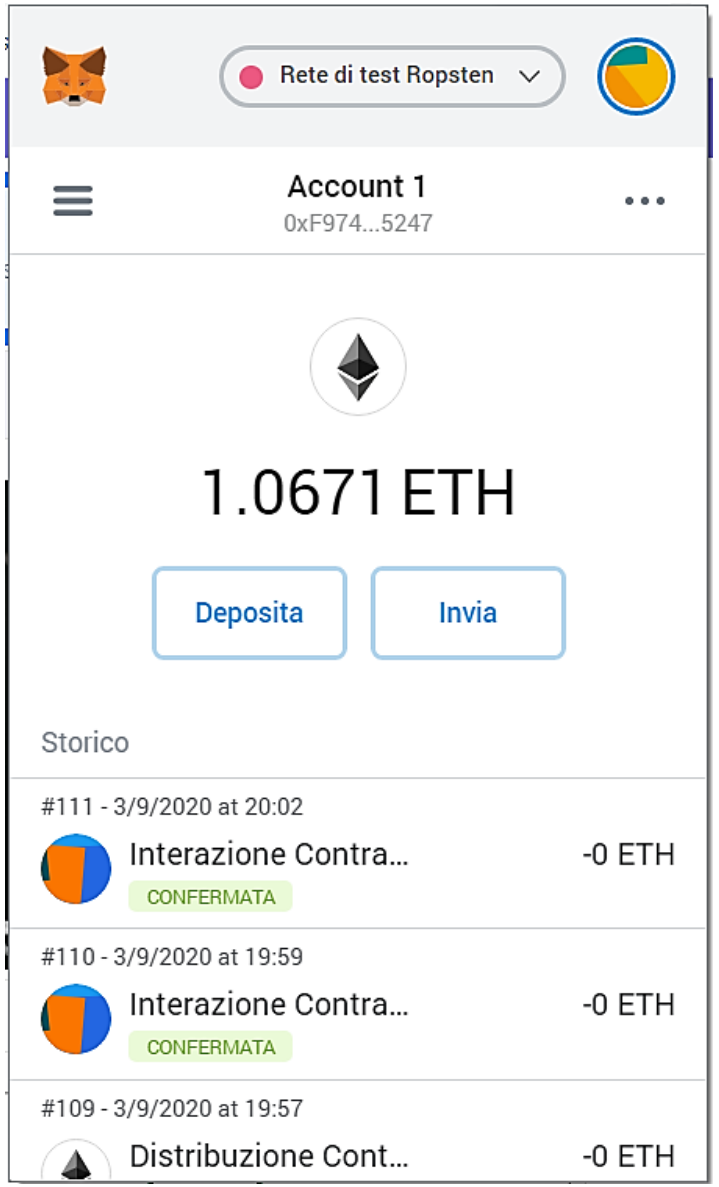
WALLETS

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**

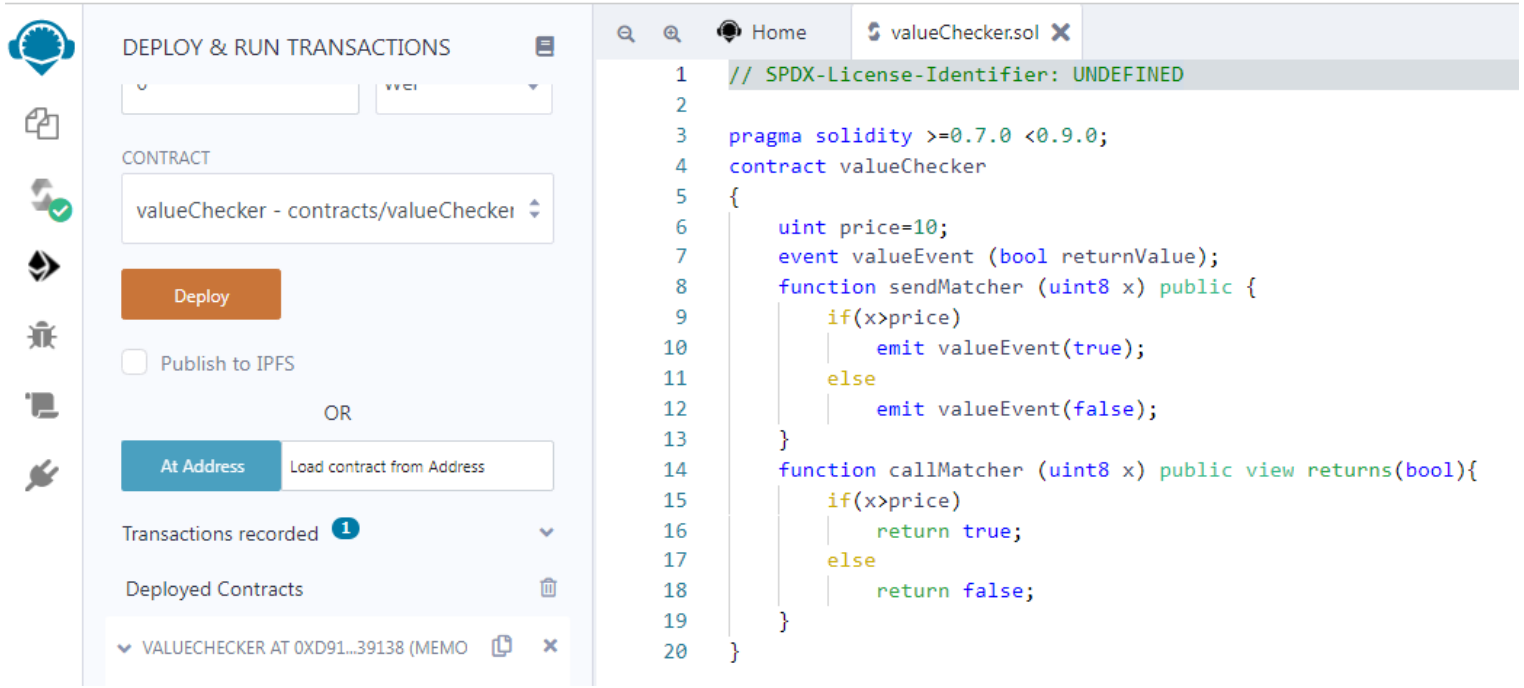
There exist **browser plug-in wallets**, which connect with the browser-based DApp UI *(e.g. Metamask)*



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



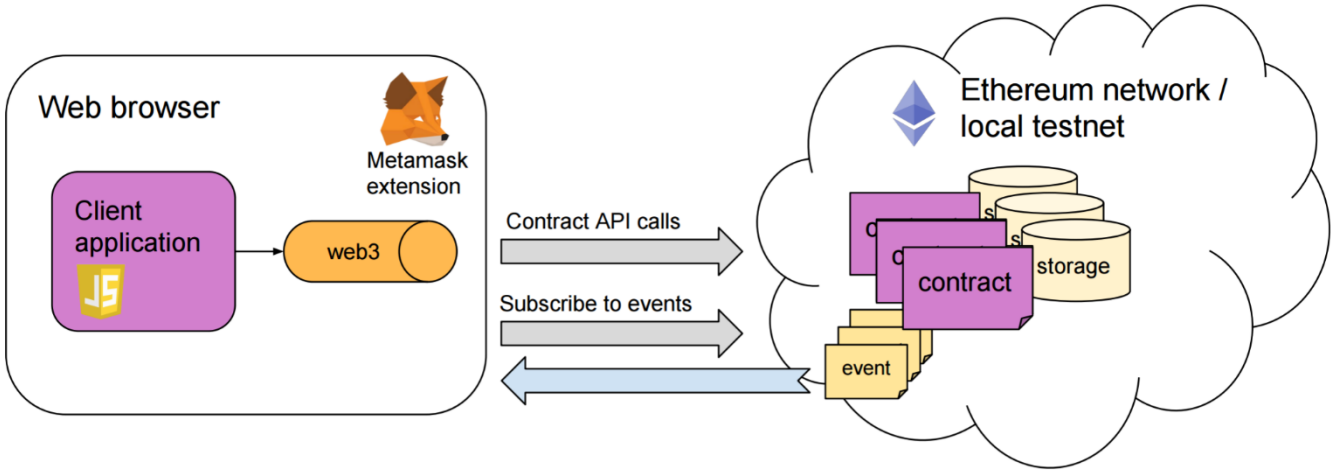
FRONT-END

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:

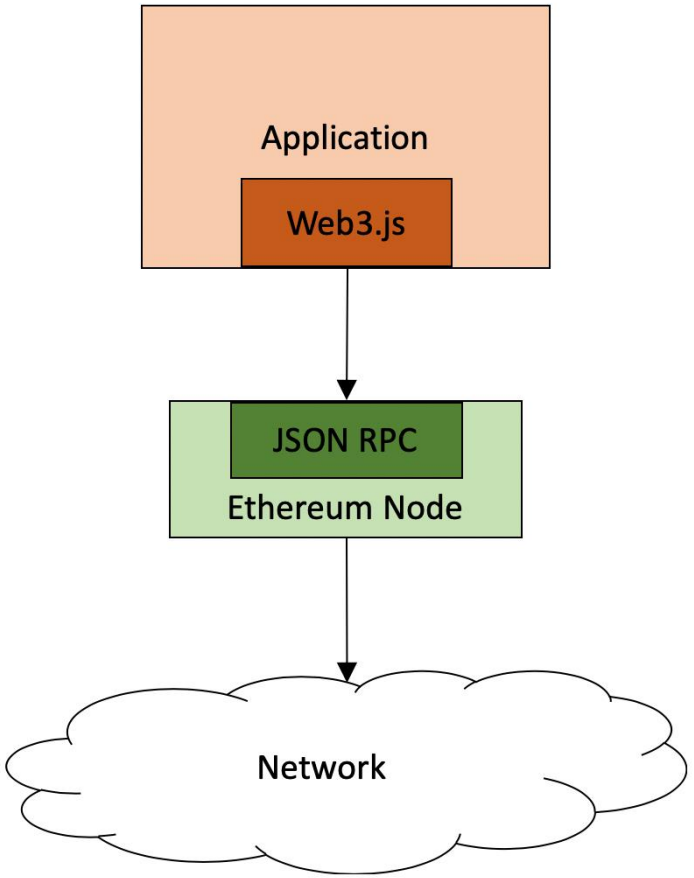
- **JavaScript/TypeScript** as a **browser application** → Apache webapp, Node.Js webapp …
- **Java** as a stand-alone application, microservice, mobile app …
- **Kotlin** as a mobile app



TOOLS, TESTNET AND FRAMEWORKS

WEB3.JS

- ***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***

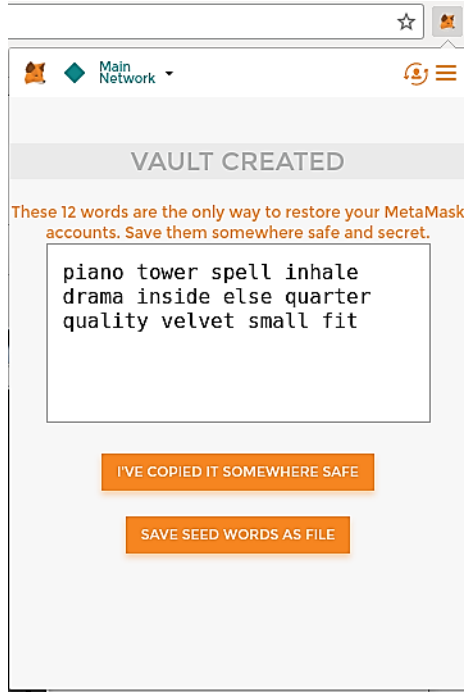
METAMASK

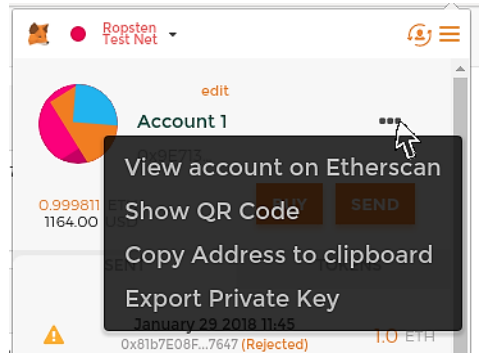
MetaMask is a **wallet software** which also provides **remote client functionalities,** since it acts as a **Web3 Provider** with respect to development tools and apps and as an **Ethereum gateway** to different network types:

- Live: Ethereum, Energy Web Chain
- Test: Goërli, 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





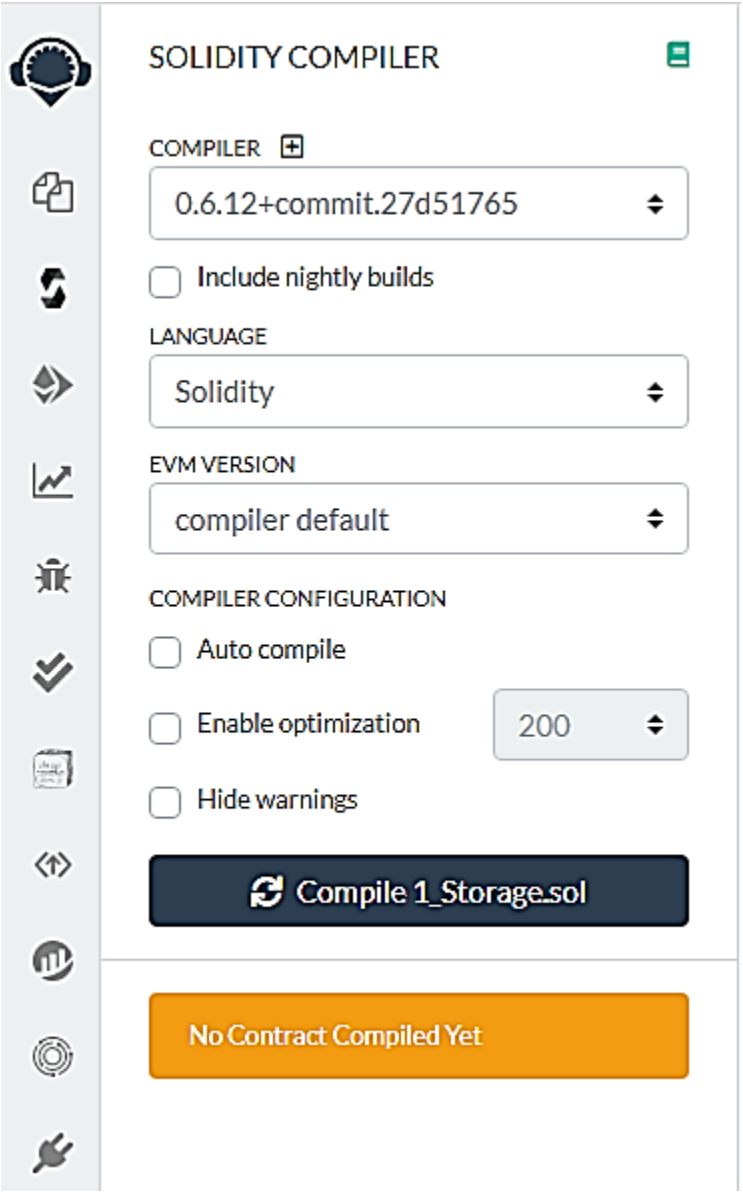
REMIX

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

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

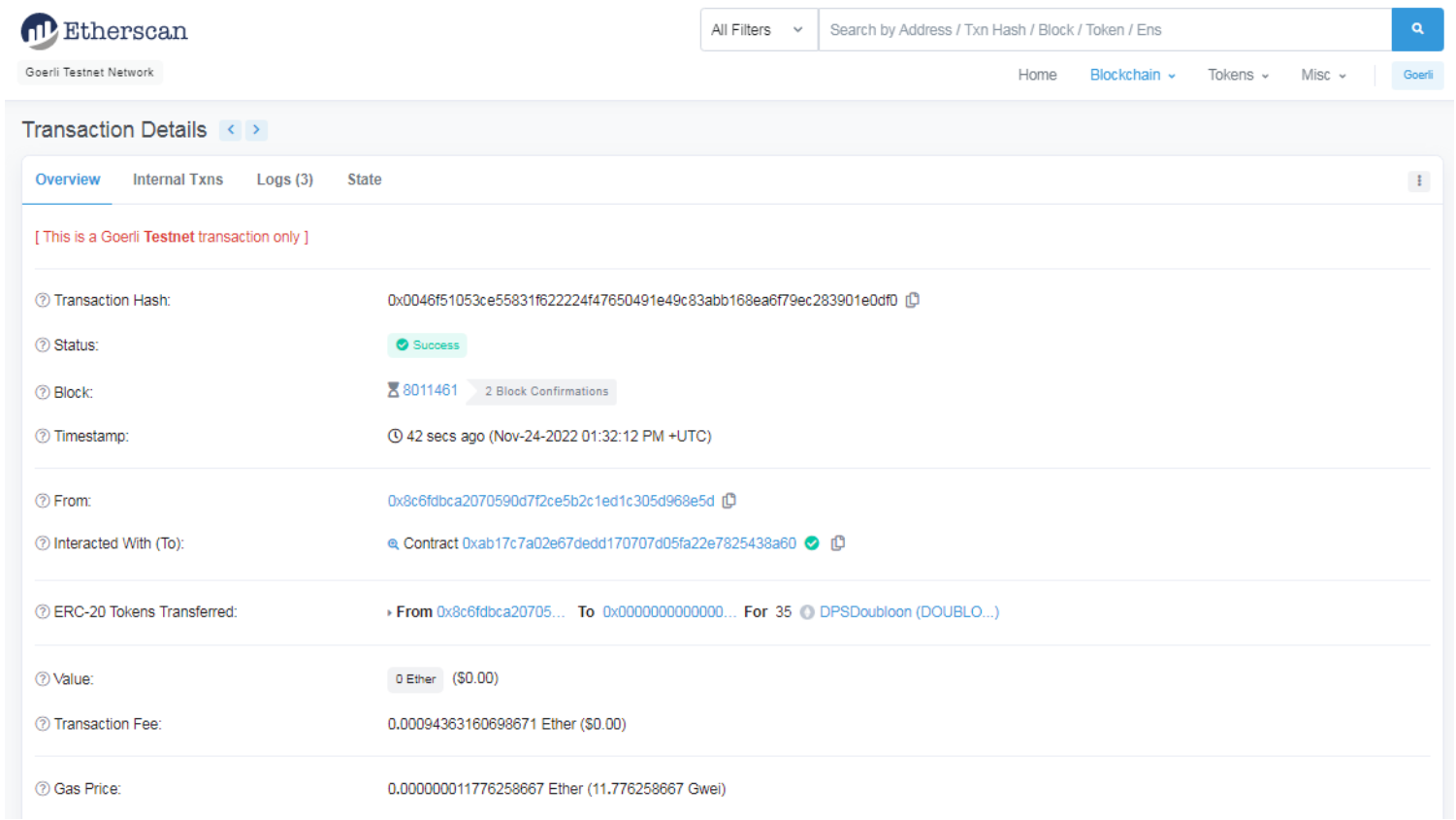
Remix can **connect to** Smart Contracts deployed on different **test** or **live environments:**

- ***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)*



GOERLI

- **Goerli** is the Ethereum Testnet (net ID 5) running the same consensum 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



SMART CONTRACT DEVELOPMENT AND TESTING

Remix + MetaMask + Goerli

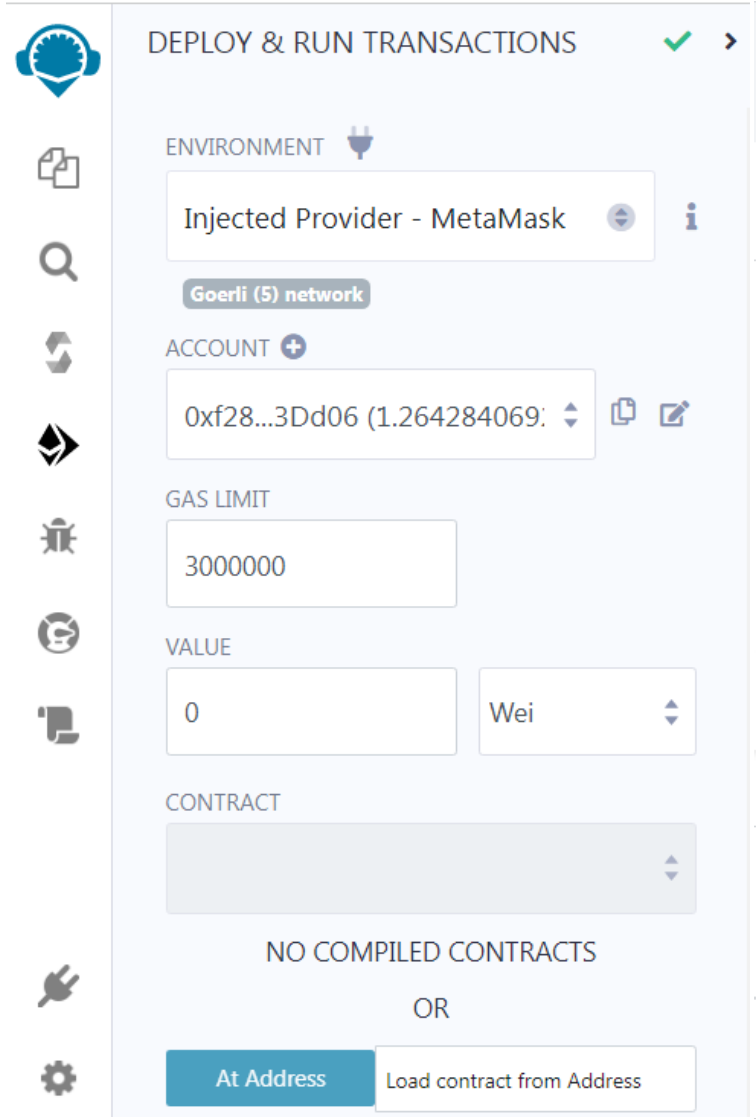
https://github.com/giongion19/valuechecker-lab/

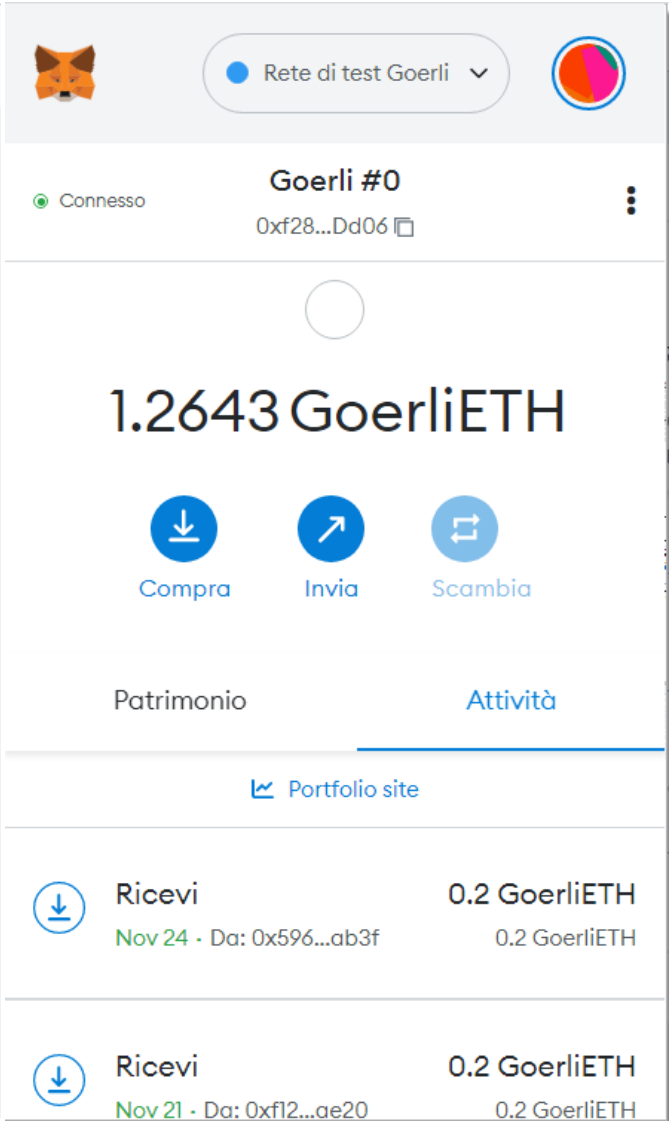
This setting exploits **tools** that are **browser-based**, such as Remix and MetaMask

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





DAPP SETTING AND LIVE TESTING

SETTING #1: APACHE WEBAPP + METAMASK + GOERLI

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

This solution is made up of:

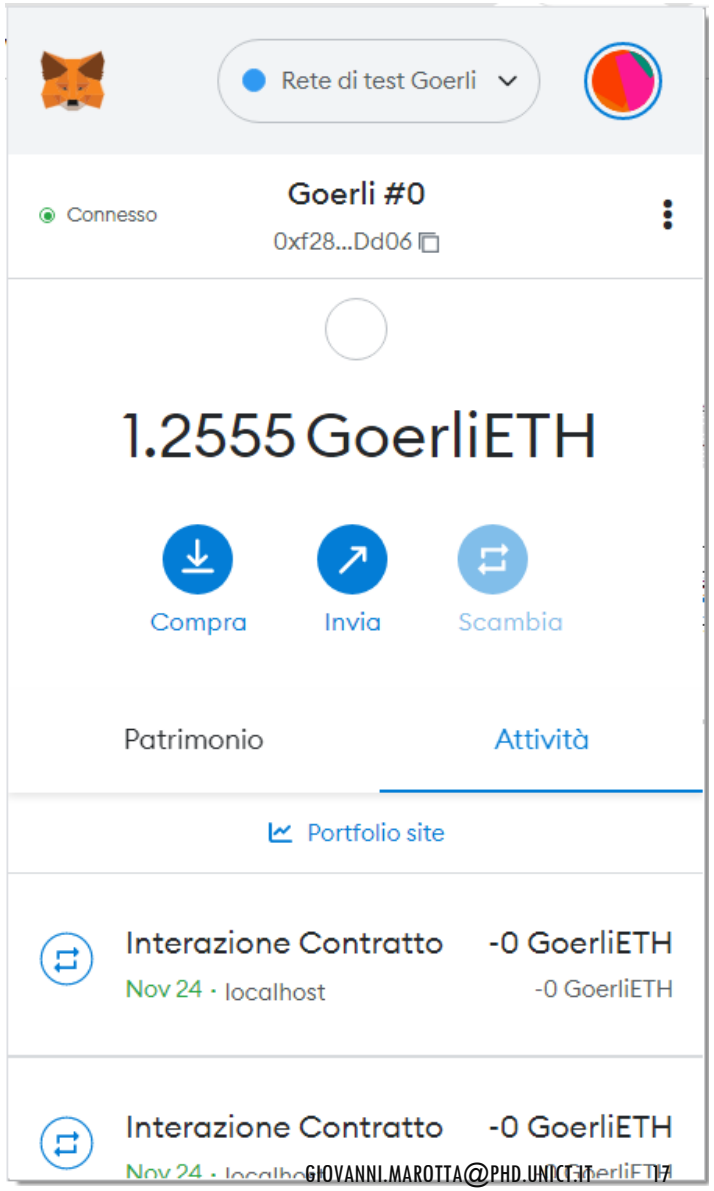
i. a HTML/JS **front-end application**, deployed on an **Apache Server**

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)