
Remix Documentation

Release 1

yann300

Feb 13, 2018

Contents

1	Overview	3
2	Quick Start	5
3	Tutorial	7

Remix is an IDE for the smart contract programming language Solidity and has an integrated debugger and testing environment.

An up to date online version is available at remix.ethereum.org

This page will host documentation and tutorial about features Remix provides.

Please go to solidity.readthedocs.io for any information regarding Solidity

CHAPTER 1

Overview

Remix provides an integrated development environment (IDE) for smart contract development. It focuses on the development and deployment of Solidity written smart contracts.

Remix is a good solution if you intend to:

- Develop smart contracts (remix integrates a solidity editor).
- Debug a smart contract's execution.
- Access the state and properties of already deployed smart contract.
- Debug already committed transaction.
- Analyze solidity code to reduce coding mistakes and to enforce best practices.
- Together with Mist (or any tool which inject web3), Remix can be used to test and debug a Dapp (see [Debugging a Dapp using Remix - Mist - Geth](#))

Developing smart contract requires a deep understanding of the associated Blockchain technology.

!! Don't use Remix against a production network unless you are completely sure what you are doing !!

This documentation describes all the features remix provides. The GUI can be separated in 4 parts. Click on one the link to get more information.

- [./file_explorer](#)
- [./solidity_editor](#)
- [./terminal](#)
- **[./tabs_panel](#)**
 - [./compile_tab](#)
 - [./run_tab](#)
 - [./settings_tab](#)
 - [./debugger_tab](#)
 - [./analysis_tab](#)

– ../support_tab

CHAPTER 2

Quick Start

(see `./quickstart_javascriptvm`)

3.1 Accessing a shared folder in Remix IDE using Remixd

Remixd is an npm module. Its purpose is to give the remix web application access to a folder from your local computer.

The code of Remixd can be checked out [here](#).

Remixd can be globally installed using the following command: `npm install -g remixd`.

Then `remixd -s <absolute-path-to-the-shared-folder>` will start Remixd and share the given folder.

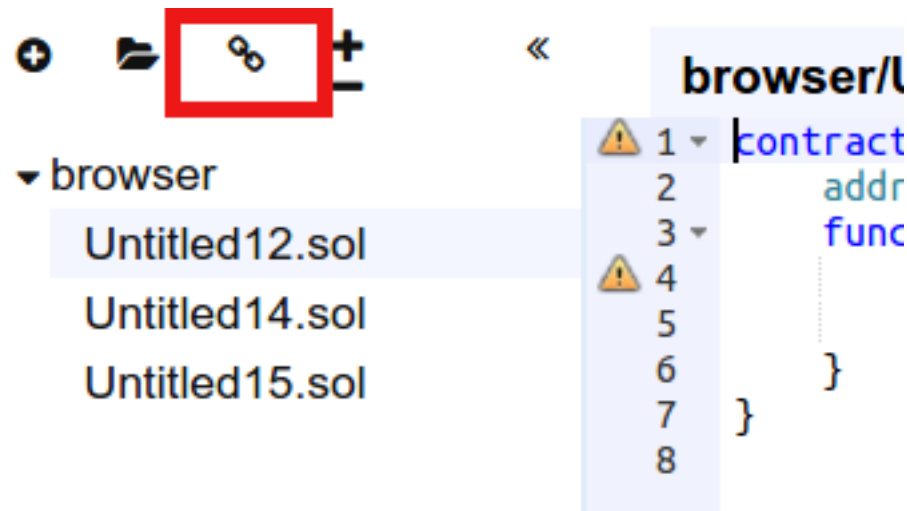
The folder is shared using a websocket connection between Remix IDE and Remixd.

Be sure the user executing Remix has read/write permission on the folder.

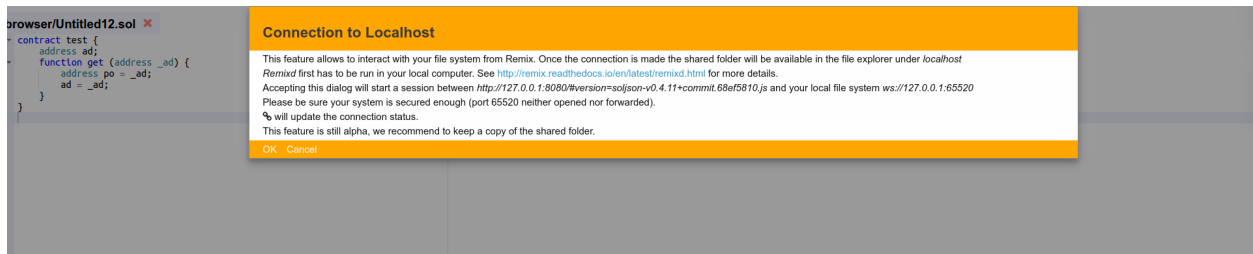
Warning: Remixd provides full read and write access to the given folder for any application that can access the TCP port 65520 on your local host.

From Remix IDE, you will need to activate the connection.

Click on the `localhost` connection icon:



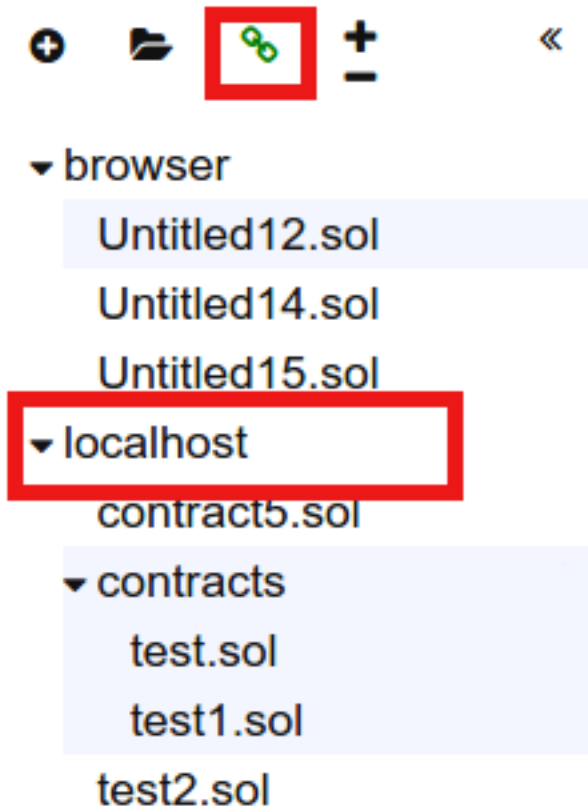
A modal dialog will ask confirmation



Accepting this dialog will start a session. Once the connection is made, the status will update and the connection icon should show up in green.

Hovering the icon will give more connection status information.

At this point if the connection is successful, the shared folder will be available in the file explorer.



3.2 Debugging a Dapp using Remix - Mist - Geth

The ultimate goal of this tutorial is to debug transactions that have been created by a dapp front end.

It is easy in Remix to debug a transaction created from its own GUI. However, setting up an environment that allows you to debug transactions created outside of Remix, require a bit more of complexity.

We will need four tools for that :

- Geth - this is the center piece and provides the blockchain environment. We will basically run geth in a *dev* mode.
- Mist - this is the Ethereum dapp browser. We will use it to browse our front end.
- Remix - this is the Ethereum IDE. We will use it to develop our Solidity contract.
- Any code editor you want - in order to write your front end :)

3.2.1 Install the environment

Install Mist

Mist is the Ethereum browser and the entry point of a Dapp.

Please download [the latest version](#) (at least 0.8.9).

Basically we will always run our front end in Mist (note that it is also possible to use [Metamask](#)).

Install Geth

[Geth](#) is the official Ethereum client.

3.2.2 Running the environment

Run Geth

We will run a test node. This node will have a new empty state and will not be synced to the main or ropsten network.

```
geth --ipcpath <test-chain-directory>/geth.ipc --datadir <test-chain-directory> --dev ↵  
↵console
```

<test-chain-directory> is the folder where keys and chain data will be stored.

--ipcpath defines the end point that other apps (like Mist) use to talk to geth.

--datadir specifies the data directory.

--dev sets the node into private chain mode and adds some debugging flags.

Then we need to create accounts and mine a bit to generate some Ether:

```
// from the geth console :  
personal.newAccount() // You can execute this command several time if you need more ↵  
↵than one account.  
miner.start() // generate some Ether.  
miner.stop() // stop mining after 30s-60s - we could also keep mining.
```

Next time we run Geth, we will only need to mine transactions (no need to recreate account).

Run Mist

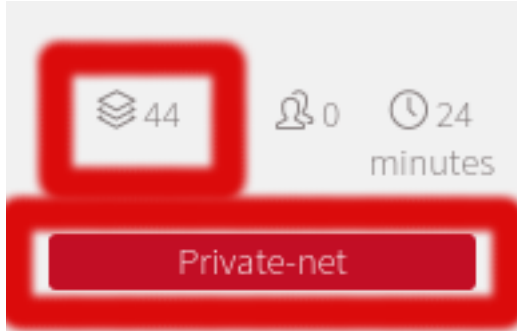
If we run Mist without any argument, its internal Geth node will run. As we have our own we need to specify the ipc path of the node installed above.

```
mist --rpc <test-chain-directory>/geth.ipc
```

(yes the option is --rpc)

Once Mist is started, verify that it is connected to the test node (that's very important!!).

On the bottom left, check that the network is `Private-net` and that the block number is the same as reported by the test node we are currently running. Run the following command in the Geth Console to check: `web3.eth.blockNumber`.



Clicking on *Wallet* will allow you to send transactions and check account balances (if you are currently mining you should see the balance increasing).

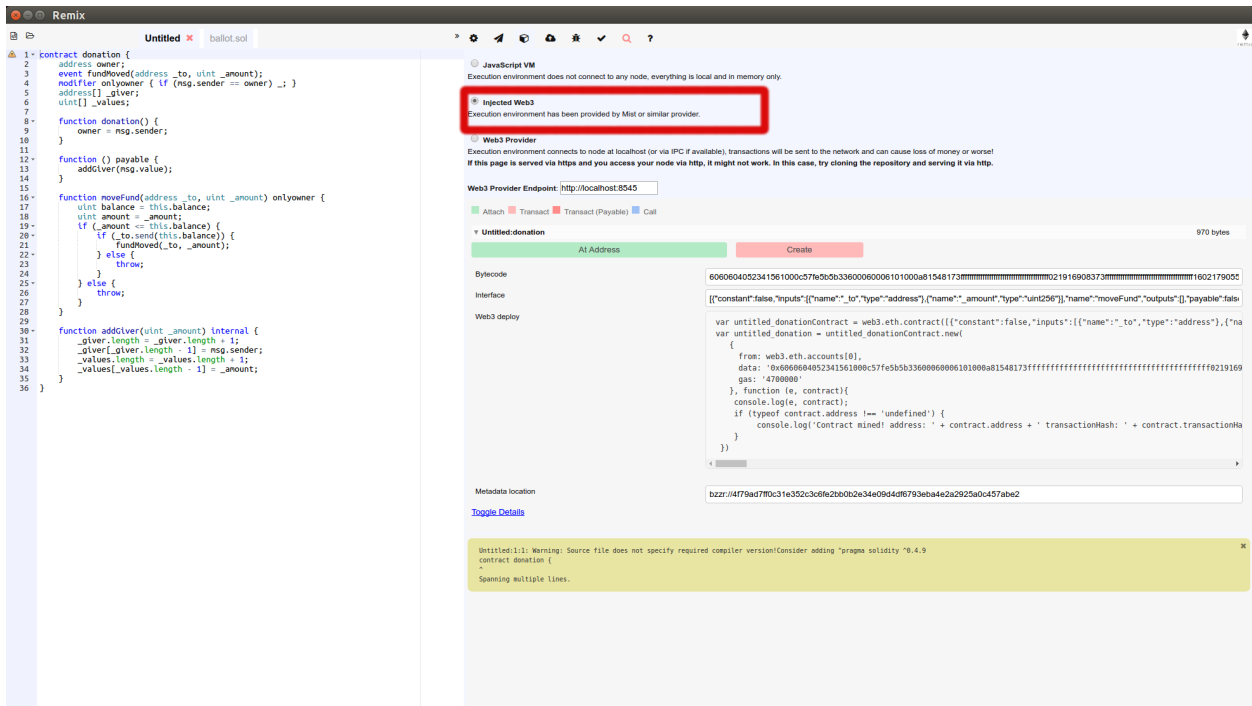
Starting Remix

In Mist click on Develop / Open Remix IDE

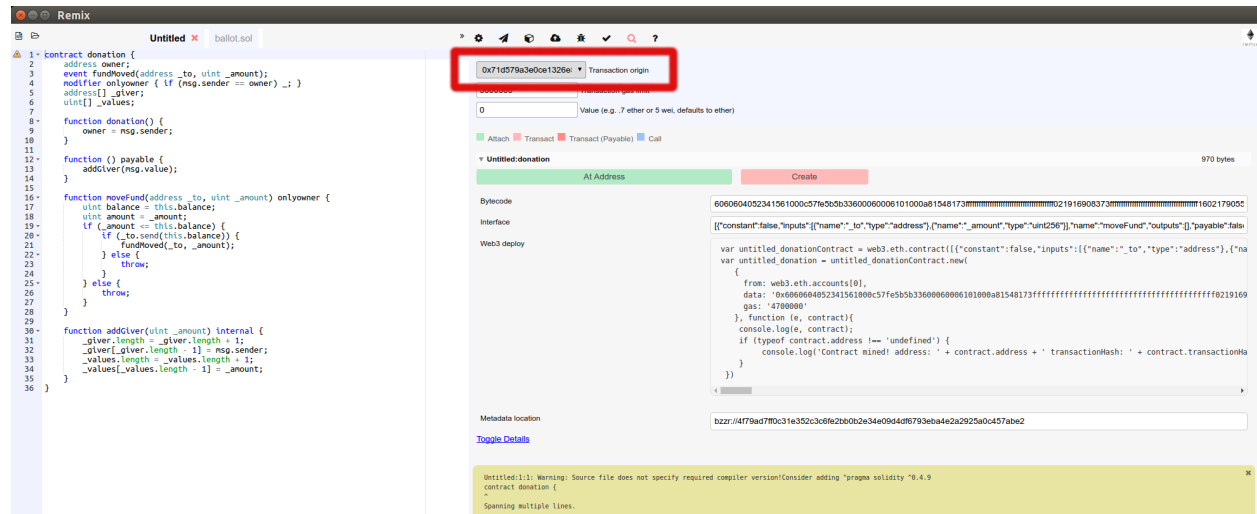
Remix will open in a new window. If this is the first time it is run, the *Ballot* contract will be loaded.

Now, we need to check if Remix is connected to Mist:

Right panel / third tab from the left, *Injected Provider* should be checked.



Right panel / second tab from the left, *Transaction* Origin should contain accounts we have previously created in Geth.



3.2.3 Developing contract / front end

Donation contract - Dapp Back end

Here is a sample solidity contract.

Copy and paste the following inside remix:

```
contract Donation {
  address owner;
  event fundMoved(address _to, uint _amount);
  modifier onlyowner { if (msg.sender == owner) _; }
  address[] _giver;
  uint[] _values;

  function Donation() {
    owner = msg.sender;
  }

  function donate() payable {
    addGiver(msg.value);
  }

  function moveFund(address _to, uint _amount) onlyowner {
    uint balance = this.balance;
    uint amount = _amount;
    if (_amount <= this.balance) {
      if (_to.send(this.balance)) {
        fundMoved(_to, _amount);
      } else {
        throw;
      }
    }
  }
}
```



```

        } else {
            throw;
        }
    }

    function addGiver(uint _amount) internal {
        _giver.push(msg.sender);
        _values.push(_amount);
    }
}

```

Dapp Front end

and here is the front end:

```

<div>
  <div>Donation Contract</div>
  <br/>
  <input id='contractaddress' placeholder='contract address'></input>
  <br/>
  <div>
    <br/>
    <input id='fromGive' placeholder='from' ></input><input placeholder='amount'
    ↪id='valueGive'></input><button id="fallbackbtn" onclick="donate()">give</button>
    <br/>
    <br/>
    <input id='fromMoveFund' placeholder='from' ></input><input id='moveFundTo'
    ↪placeholder='move to' ></input><input id='amountToMove' placeholder='amount' ></
    ↪input><button id="movefundbtn" onclick="movefund()">moveFund</button>
    <br/>
    <br/>
    <div id='wait' ></div>
  </div>
  <br/>
  <br/>
  <div id='log'>
  </div>
</div>

<script type="text/javascript">
function donate () {
  var donation = contractspec.at(document.getElementById('contractaddress').value)
  donation.donate({
    from: document.getElementById('fromGive').value,
    value: document.getElementById('valueGive').value
  }, function (error, txHash) {
    tryTillResponse(txHash, function (error, receipt) {
      alert('done ' + txHash)
    })
  })
}

function movefund () {
  var donation = contractspec.at(document.getElementById('contractaddress').value)
  donation.moveFund(
    document.getElementById('moveFundTo').value,
    document.getElementById('amountToMove').value,

```

```
function (error, txHash) {
    tryTillResponse(txHash, function (error, receipt) {
        alert('done ' + txHash)
    })
})

}

var contractspec = web3.eth.contract([{"constant":false,"inputs":[{"name":"_to","type":
↪":"address"}, {"name":"_amount","type":"uint256"}], "name":"moveFund", "outputs":[],
↪"payable":false, "type":"function"}, {"constant":false, "inputs":[], "name":"donate",
↪"outputs":[], "payable":true, "type":"function"}, {"inputs":[], "payable":false, "type":
↪"constructor"}, {"anonymous":false, "inputs":[{"indexed":false, "name":"_to", "type":
↪"address"}, {"indexed":false, "name":"_amount", "type":"uint256"}], "name":"fundMoved",
↪"type":"event"}]);

function tryTillResponse (txhash, done) {
    document.getElementById('wait').innerHTML = 'waiting for the transaction to be_
↪mined ...'
    web3.eth.getTransactionReceipt(txhash, function (err, result) {
        if (!err && !result) {
            // Try again with a bit of delay
            setTimeout(function () { tryTillResponse(txhash, done) }, 500)
        } else {
            document.getElementById('wait').innerHTML = ''
            var log = document.createElement("div")
            log.innerHTML = JSON.stringify(result)
            document.getElementById('log').appendChild(log)
            done(err, result)
        }
    })
}

</script>
```

I would suggest serving this file using `http-serve`, but you can use any web server you like.

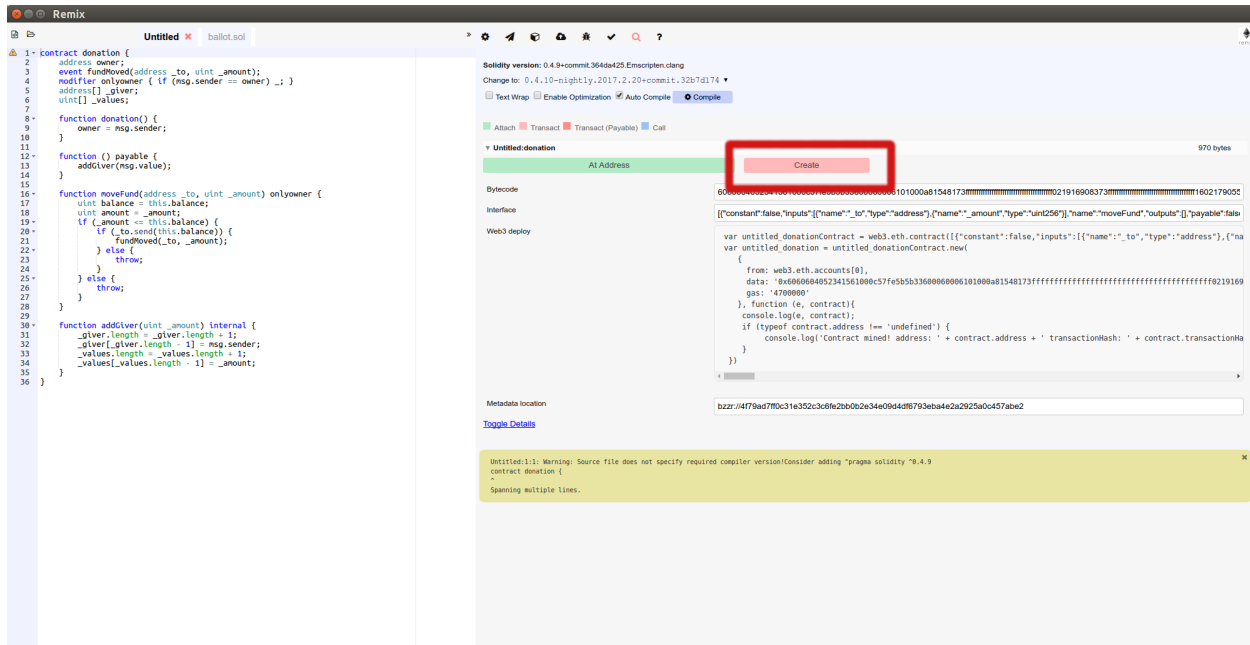
Example: Dapp Front End https://github.com/ltfschoen/dapp_front_end

Important notice !

The variable `contractspec` contains the abi of the donation contract. This means that if you change something in the contract interface (function names, parameters, ...) you need to copy the new abi from remix to the front end.

3.2.4 Deploying

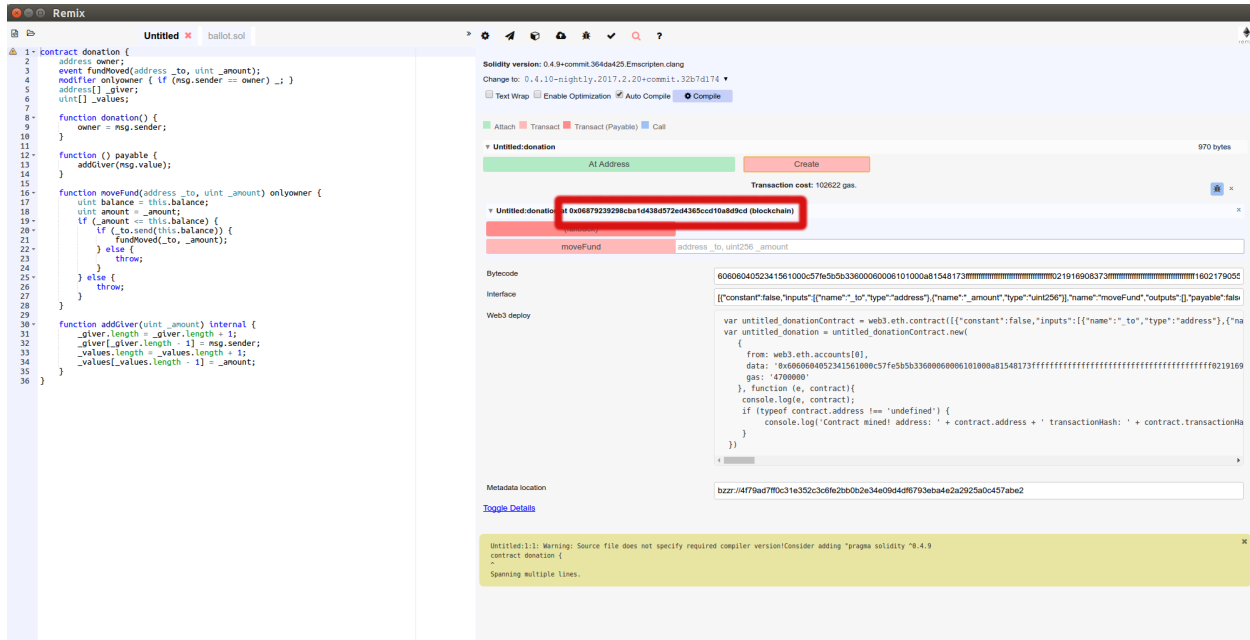
Right panel / Red button Create



This creates a new transaction that deploys the Donation contract (Mist will ask for the usual passphrase check).

Wait for the transaction to be mined (don't forget to activate mining `miner.start()`). Once this is done, you can use it by executing the `moveFund` and `donate` function. But this is not what we want to achieve. We want to run and debug those functions from the front end.

Remix also display the address of the contract. Save it, we'll need this address later.



3.2.5 Debugging

From Mist, browse the above front end. In the first field, paste the address of the newly created contract. Now, let's call the first function (label `give`).

You will need an account and a value.

The account could be any account that is declared in the Wallet section of Mist. This is the sender of the transaction that we are going to create. The value should be no more than the actual balance of the account - the unit is in *wei*, so just put 100 (100 wei), that should be fine.

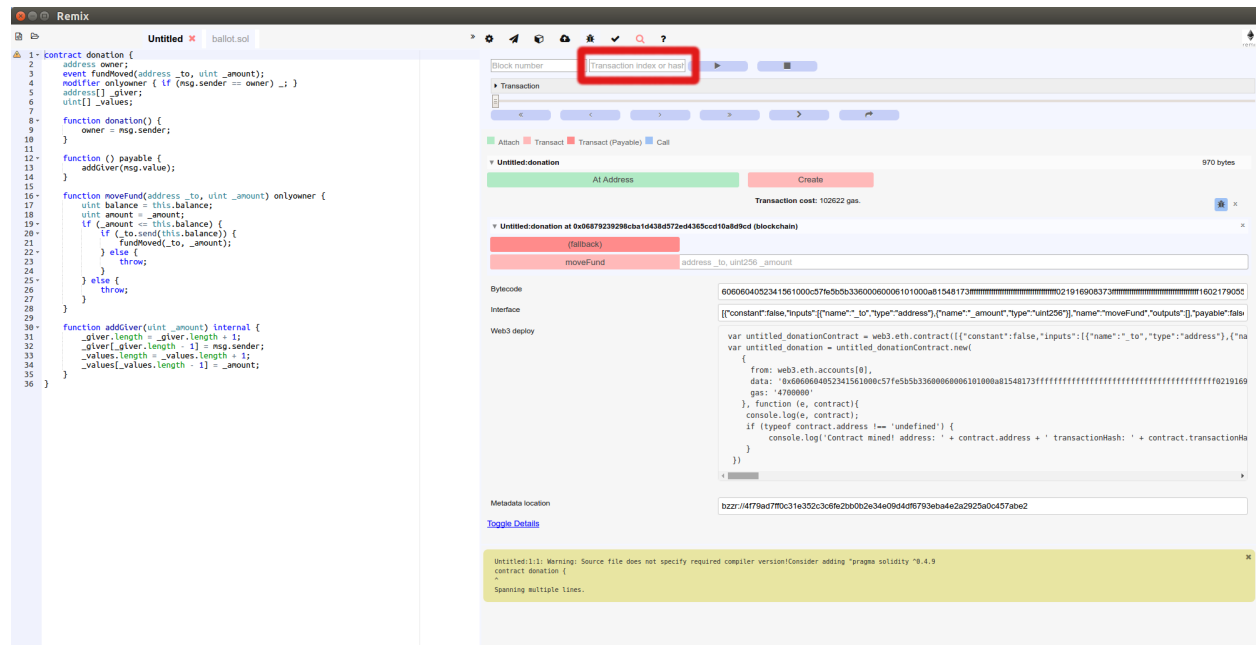
Click on Give and wait for the transaction to be mined.

The HTML block with id `log` is filled by all the transactions created from the front end. It was easier for the purpose of this tutorial to just log transactions in a div but you can have your own logging mechanism.

There is only one field that we need, this is the `transactionHash`.

Copy it and switch to Remix. On the right side, the fifth panel shows a small “bug” icon, that is the debugger.

Paste the hash into the transaction field and click on the play button.



You are now entering a debug session for the call to `donate`.

Debugging in Remix is easier than with common tools like `gdb` because you can freely move in time. Use the slider to change the current step and click on the panels below to expand them and explore the current state, local variables, etc. There are also breakpoints to move between sections of the code quickly, but more on all that later.

At the time of this writing, there is an issue that could break the contract creation. The a workaround for that at <https://github.com/ethereum/go-ethereum/issues/3653>. Please follow the workaround or wait for this issue to be closed.

Also, although retrieving a contract’s storage when Remix is using the JavaScript VM is working well, there is still work to be done when Remix is using `eth` or `geth` as backend.

3.3 Debugging a Transaction

The goal of this tutorial is to explain how to debug transaction using Remix.

3.3.1 Start debugging

There are two different ways to start debugging, each way correspond to a different use case.

From the Transaction GUI

We will not explain in detail here how to write or deploy contract. Let us start with a basic contract (replace this one by your's):

```
contract Donation {
  address owner;
  event fundMoved(address _to, uint _amount);
  modifier onlyowner { if (msg.sender == owner) _; }
  address[] _giver;
  uint[] _values;

  function Donation() {
    owner = msg.sender;
  }

  function donate() payable {
    addGiver(msg.value);
  }

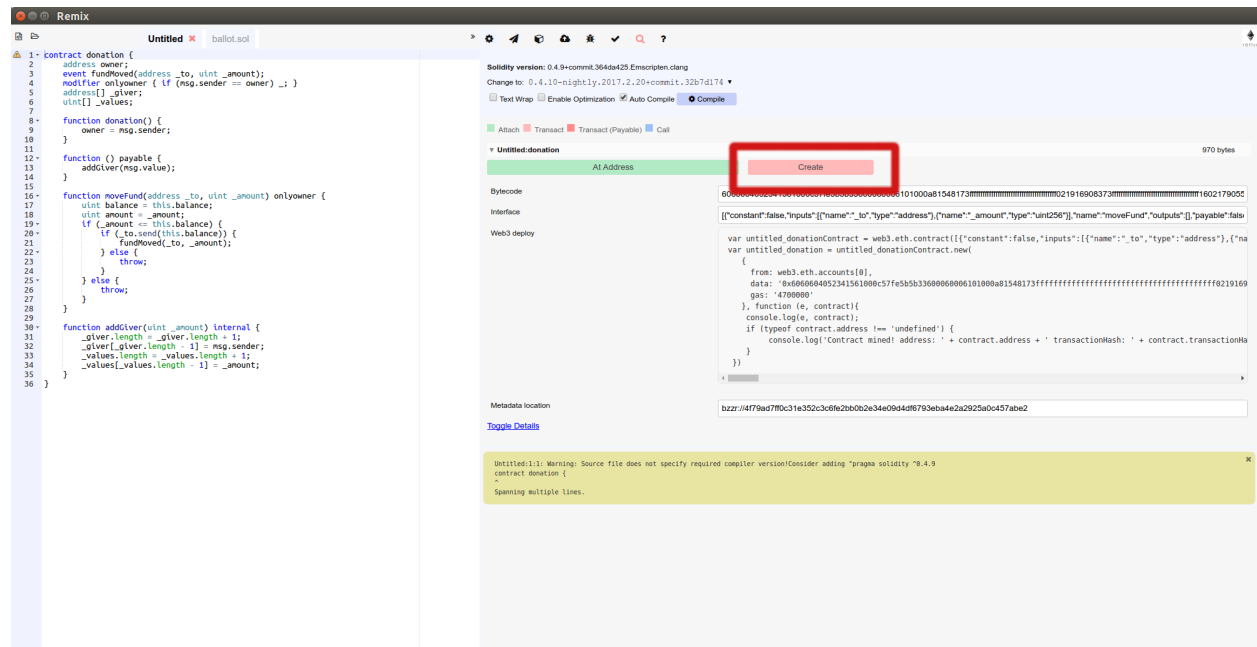
  function moveFund(address _to, uint _amount) onlyowner {
    uint balance = this.balance;
    uint amount = _amount;
    if (_amount <= this.balance) {
      if (_to.send(this.balance)) {
        fundMoved(_to, _amount);
      } else {
        throw;
      }
    } else {
      throw;
    }
  }

  function addGiver(uint _amount) internal {
    _giver.push(msg.sender);
    _values.push(_amount);
  }
}
```

For the purpose of this tutorial, we will run the JavaScript VM (that's the default mode when you don't use Remix with Mist or Metamask). This simulates a custom blockchain. You could do the same using a proper backend node.

Now, let's deploy the contract:

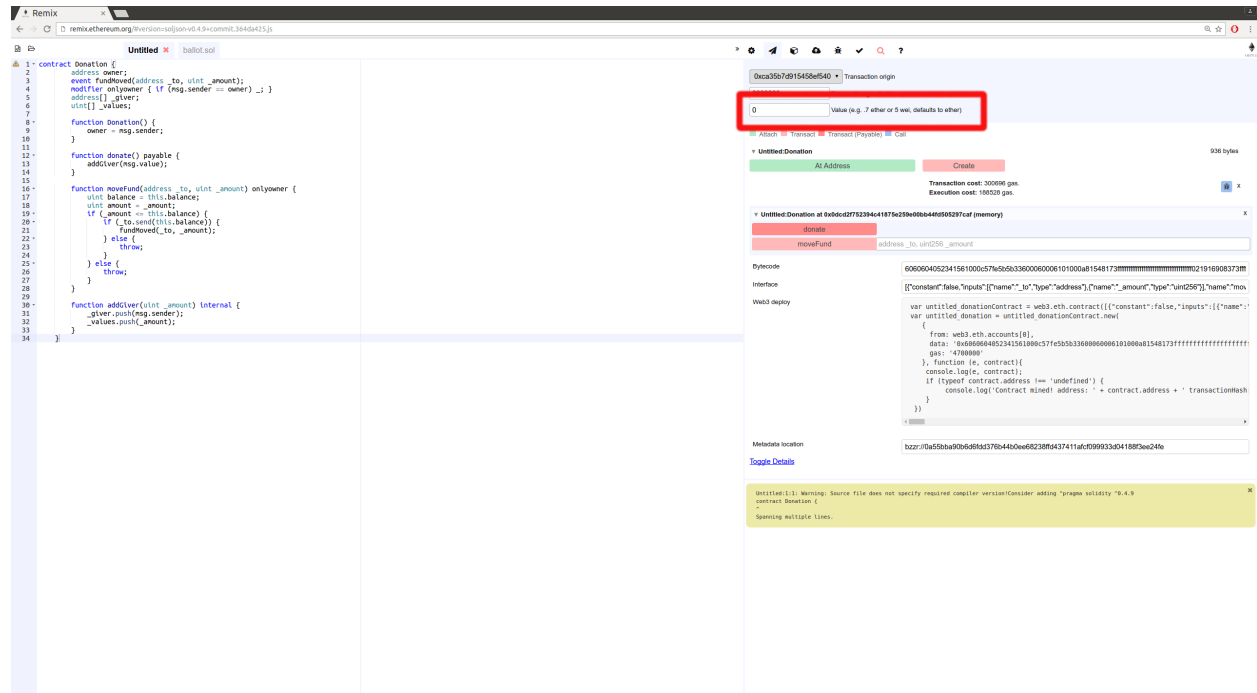
Right panel / Red button Create



Then we should call the `Donate` function (that will send Ether to the contract).

Let's set the amount of Ether:

Right panel / second tab from the left - fill in the "value" input ('1 ether' for instance)



Then click on `Donate`. As we are using the JavaScript VM, everything goes almost instantly.

Remix displays also some information related to each transaction result. In the terminal, the transaction is logged and you can start debugging it.

The screenshot displays the Remix IDE interface. The top panel shows a Solidity contract named `Donation` with the following code:

```

1 contract Donation {
2   address owner;
3   event fundMoved(address _to, uint _amount);
4   modifier onlyowner { if (msg.sender == owner) _; }
5   address[] _giver;
6   uint[] _values;
7
8   function Donation() {
9     owner = msg.sender;
10  }
11
12  function donate() payable {
13    addGiver(msg.value);
14  }
15
16  function moveFund(address _to, uint _amount) onlyowner {
17    uint balance = this.balance;
18    uint amount = _amount;
19    if (_amount <= this.balance) {
20      if (_to.send(this.balance)) {
21        fundMoved(_to, _amount);
22      } else {
23        throw;
24      }
25    } else {
26      throw;
27    }
28  }
29
30  function addGiver(uint _amount) internal {
31    _giver.push(msg.sender);
32    _values.push(_amount);
33  }
34 }

```

The bottom panel shows the execution logs. The first log is:

```

[vm] from:0xca3...a733c, to:browser/Untitled.sol:Donation.constructor(), value:0 wei, data:0x606...b0029, 0 logs, hash:0xe6f...4c3f5

```

The second log, which is highlighted with a red border, is:

```

[vm] from:0xca3...a733c, to:browser/Untitled.sol:Donation.donate() 0x692...77b3a, value:1000000000000000000 wei, data:0xed8...8c68e, 0 logs, hash:0xd05...12769

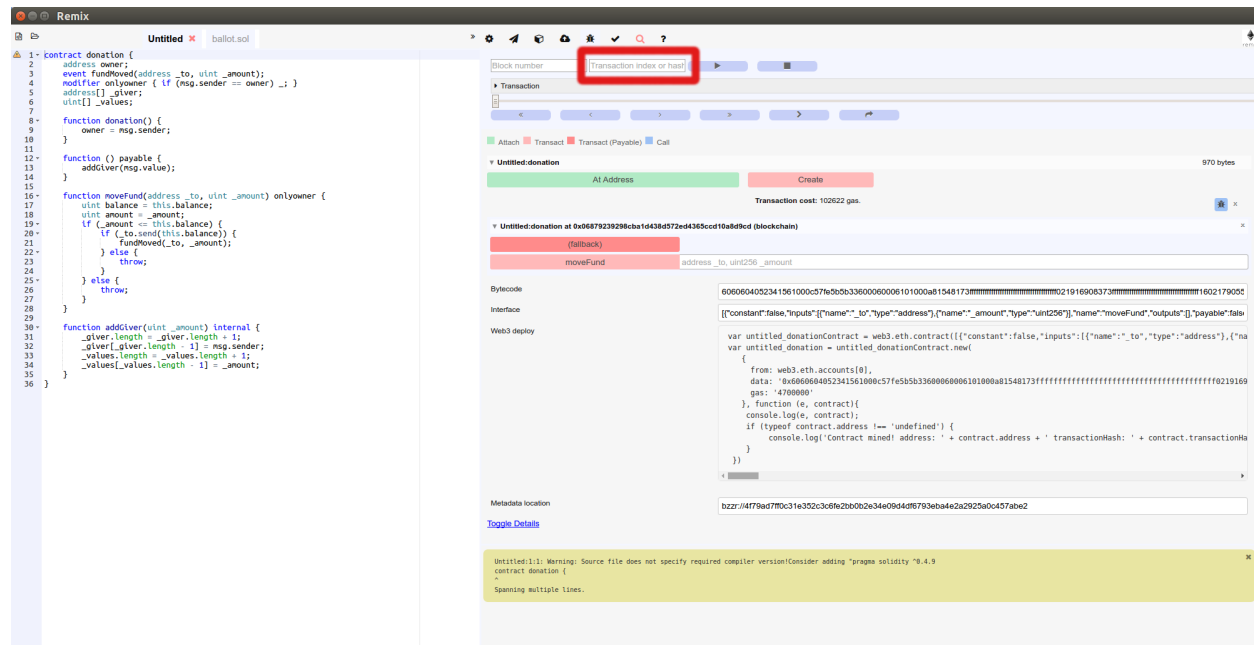
```

Below the logs, there are buttons for "Details" and "Debug".

From the Debugger

The debugger can be found in the right panel / 5th tab from the left.

You can start a debug session by providing either a transaction hash or a block number and transaction index.

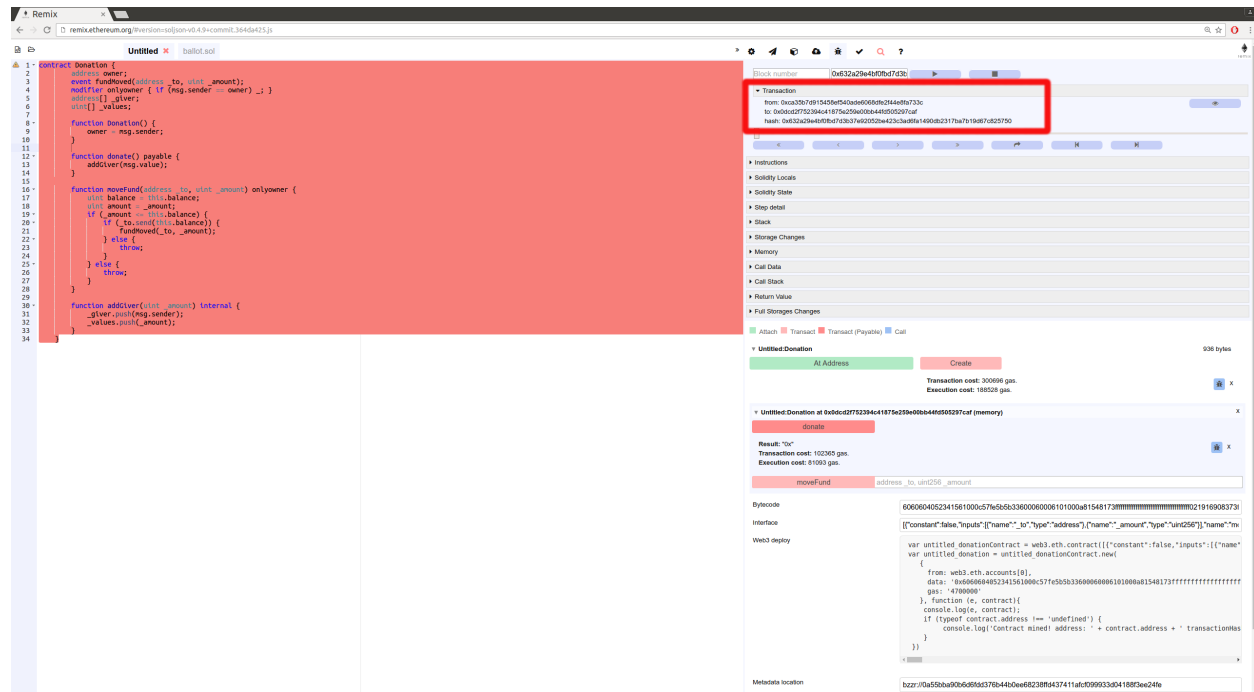


Click the play button to start debugging.

3.3.2 Using the debugger

The debugger allows one to see detailed informations about the transaction's execution. It uses the editor (left panel) to display the location in the source code where the current execution is.

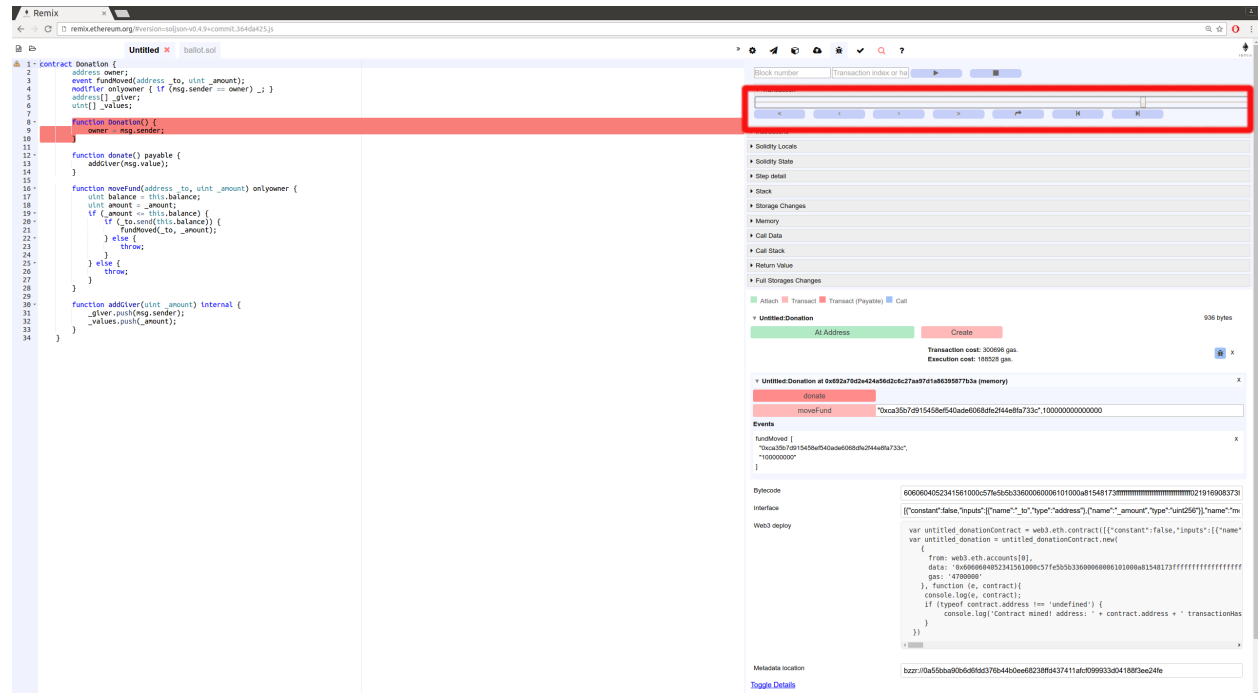
The transaction panel displays basic information about the current transaction.



The navigation part contains a slider and buttons that can be used to step through the transaction execution.

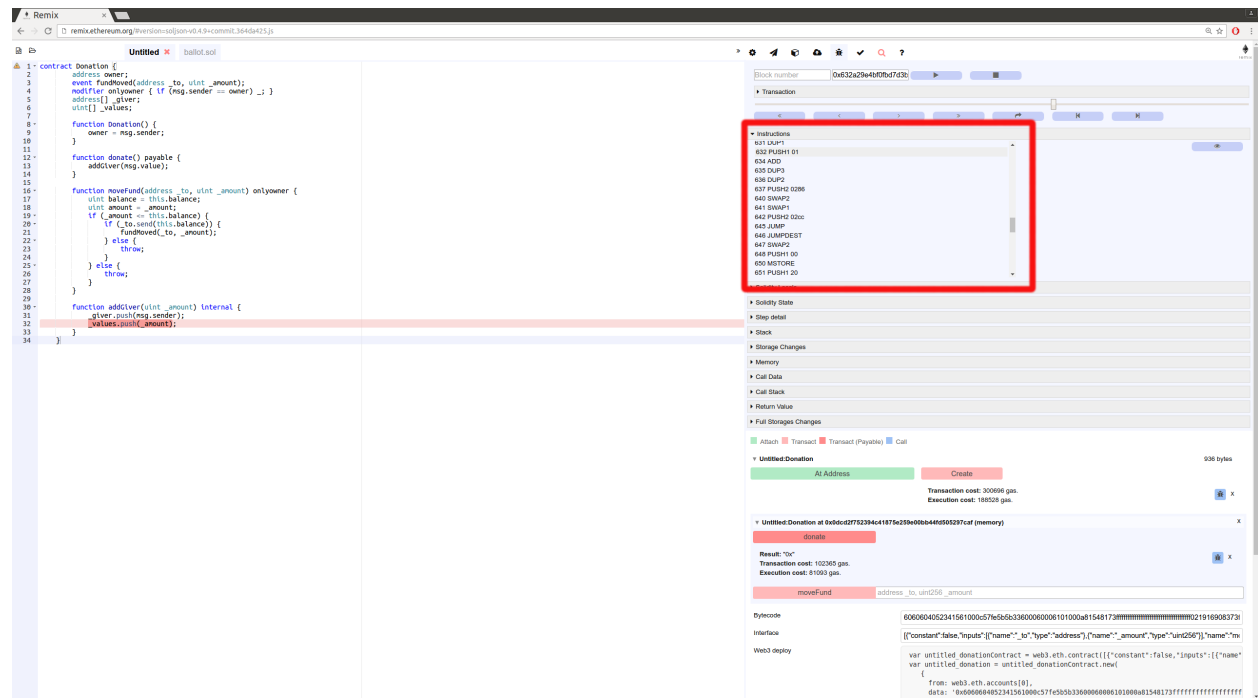
From the left to the right:

step over back, step into back, step into forward, step over forward, jump out (jump out of the current call), jump to the previous breakpoint, jump to the next breakpoint.



11 panels give detailed information about the execution:

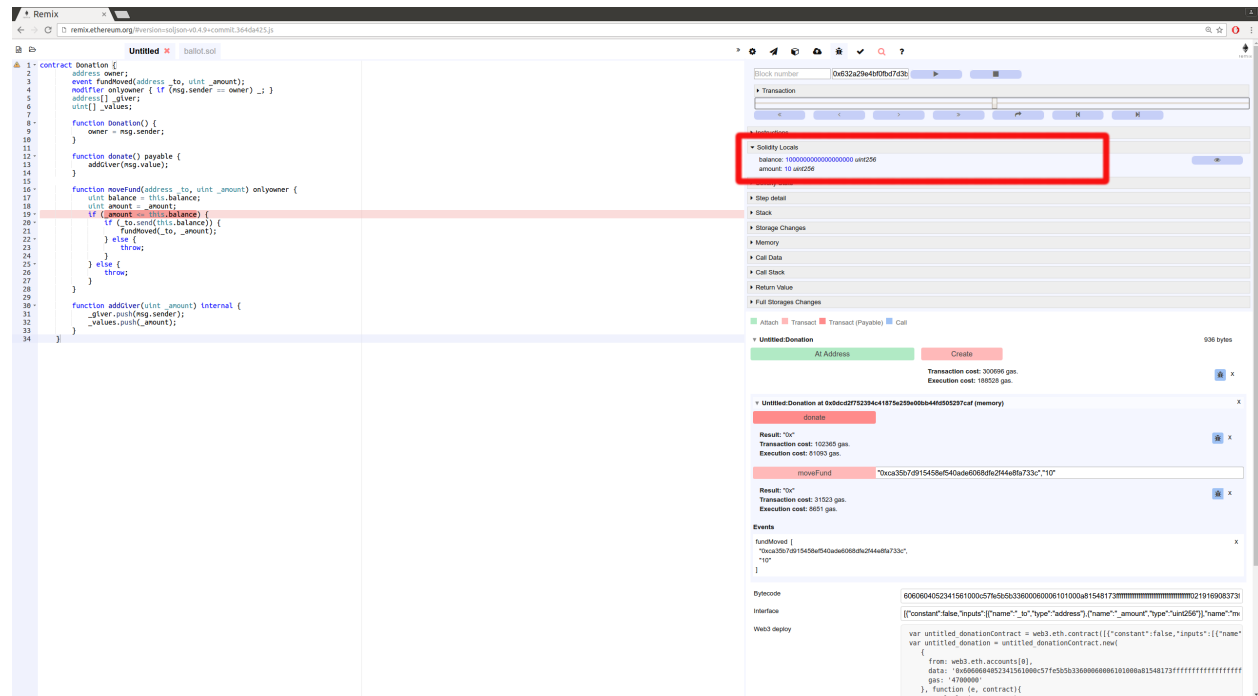
Instructions



The Instructions panel displays the bytecode of the current executing contract- with the current step highlighted.

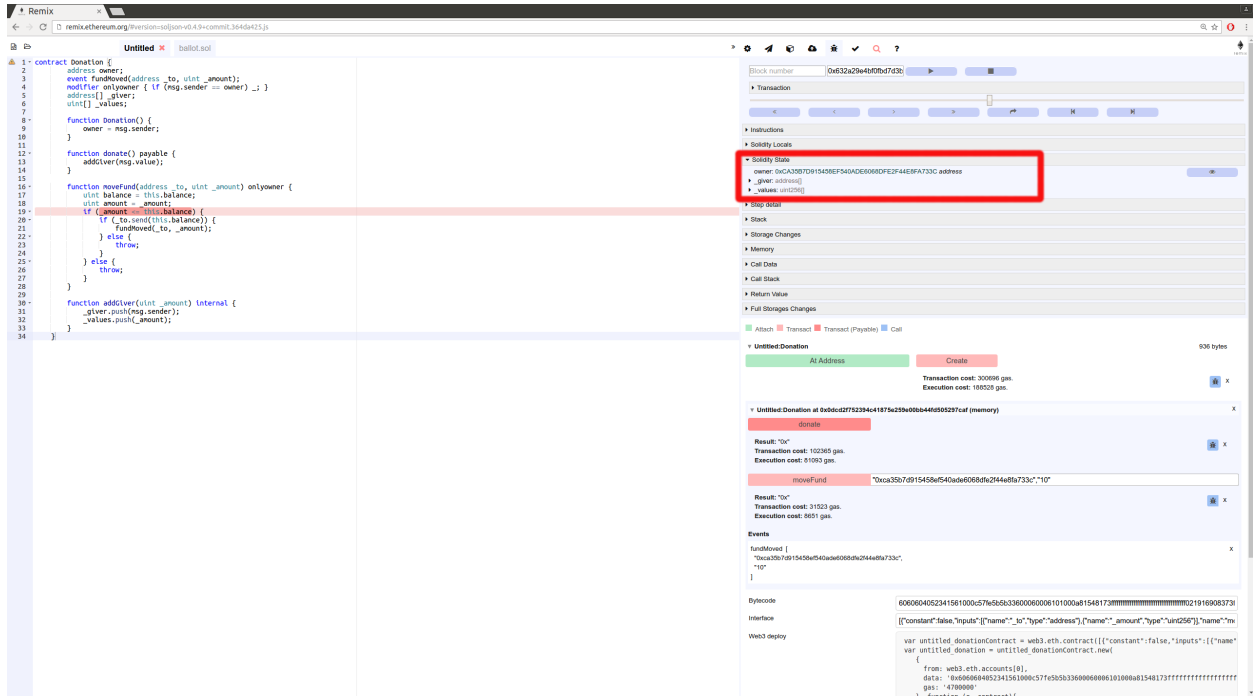
Important note: When this panel is hidden, the slider will have a coarser granularity and only stop at expression boundaries, even if they are compiled into multiple EVM instructions. When the panel is displayed, it will be possible to step over every instruction, even those that refers to the same expression.

Solidity Locals



The Solidity Locals panel displays local variables associated with the current context.

Solidity State



The Solidity State panel displays state variables of the current executing contract.

Low level panels

These panels display low level informations about the execution:

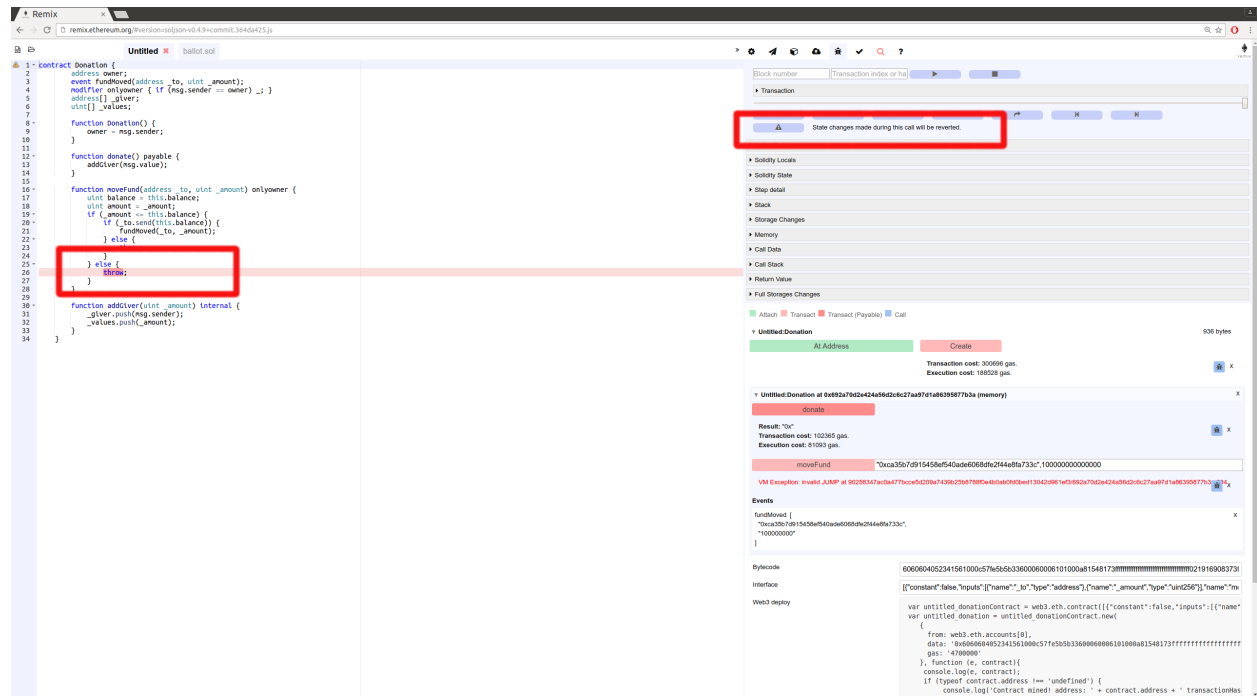
- Stack
- Storages Changes
- Memory
- Call Data
- Call Stack
- Return Value (only if the current step is a RETURN opcode)
- Full Storages Changes (only at the end of the execution - display every storage change of every modified contract)

Reverted Transaction

A transaction could be reverted (either because of out of gas exception, Solidity `throw` or low level exception).

In that case it is important to be aware of the exception and to locate where the exception is in the source code.

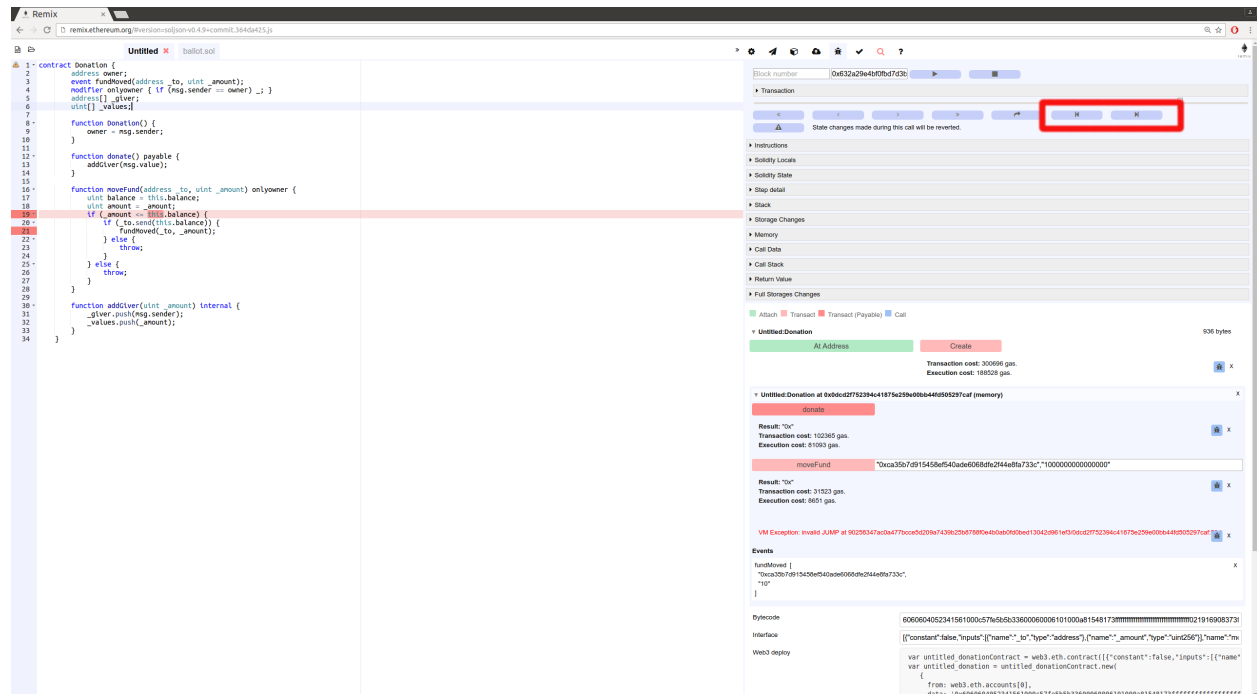
Remix will warn you when the execution throws an exception. The warning button will jump to the last opcode before the exception happened.



Breakpoints

The two last buttons from the navigation area are used to jump either back to the previous breakpoint or forward to the next breakpoint.

Breakpoints can be added and removed by clicking on the line number.



When a debug session is started, the execution will jump to the first encountered breakpoint.

Important note: If you add a breakpoint to a line that declares a variable, it might be triggered twice: Once for initializing the variable to zero and second time for assigning the actual value. As an example, assume you are debugging the following contract:

```
contract ctr {
  function hid () {
    uint p = 45;
    uint m;
    m = 89;
    uint l = 34;
  }
}
```

And let's say that breakpoints are set for the lines

```
uint p = 45;
```

```
m = 89;
```

```
uint l = 34;
```

then clicking on `Jump to next breakpoint` will stop at the following lines in the given order:

```
uint p = 45; (declaration of p)
```

```
uint l = 34; (declaration of l)
```

```
uint p = 45; (45 assigned to p)
```

```
m = 89; (89 assigned to m)
```

```
uint l = 34; (34 assigned to l)
```

3.4 Importing Source Files in Solidity

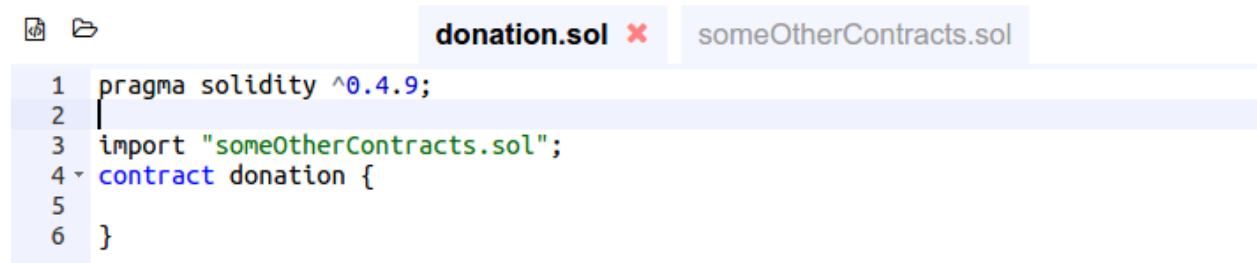
This tutorial will show you how to import local and external files.

The compilation result will also contain contracts implemented in the imported files.

For a detailed explanation of the `import` keyword see the [Solidity documentation](#)

3.4.1 Importing a local file

Other files in Remix can be imported just by specifying their path. Please use `./` for relative paths to increase portability.



```
1 pragma solidity ^0.4.9;
2 |
3 import "someOtherContracts.sol";
4 contract donation {
5
6 }
```

3.4.2 Importing from Github

It is possible to import files directly from github with URLs like `http://github.com/<owner>/<repo>/<path to the file>`.

```
1 pragma solidity ^0.4.9;
2
3 import "http://github.com/ethereum/dapp-bin/standardized_contract_apis/datafeed.sol";
4
5 contract donation {
6
7 }
```

3.4.3 Importing from Swarm

Files can be imported using all URLs supported by swarm. If you do not have a swarm node, swarm-gateways.net will be used instead.

The screenshot shows the Remix IDE interface. The main editor displays a Solidity contract named `TokenTest` with the following code:

```
1 pragma solidity ^0.4.0;
2
3 import {Token} from
4 "bzz://b17e450dadb731fd58201ed2c513d9733f8b62dff6c6318e4dc30cdb4e1bff186/std/Token.sol";
5
6 contract MyToken is Token {
7     /* implementation goes here */
8 }
9
```

The right sidebar shows the contract's ABI. It includes the Solidity version (`0.4.9+commit.364da425`), compilation options (`Text Wrap`, `Enable Optimization`, `Auto Compile`), and a legend for transaction types (`Attach`, `Transact`, `Transact (Payable)`, `Call`). The contract name `TokenTest:MyToken` is listed, followed by its address (`bzz://b17e450dadb731fd58201ed2c513d9733f8b6`) and the ABI interface.