Lesson 2

Validator selection and consensus

Proof-of-Staked Authority (PoSA) consensus ("Parlia")

This is a combination of Proof of Authority and Proof of Stake

1. Blocks are produced by a limited set of validators;
2. Validators take turns to produce blocks in a PoA manner, similar to Ethereum's Clique consensus design
3. Validator set are elected in and out based on staking-based governance;

A fast finality mechanism similar to that used in ethereum was introduced in BEP 126.

Finalising a block

1. A block is proposed by a validator and propagated to other validators.
2. Validators use their BLS private key to sign for the block as a vote message.
3. Gather the votes from validators into a pool.
4. Aggregate the BLS signature if its direct parent block has gotten enough votes when proposing a new block.
5. Set the aggregated vote attestation into the extra field of the new block's header.
6. Validators and full nodes who received the new block with the direct parent block's attestation can justify the direct parent block.
7. If there are two continuous blocks have been justified, then the previous one is finalized.

BLS stands for Boneh–Lynn–Shacham Signatures [See](#)

BlockExplorers

Mainnet

- BscScan - https://bscscan.com/
- Bitquery - https://explorer.bitquery.io/bsc
- BSCTrace - https://bsctrace.com/

Testnet

- BscScan - https://testnet.bscscan.com/
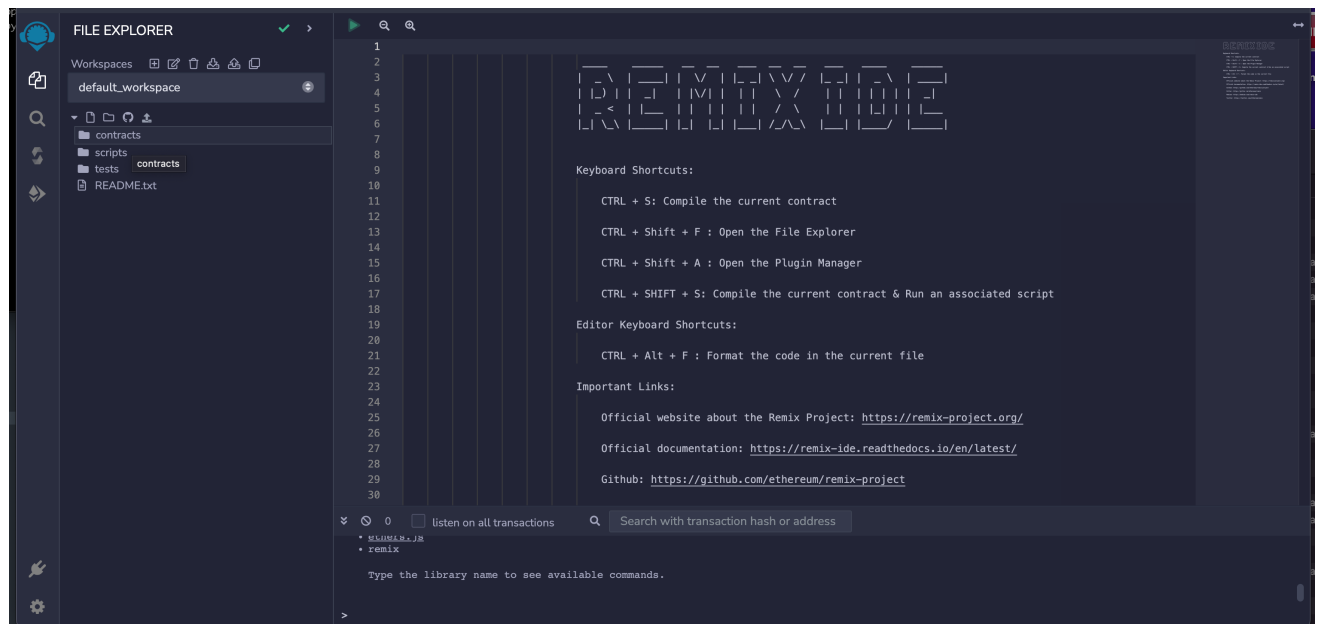- Bitquery - https://explorer.bitquery.io/bsc_testnet

Faucets

Test net Faucet

Introduction to Remix

Remix is an IDE for Solidity development.
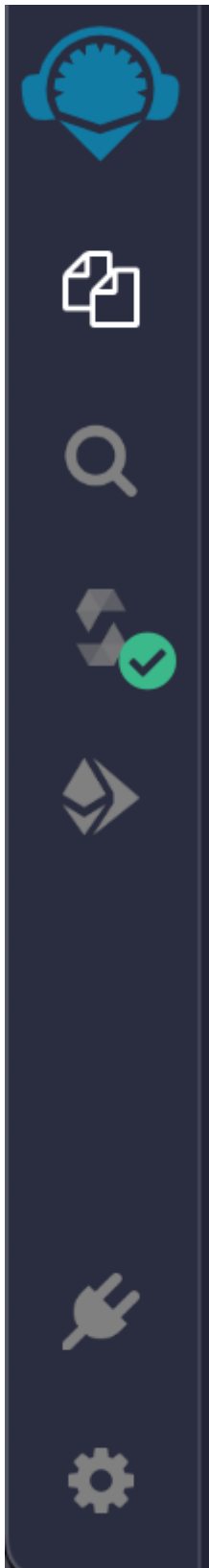It can run in the browser, or on the desktop.

Browser : https://remix.ethereum.org/
Desktop : `npm install –g @remix–project/remixd`
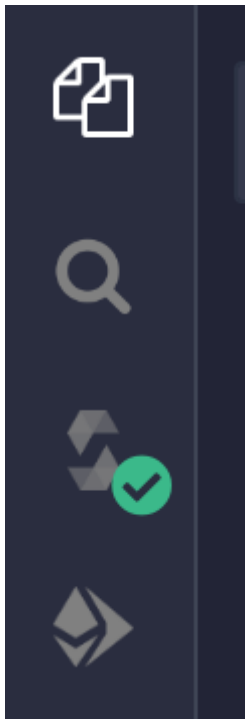
Remix documentation

Plugins

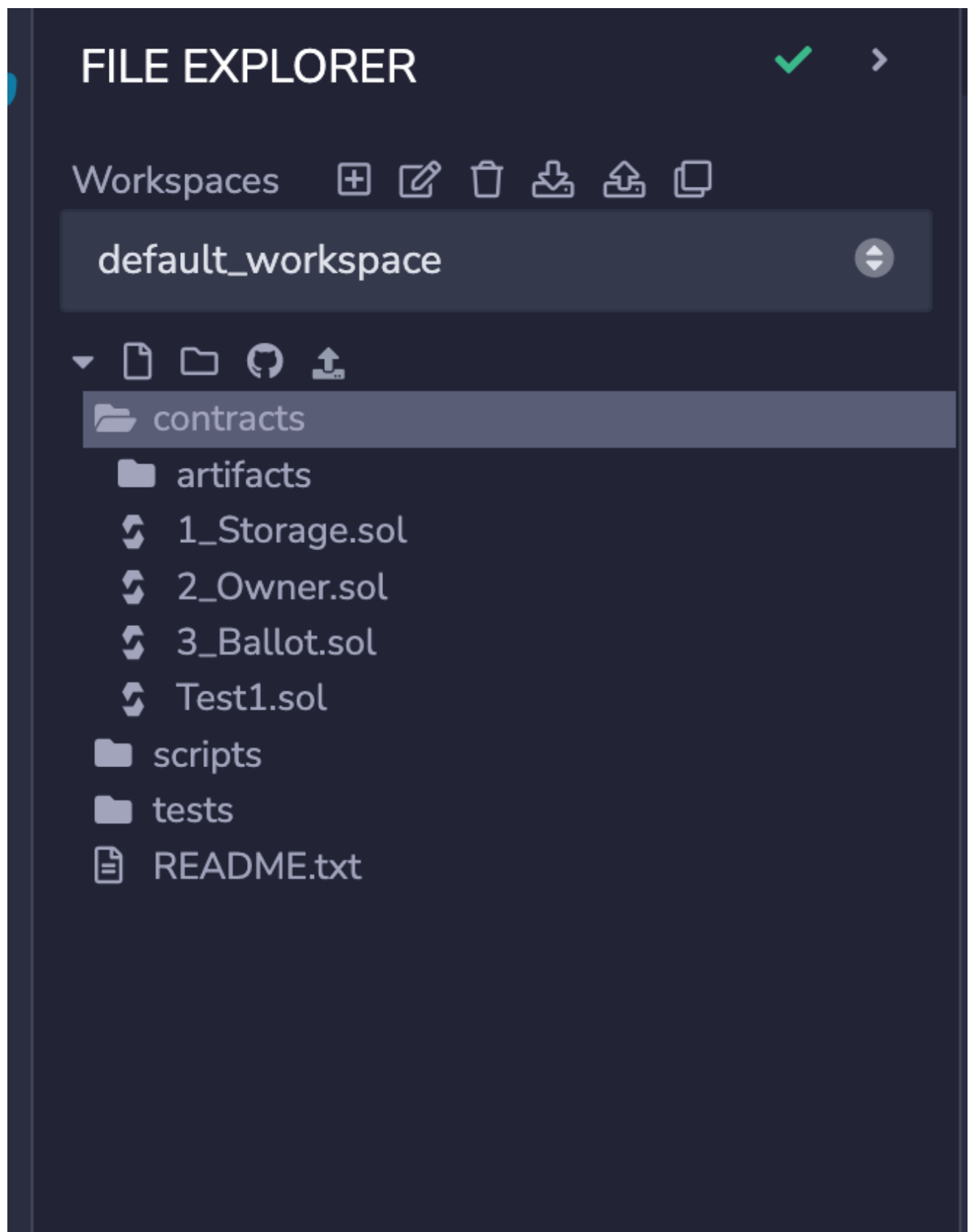The panel on the left allows us to add plugins

File Explorer

If we select the file icon in the plugin panel

we will see the file explorer in the middle panel

The middle panel is the file explorer, here we can navigate between the contracts we are working on.

FILE EXPLORER ✔ ›

Workspaces ⊞ ✎ 🗑 ⬇ ⬆ ⧉

default_workspace ⇕

▾ 📄 📁 🐙 ⬆

📂 contracts
📁 artifacts
↻ 1_Storage.sol
↻ 2_Owner.sol
↻ 3_Ballot.sol
↻ Test1.sol
📁 scripts
📁 tests
📄 README.txt

Editor Panel

Here we edit our smart contracts

```
1    // SPDX-License-Identifier: GPL-3.0
2
3    pragma solidity >=0.7.0 <0.9.0;
4
5    /**
6     * @title Storage
7     * @dev Store & retrieve value in a variable
8     * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
9     */
10   contract Storage {
11
12       uint256 number;
13
14       /**
15        * @dev Store value in variable
16        * @param num value to store
17        */
18       function store(uint256 num) public {
19           number = num;
20       }
21
22       /**
23        * @dev Return value
24        * @return value of 'number'
25        */
26       function retrieve() public view returns (uint256){
27           return number;
28       }
29   }
```

Ouptut panel

This shows the activity on our local blockchain and the results of our transactions


```
Welcome to Remix 0.27.0

Your files are stored in indexedDB, 3.66 MB / 2 GB used

You can use this terminal to:
• Check transactions details and start debugging.
• Execute JavaScript scripts:
    - Input a script directly in the command line interface
    - Select a Javascript file in the file explorer and then run \`remix.execute()\` or \`remix.exeCurrent()\`  in the command line interface
    - Right click on a JavaScript file in the file explorer and then click \`Run\`

The following libraries are accessible:
• web3 version 1.5.2
• ethers.js
• remix

Type the library name to see available commands.

>
```

Compiling the contract

Choosing the compile icon in the left hand panel will display the compiler panel

COMPILER + 🗋

0.8.7+commit.e28d00a7 ⬍

☐ Include nightly builds
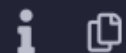
☑ Auto compile

☐ Hide warnings

Advanced Configurations ❯

🔄 Compile 1_Storage.sol

Compile and Run script   ⓘ ⎙
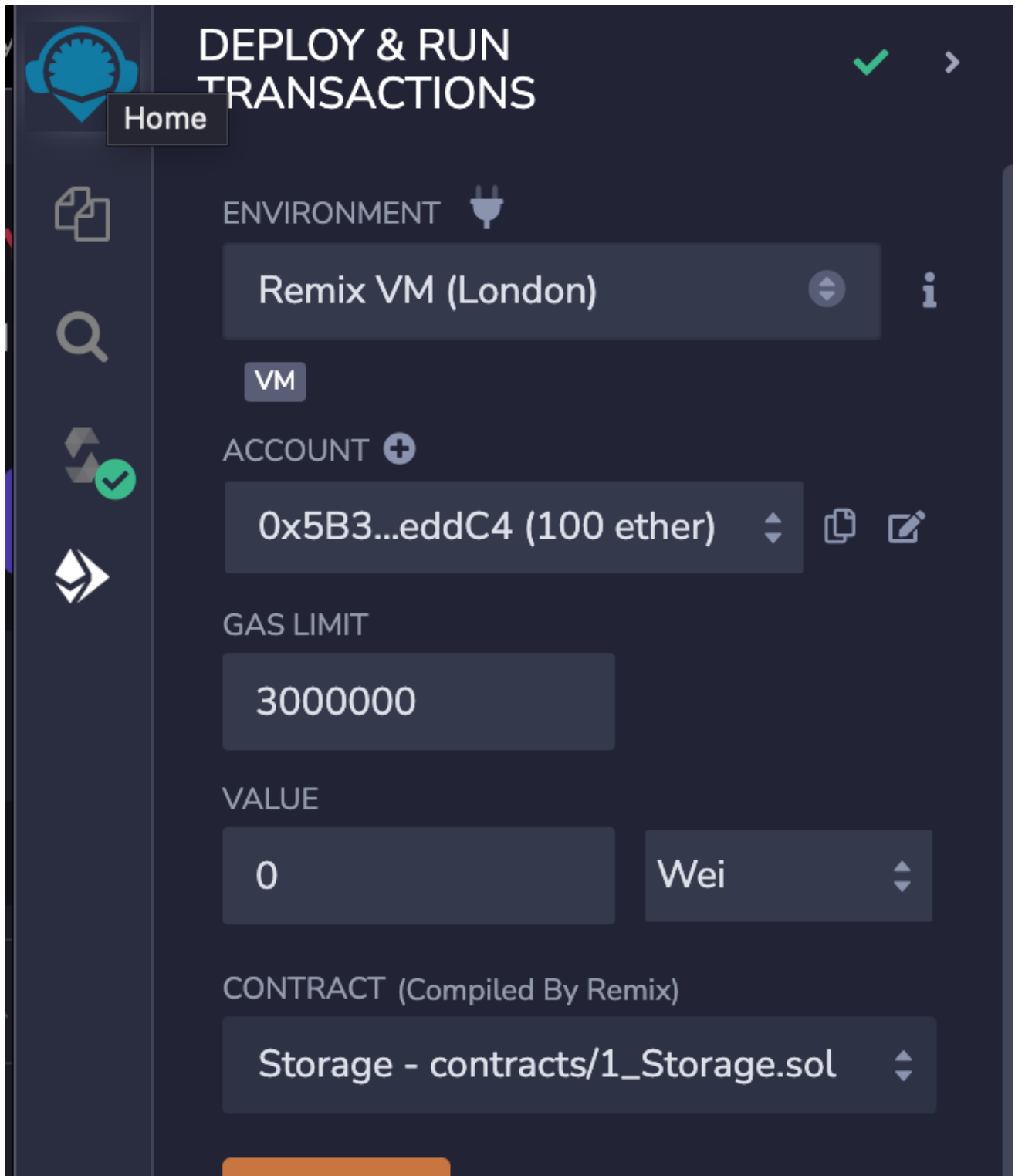
CONTRACT

Storage (1_Storage.sol) ⬍

Publish on Ipfs IPFS

Publish on Swarm ⬤

Compilation Details

Deploying a contract

Chosing the deploy and run transactions icon gives us a panel that allows us to deploy our compiled transaction to a network and run transactions.
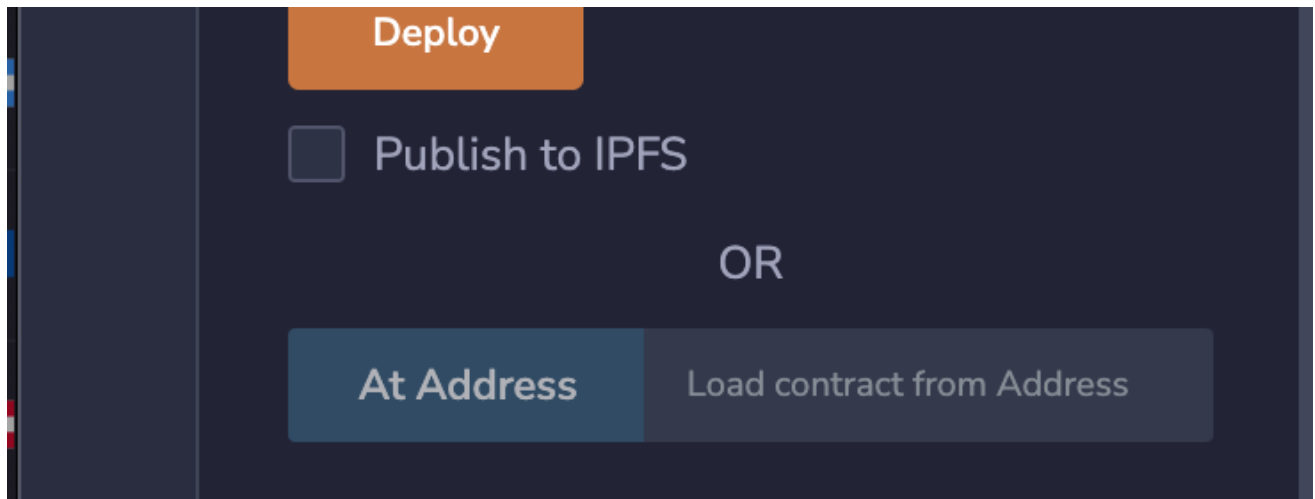
Choosing the network / environment
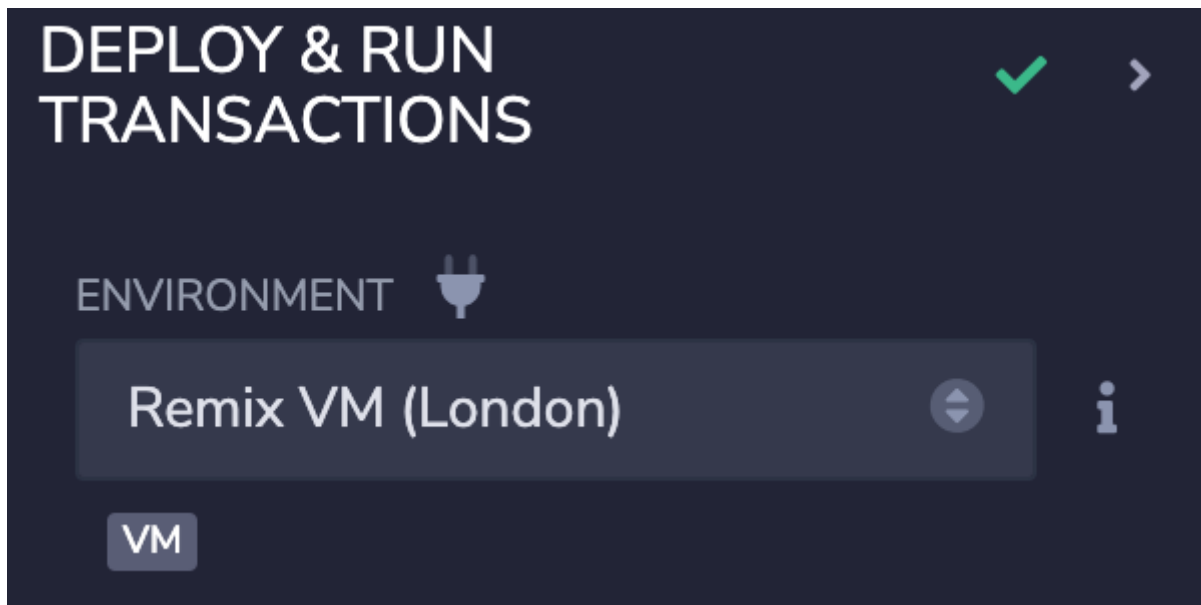
This provides a local in browser blockchain, this is the simplest environment when initially developing contracts.

We deploy the contract with the deploy button, you may need to find the correct contract in the drop down list.
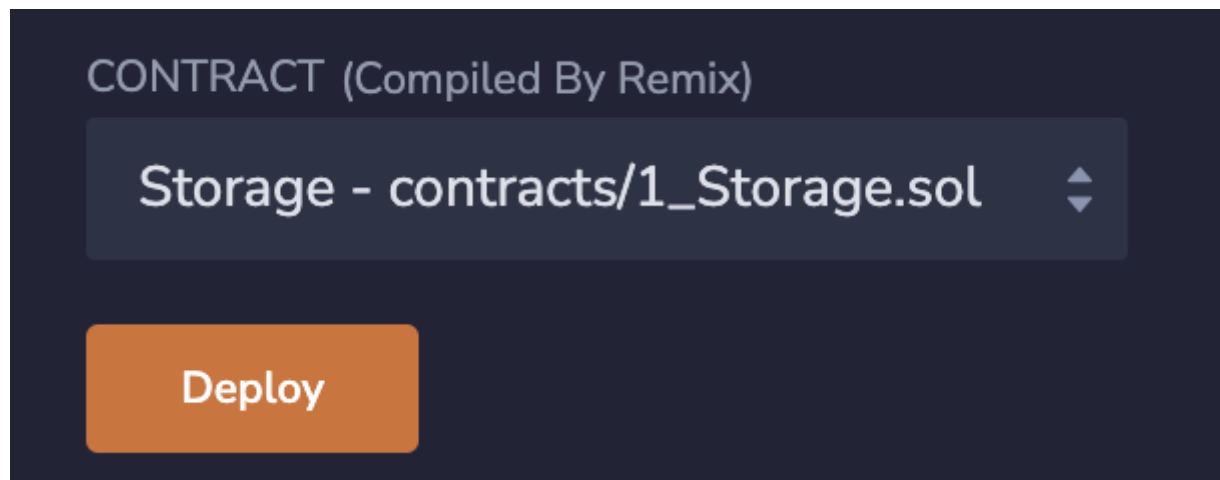
If the contract deploys correctly, you will see the results in the output window.
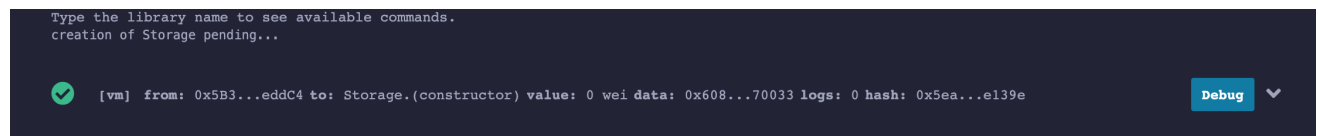
Once it has deployed, you will see details of the contract in the deploy panel

You can interact with your contracts from here by sending transactions
Note that the contract has an address on the blockchain similar to a wallet address

Deployed Contracts 🗑

STORAGE AT 0XD91...39138 (MEM ⧉ ✕

Balance: 0 ETH

store | uint256 num | ⌄

retrieve

Low level interactions ⓘ

CALLDATA

| | Transact |

Solidity part 1

Introduction

How to define the Solidity compiler version ?

```solidity
// Any compiler version from the 0.8 release (= 0.8.x)
pragma solidity ^0.8.0;

// Greater than version 0.7.6, less than version 0.8.4
pragma solidity >0.7.6 <0.8.4;
```

How to define a contract ?

Use the keyword contract followed by your contract name.

```solidity
contract Score {

    // You will start writing your code here =)

}
```

How to write a variable in Solidity ?

A contract would need some state. We are going to declare a new variable that will hold our score.

```solidity
contract Score {

    uint score = 5;

}
```

Important !

Solidity is a statically typed language. So you always need to **declare the variable type** (here `uint`) before the variable name.

Do not forget to end your declaration statements with a semicolon `;`

`uint` defines an *unsigned integer* of 256 bits by default.
You can also specify the number of bits, by range of 8 bits. Here are some examples below:

Type	Number range
uint8	0 to 255
uint16	0 to 65,535
uint32	0 to 4,294,967,295
uint64	0 to 18,446,744,073,709,551,615
uint128	0 to 2^128
uint256	0 to 2^256

Getters and Setters

We need a way to write and retrieve the value of our `score`. We achieve this by creating a **getter** and **setter** functions.

In Solidity, you declare a `function` with the keyword function followed the *function name* (here `getScore()`).

```
contract Score {

    uint score = 5;

    function getScore() returns (uint) {
        return score;
    }

    function setScore(uint new_score) {
        score = new_score;
    }
}
```

Let's look at both functions in detail.

Getter function using `return`

Definiton : In Solidity, a **getter** is a function that returns a value.

To return a value from a function (here our `score`), you use the following keywords:

- *In the function definition:* `returns` + variable type returned between parentheses for example (`uint`)
- *In the function body:* `return` followed by what you want to return for example `return score;` or `return 137;`

Setter function: pass parameters to our function

Definition : In Solidity, a **setter** is a function that modifies the value of a variable (**modifies the state of the contract**). To creates a **setter**, you must specify the **parameters** when you declare your function.

After your function name, specifies between parentheses 1) the **variable type** (`uint`) and 2) the **variable name** (`new_score`)

> ## Compiler Error:
>
> Try entering this code in Remix. We are still not there. The compiler should give you the following error:
>
> ```
> Syntax Error: No visibility specified. Did you intend
> to add "public" ?
> ```
>
> Therefore, we need to specify a visibility for our function. We are going to cover the notion of **visibility** in the next section.

Function visibility

Introduction

To make our functions work, we need to specify their *visibility* in the contract.

Add the keyword `public` after your function name.

```solidity
contract Score {

    uint score = 5;

    function getScore() public returns (uint) {
        return score;
    }

    function setScore(uint new_score) public {
        score = new_score;
    }
}
```

What does the `public` **keyword mean** ?

There are four types of *visibility* for functions in Solidity : `public`, `private`, `external` and `internal`. The table below explains the difference.

Visibility	Contract itself	Derived Contracts	External Contracts	External Addresses
`public`	✓	✓	✓	✓
`private`	✓			
`internal`	✓	✓		
`external`			✓	✓

Learn More:

- Those keywords are also available for state variables, except for `external`.
- For simplicity, you could add the `public` keyword to the variable. This would automatically create a **getter** for the variable. You would not need to create a **getter** function manually. (see code below)

```
uint score public;
```

Try entering that in Remix. We are still not getting there ! You should receive the following Warning on Remix.

Compiler Warning:

```
Warning: Function state mutability can be restricted
to view.
```

`View` **vs** `Pure` **?**

- `view` functions can **only read** from the contract storage. They can't modify the contract storage. Usually, you will use `view` for getters.
- `pure` functions can **neither read nor modify** the contract storage. They are only used for *computation* (like mathematical operations).

Because our function `getScore()` only reads from the contract state, it is a `view` function.

```solidity
function getScore() public view returns (uint) {
    return score;
}
```

Adding Security with Modifiers

Our contract has a security issue: **Anyone can modify the score.**

Solidity provides a global variable `msg`, that refers to the address that interacts with the contract's functions. The `msg` variables offers two associated fields:

- `msg.sender`: returns the address of the caller of the function.
- `msg.value`: returns the value in **Wei** of the amount of Ether sent to the function.

How to restrict a function to a specific caller ?

We should have a feature that enables only certain addresses to change the score (your address). To achieve this, we will introduce the notion of **modifiers**.

Definition : A `modifier` is a special function that enables us to change the behaviour of functions in Solidity. It is mostly used to automatically check a condition before executing a function.

We will use the following modifier to restrict the function to only the *contract owner*.

```solidity
address owner;

modifier onlyOwner {
    if (msg.sender == owner) {
        _;
    }
}

function setScore(uint new_score) public onlyOwner {
    score = new_score;
}
```

The `modifier` works with the following flow:

1. Check that the address of the caller (`msg.sender`) is equal to `owner` address.

2. If 1) is true, it passes the check. The `_;` will be replaced by the function body where the modifier is attached.

A `modifier` can receive arguments like functions. Here is an example of a modifier that requires the caller to send a specific amount of Ether.

```solidity
modifier Fee(uint fee) {
    if (msg.value == fee) {
        _;
    }
}
```

However, we still haven't defined who the owner is. We will define that in the **constructor**.

Constructor

Definition : A **constructor** is a function that is **executed only once** when the contract is deployed on the Ethereum blockchain.

In the code below, we define the contract owner:

```solidity
contract Score {

    address owner;

    constructor() {
        owner = msg.sender;
    }

}
```

Learn More:

Constructors are optional. If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor () {}`

Warning !

Prior to version 0.4.22, constructors were defined as function with the same name as the contract. This syntax was deprecated and is not allowed in version 0.5.0.

Events

Events are only used in Web3 to output some return values. They are a way to show the changes made into a contract.

Events act like a log statement. You declare **Events** in Solidity as follow:

```
// Outside a function
event myEvent(type1, type2, ... );

// Inside a function
emit myEvent(param1, param2, ... );
```

To illustrate, we are going to create an `event` to display the new score set. This `event` will be passed within our `setScore()` function. **Remember that you should pass the score after you have set the new variable.**

```
event Score_set(uint);

function setScore(uint new_score) public onlyOwner {
    score = new_score;
    emit Score_set(new_score);
}
```

You can also use the keyword `indexed` in front of the parameter's types in the `event` definition.

It will create an **index** that will enable to search for events via Web3 in your front-end.

```
event Score_set(uint indexed);
```

Note !

`event` can be used with any functions types (`public`, `private`, `internal` or `external`). However, they are **only visible outside the contract**. events are are not visible internally by Solidity. You cannot read an `event`. So a function cannot read the `event` emitted by another function for instance.

Learn more !

When you call an `event` in Solidity, the arguments passed to it are stored in the transaction's log - a special data structure in the Ethereum blockchain. These logs are associated with the address of the contract, incorporated into the blockchain, and stay there as long as a block is accessible.

References Data Types: Mappings

Mappings are another important complex data type used in Solidity. They are useful for association, such as associating an address with a balance or a score. You define a mapping in Solidity as follow:

```solidity
mapping(KeyType => ValueType) mapping_name;
```

You can find below a summary of all the datatypes supported for the key and the value in a mapping.

Type	Key	Value
`int/uint`	✓	✓
`string`	✓	✓
`bytes`	✓	✓
`address`	✓	✓
`struct`	X	✓
`mapping`	X	✓
`enums`	X	✓
`contract`	X	✓
`fixed-sized array`	✓	✓
`dynamic-size array`	X	✓
`variable`	X	X

You can access the value associated with a key in a mapping by specifying the key name inside square brackets `[]` as follows: `mapping_name[key]`.

Our smart contract will store a mapping of all the user's addresses and their associated score. The function `getUserScore(address _user)` enables to retrieve the score associated to a specific user's address.

```solidity
mapping(address => uint) score_list;
```

```solidity
function getUserScore(address user) public view returns (uint) {
    return score_list[user];
}
```

Tips:

you can use the keyword public in front of a mapping name to create automatically a **getter** function in Solidity, as follows:

```solidity
mapping(address => uint) public score_list;
```

Learn More:

In Solidity, mappings do not have a length, and there is no concept of a value associated with a key.

Mappings are virtually initialised in Solidity, such that every possible key exists and is mapped to a value which is the default value for that datatype.

Reference Data Types: Arrays

Arrays are also an important part of Solidity. You have two types of arrays (`T` represents the data type and `k` the maximum number of elements):

- **Fixed size** array : `T[k]`
- **Dynamic size** array : `T[]`

```
uint[] all_possible_number;
uint[9] one_digit_number;
```

In Solidity, arrays are ordered numerically. Array indices are zero based. So the index of the 1st element will be **0**. You access an element in an array in the same way than you do for a mapping:

```
uint my_score = score_list[owner];
```

You can also used the following two methods to work with arrays:

`array_name.length` : returns the number of elements the array holds.
`array_name.push(new_element)` : adds a new element at the end of the array.

Structs

We can build our own datatypes by combining simpler datatypes together into more complex types using structs.
We use the keyword **struct**, followed by the structure name , then the fields that make up the structure.
For example:

```
struct Funder {
    address addr;
    uint amount;
}
```

Here we have created a datatype called Funder, that is composed of an address and a uint.
We can now declare a variable of that type

```
Funder giver;
```

and reference the elements using dot notation

```
giver.addr = address
(0xBA7283457B0A138890F21DE2ebCF1AfB9186A2EF);
giver.amount = 2500;
```

The size of the structure has to be finite, this imposes restrictions on the elements that can be included in the struct.

Example of a contract using reference datatypes

```
pragma solidity ^0.8.0;

contract ListExample {

  struct DataStruct {
    address userAddress;
    uint userID;
  }
```

```solidity
  DataStruct[] public records;

  function createRecord1(address _userAddress, uint _userID)
public  {
     DataStruct memory newRecord;
     newRecord.userAddress = _userAddress;
     newRecord.userID    = _userID;
  }

  function createRecord2(address _userAddress, uint _userID)
public  {

records.push(DataStruct({userAddress:_userAddress,userID:_userID}
));
  }

  function getRecordCount() public view returns(uint recordCount)
{
     return records.length;
  }
}
```