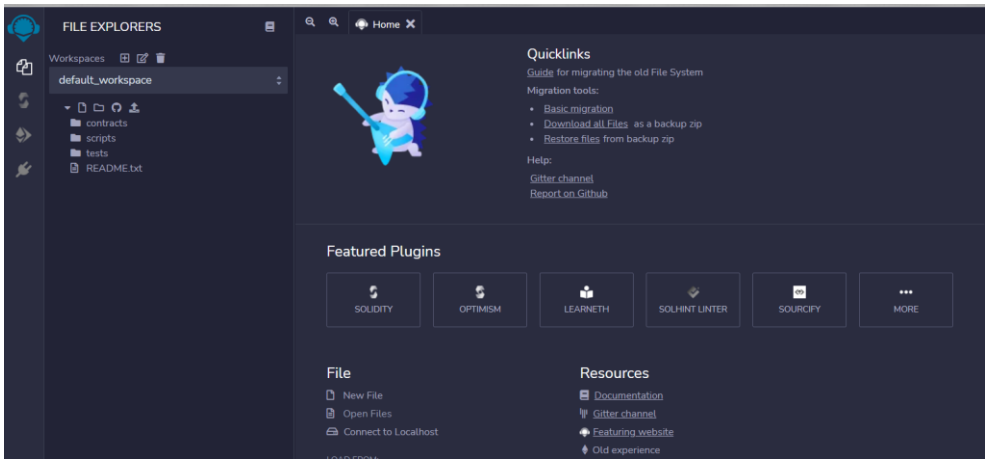
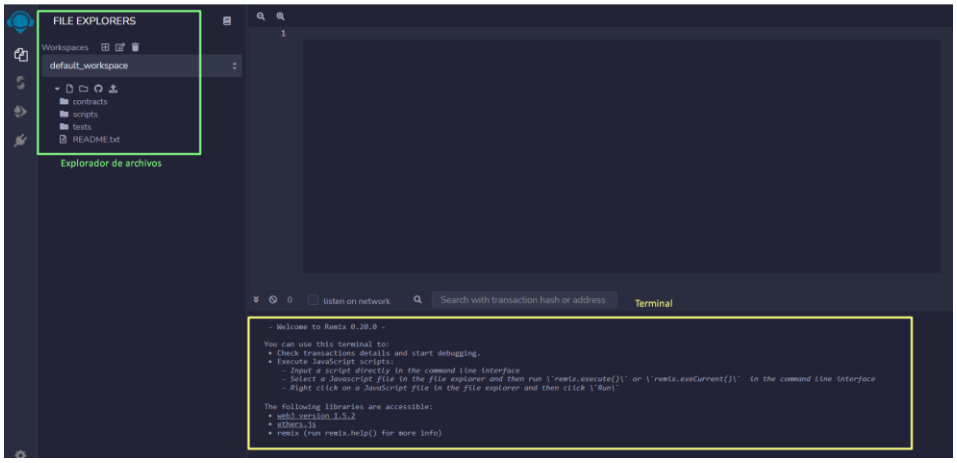
1. Debemos ingresar a Remix IDE link https://remix.ethereum.org/



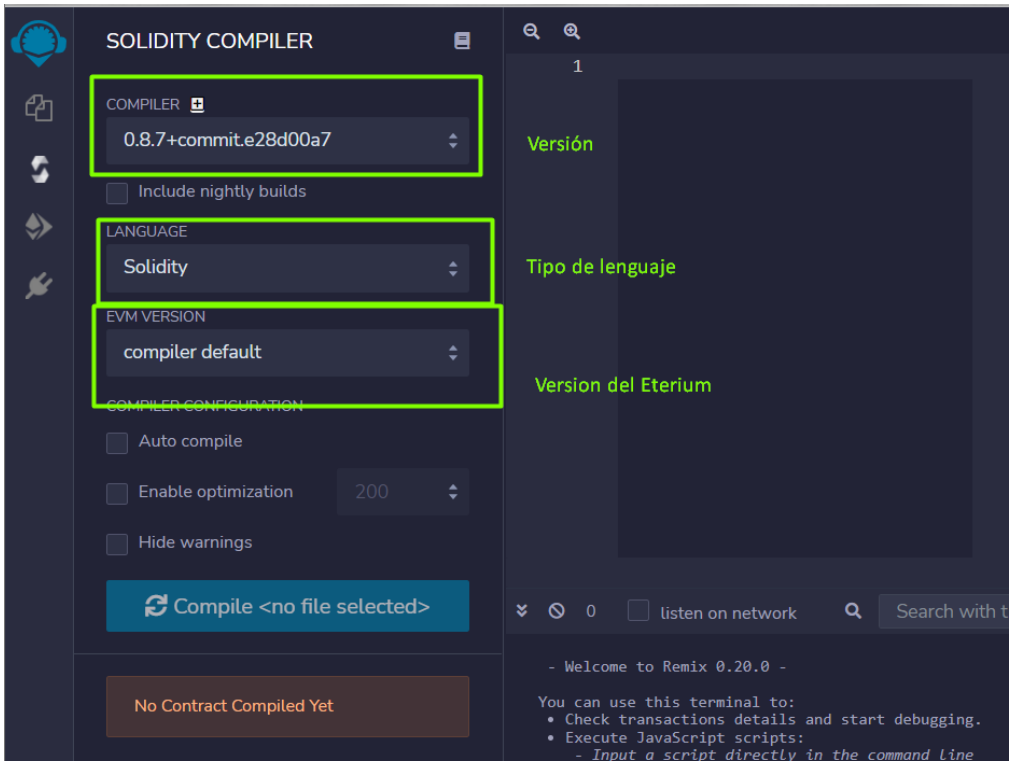
2. En la parte superior izquierda tendremos



El explorador de archivos nos permite visualizar y navegar a través de nuestro archivos, contratos, scripts y test.

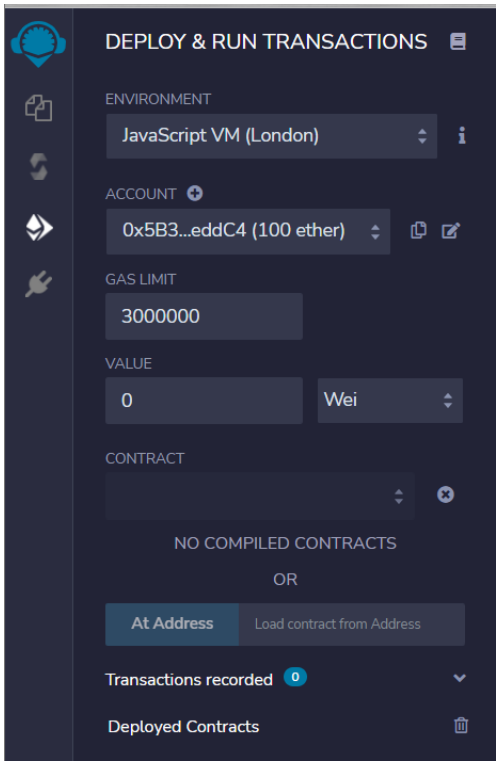
La terminal permite ver la ejecución de las funciones y los resultados de las mismas. En contratos es importante permite ver si las transacciones son exitosas o no.

3. Solidify Compiler



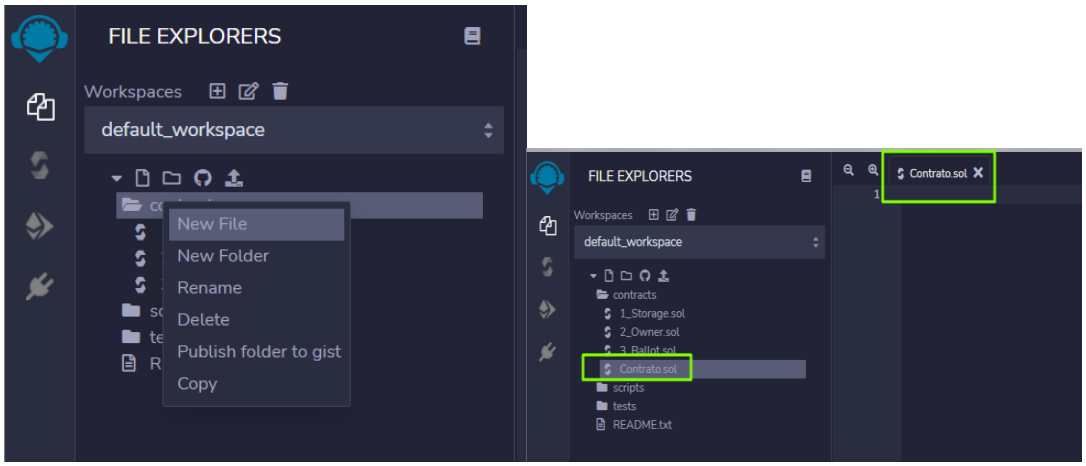
Es el compilador integrado del IDE, podemos seleccionar las versiones del lenguaje, el lenguaje que queremos usar y la versión del Eterium que nos va indicar donde se estará ejecutando el programa.

4. Transacciones



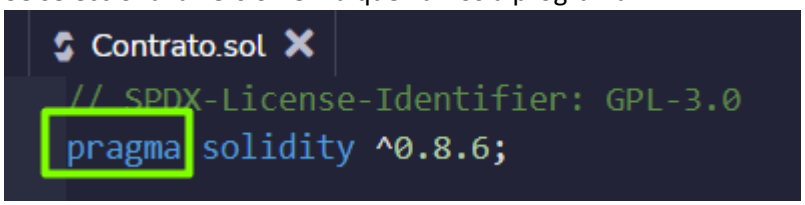
Permite que podemos subir los archivos compilados a Blockchain como subirlo a la nube.

5. Crear nuevo contrato



Recordar que cada contrato tiene su propio archivo y no es obligatorio que interactúe con otros.

6. Se selecciona la versión en la que vamos a programar



7. Al igual que otros lenguajes debemos instanciar la clase

```
contract Contrato {
```

8. Se procede a declarar variables

```
struct Task {
    uint id;
    string name;
    string description;
}
```

Contrato.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.6;

contract Contrato {

    struct Task {
        uint id;
        string name;
        string description;
    }
```

9. Creamos una lista de tareas y le asignamos un nombre

```
Task[] tasks;
```

10. Creamos una función para las tareas

```
function createTask(string memory _name, string memory _description) public {
    tasks.push(Task(nextId, _name, _description));
    nextId++;
}
```

En este caso se crea una función que nos permita añadir datos a nuestra lista de tareas y guardarlas cuando se ejecute la función.
Usamos Next ID como variable de incremento esto nos permie que los ID se agreguen solos a medida que se van haciendo las tareas y el auto incrementable les dará el valor empezara en 0 e ira sumando 1 con su creación.

11. Se crea una función para traer datos de la tarea

```
function findIndex(uint _id) internal view returns (uint) {
    for (uint i = 0; i < tasks.length; i++) {
        if (tasks[i].id == _id) {
            return i;
        }
    }
    revert("Tarea no encontrada");
}
```

Se crea un ciclo for que recorra todas las tareas que tenemos
EL if se implementa para que nos traiga los datos de la transacción que nosotros digitemos
Llegado caso no encuentre el ID no retorna el mensaje que tenemos en “revert”.
Adicional cuando usamos “internal view” quiere decir que esta función no será visible es decir que por consola no nos va mostrar alguna información esta función será usada por otra.

12. Creación de función para leer nuestra lista de tareas y guardarla en el index

```
function readTask(uint _id) public view returns (uint, string memory, string memory) {
    uint index = findIndex(_id);
    return (tasks[index].id, tasks[index].name, tasks[index].description);
}
```

En a la función usamos el comando findIndex para el ID, nos retorna una tarea y guardarlas cuando se ejecute la función.

13. Actualizar tarea

```
function updateTask(uint _id, string memory _name, string memory _description) public {
    uint index =  findIndex(_id);
    tasks[index].name = _name;
    tasks[index].description = _description;
}
```

Esta tarea nos permite editar los datos de una tarea, solicitamos los nuevos datos y se actualizan.

14. Borrar tarea

```
function deleteTask(uint _id) public {
    uint index = findIndex(_id);
    delete tasks[index];
}
```

Esta tarea va buscar el ID ingresado y procederá a borrarla si se encuentra el ID.
Cabe mencionar que borrar un arreglo es algo complicado, pero para aplicarlo a BLOCKchain y que los datos sean históricos, lo que va hacer la función es a regresar los valores por defecto que tenia esa ubicación del array.

15. Ejecución del programa