Importing into Remix

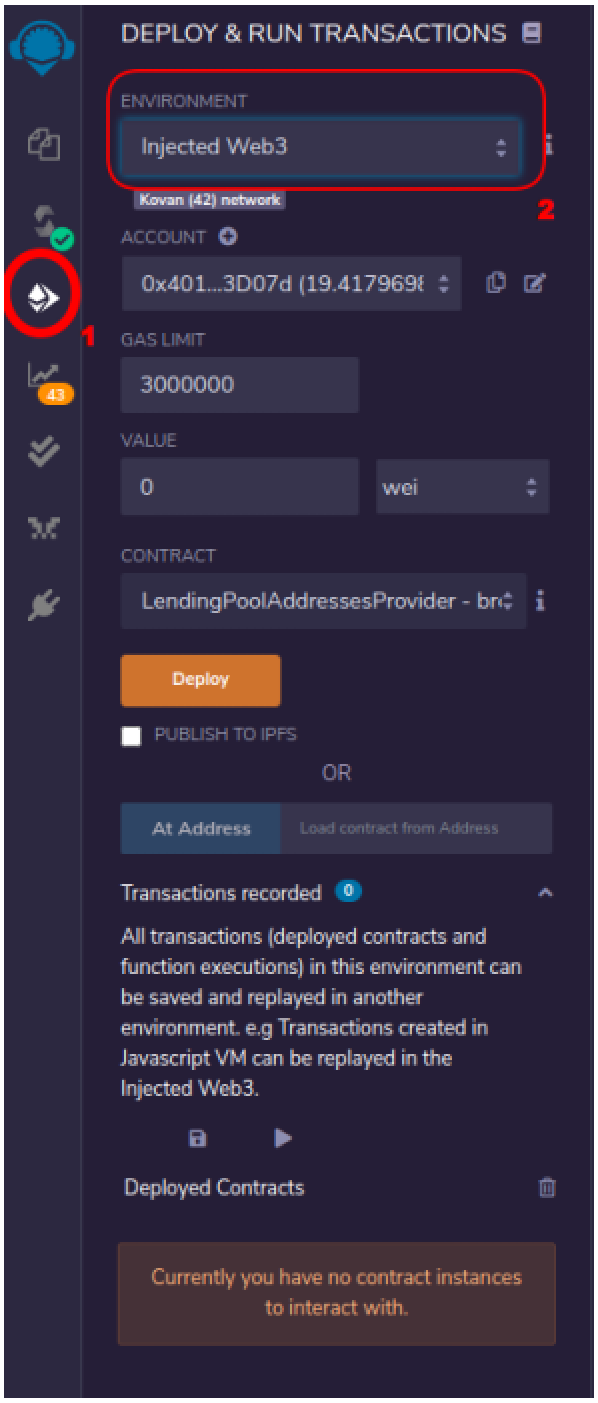
Use Firefox browser. The compiler in Remix sometimes crashes in other browsers

1. You'll need to create a blank file in Remix. This allows you to add directories
2. Create all the top level directories and treat the node dependecies like openzeppelin-solidity as a top level contract
3. Add the subdirectories. At this point you shouldn't have added any code.
4. Add the files and copy in the code.
5. Remember to change the openzeppelin-solidity import references to use relative paths.
6. Compile each file to check that you've imported correctly.
   ->pro tip: when you get an error about a contract having already been declared, look at the import above the offending line. Go to that file. One of the imports in this file has an incorrect path

Deploying to Kovan from remix

The first step to deploying is to find the innermost contract in the Aave architecture. In Aave there's a contract registry that is deployed once. All the contracts reference this registry contract to retrieve the correct versions of their dependencies.

To deploy, click on the deploy menu option (1) and set Environment to Injected Web3 (2). This will prompt your metamask to connect to Remix.


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT
Injected Web3
Kovan (42) network

ACCOUNT
0x401...3D07d (19.417969

GAS LIMIT
3000000

VALUE
0 wei

CONTRACT
LendingPoolAddressesProvider - br

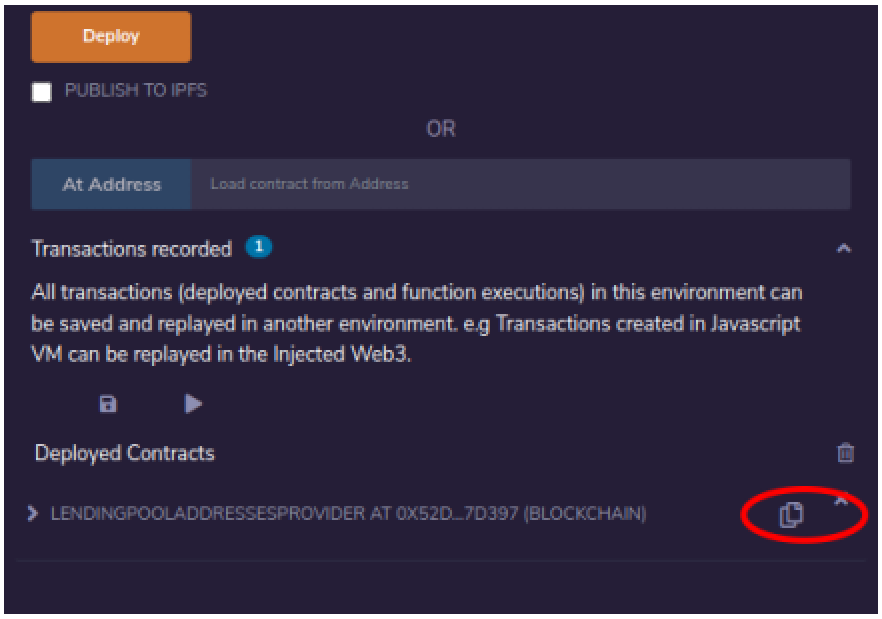
Transactions recorded 0

All transactions (deployed contracts and function executions) in this environment can be saved and replayed in another environment. e.g Transactions created in Javascript VM can be replayed in the Injected Web3.

Deployed Contracts

Currently you have no contract instances to interact with.

Deploy LendingPoolAddressesProvider.sol:
Your deployed contract will be displayed in a list.


Deploy
PUBLISH TO IPFS
OR
At Address
Load contract from Address
Transactions recorded 1

All transactions (deployed contracts and function executions) in this environment can be saved and replayed in another environment. e.g Transactions created in Javascript VM can be replayed in the Injected Web3.

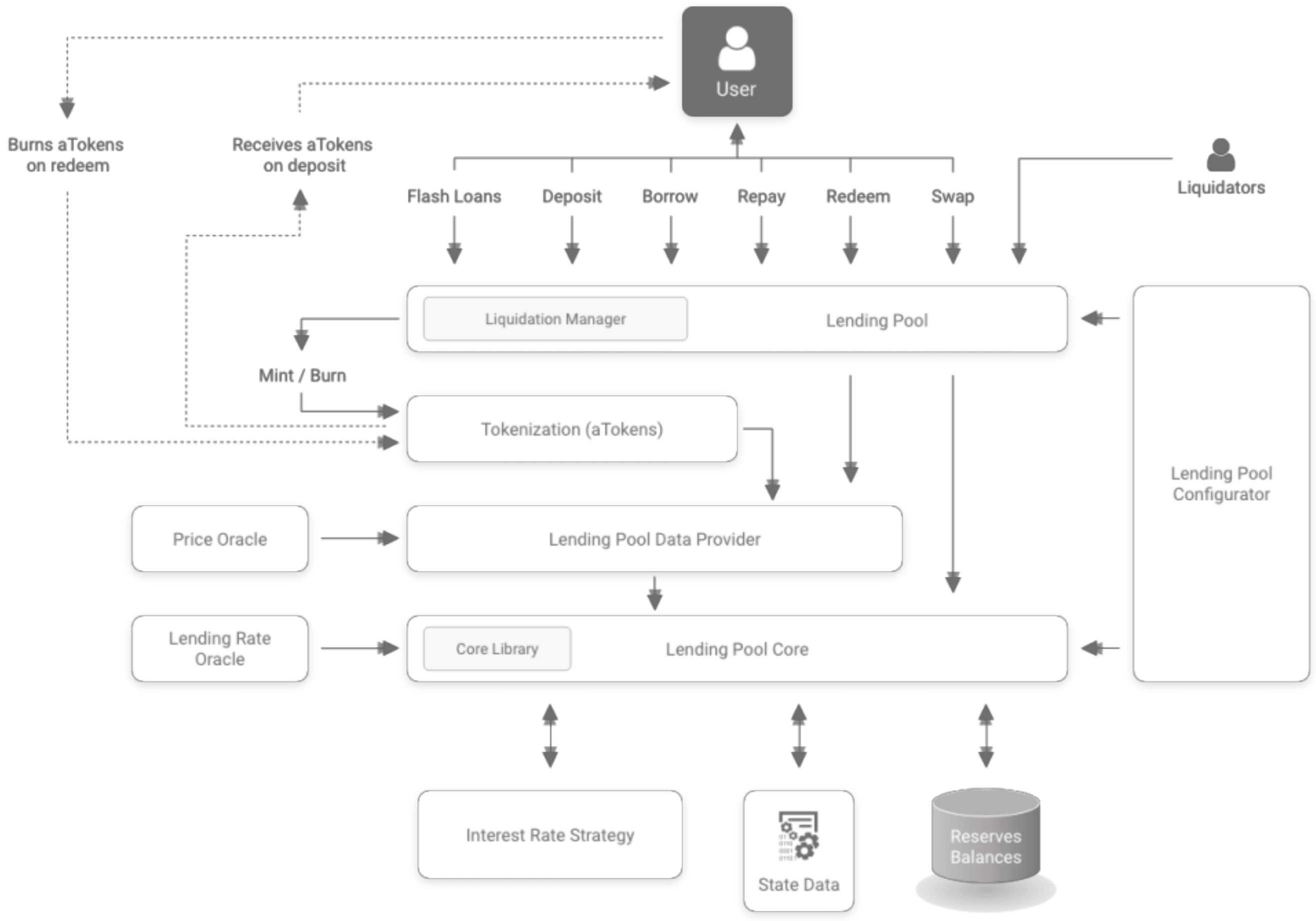
Deployed Contracts
LENDINGPOOLADDRESSESPROVIDER AT 0X52D...7D397 (BLOCKCHAIN)

Click the copy icon to retrieve the address. Don't worry about ABIs. The repository has them already.

Note: all deployments happening from 0x4011Ba13ACD2E078B0234C125D04e5C61803D07d. Since many of these contracts are owned, ownership transfer will have to occur at some point.

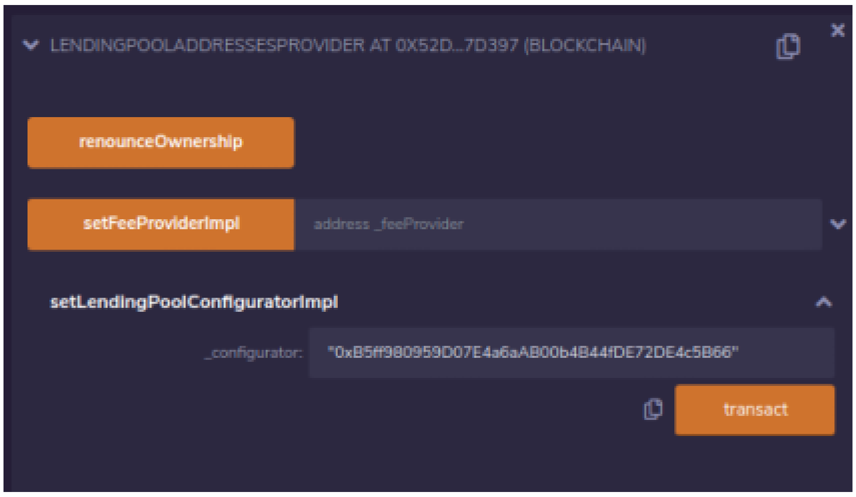
```
LendingPoolAddressesProvider Kovan address:
0x1a84C655a3F178BeA02DcBc9F46Bc84c17c36FC0
```

Next we should look at all the expected contracts in LendingpoolAddressesProvider and deploy them and their dependencies.
We'd have to start with their dependencies first. The best dependencies to start with are those that have none. The diagram below is the core architecture of the Aave smart contracts:



let's start the LendingPoolConfigurator. This contract was written before EIP-170 which sets an upper limit on contract size. To get around this, we set Remix to optimize compile.

We initially get an address for LendingPoolConfigurator. We inform LendingPoolAddress provider of its existence by calling the set function for Configurator. This spawns a new proxy contract. If we call LendingPoolAddressProvider.getLendingPoolConfigurator, we now get a new address. This is the configurator address in the system.


LENDINGPOOLADDRESSESPROVIDER AT 0X52D...7D397 (BLOCKCHAIN)
renounceOwnership
setFeeProviderImpl
address _feeProvider
setLendingPoolConfiguratorImpl
transact

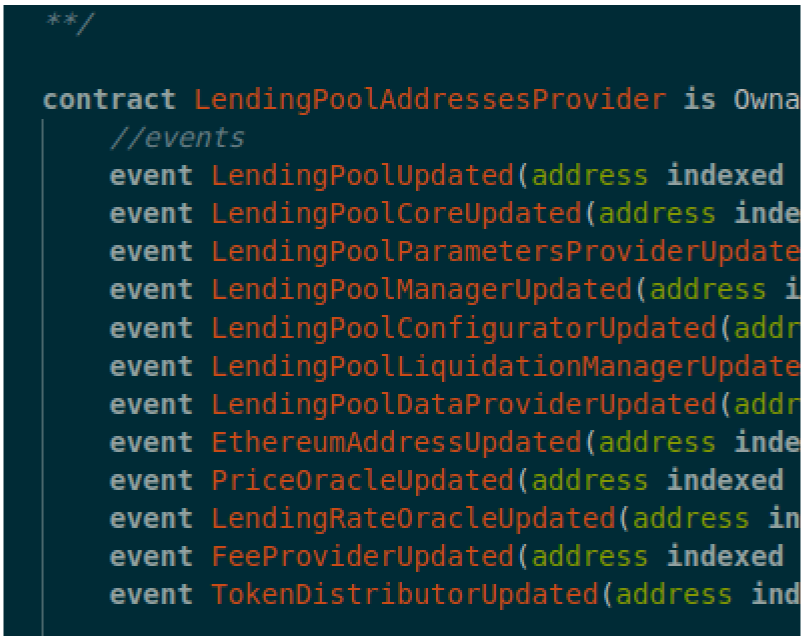
_configurator: "0xB5ff980959D07E4a6aAB00b4B44fDE72DE4c5B66"

```
LendingPoolConfigurator: 0xF1bEF1a345371716B607590003f2aBC180C7036C
```

Following the steps above,

```
LendingPoolParametersProvider: 0x71B78196dFB678A6B796AfaBFe629a1A42BCD32d
```

We'll continue to deploy and initialize all the contracts listed at the top of lendingPoolAddressProvider. Note that some of the contracts aren't wrapped in proxies so the address won't change on set.



```
 **/

contract LendingPoolAddressesProvider is Owna
    //events
    event LendingPoolUpdated(address indexed
    event LendingPoolCoreUpdated(address inde
    event LendingPoolParametersProviderUpdate
    event LendingPoolManagerUpdated(address i
    event LendingPoolConfiguratorUpdated(addr
    event LendingPoolLiquidationManagerUpdate
    event LendingPoolDataProviderUpdated(addr
    event EthereumAddressUpdated(address inde
    event PriceOracleUpdated(address indexed
    event LendingRateOracleUpdated(address in
    event FeeProviderUpdated(address indexed
    event TokenDistributorUpdated(address ind
```

```
LendingPool: 0x667091e8c1FA63977C1Ce99Ae5653Af677C17655
LendingPoolCore: 0x6444Bff5143F4722A47877774A39dD9250D529A5
LendingPoolConfigurator: 0xF1bEF1a345371716B607590003f2aBC180C7036C
LendingPoolParametersProvider: 0x71B78196dFB678A6B796AfaBFe629a1A42BCD32d
LendingPoolManager: this is a wallet address (preferably multisig). For
simplicity, I'll initialize it as my kovan deploy address:
0x4011Ba13ACD2E078B0234C125D04e5C61803D07d
LendingPoolLiquidationManager: 0xb0A659f600Ef834BA2c172EA790243bCAbe098ed
LendingPoolDataProvider: 0x9e5d3229359eB3640D60066cBb1de83F216158CE
EthereumAddress - unused in this contract
PriceOracle -> first deployed the provided mock PriceOracle:
0x31DAe81A5c6C435f8d7d43990822775181bA1403
LendingRateOracle: 0xDF9DD372dfd530972e776E91b4C9baD7d334F741
FeeProvider: 0xdd42eb4633aAE0609f72D8163edDa309f2E0f4c6
TokenDistributor: 0x8b48d8eE7B81988df5321f62a73Cb0AaD2B1F412
```

Note on the price oracle: I've given arbitrary values for the ETHUSD and Kovan Dai price so that zero isn't returned. Once the reserves are deployed, I'll update for each token.

We then add the addresses for all the mock contracts
First the MockAggregators

```
BAT: 0x2D49d05B28533365F24E0FA0FF481A18C8178c32
DAI: 0xA9B6E1f006163FB07698071b4B5a9cf13542d1eA
KNC: 0x3fcB24dC871e7318905947272A26E9b1f9F1B944
LEND: 0x17b8DABd83a0140e70D96D0733C64645A2B5CdbF
```

```
LINK: 0xAA6ceA3E84A2717035b9644f9Ff54C74D51259C6
MANA: 0xa5104818BB8D9a2023f0BbEE039f9dFaA26284F2
MKR: 0xEbA960fe41A32E52D32C77c34fE5971712FD3a3e
REP: 0x7031F74E8cCe0Ec2c8BfB5Fec688bD6ccb810B26
SUSD: 0x07136Ccc2e89264ECbed87397eA863389fC01DFD
TUSD: 0xEd1ec66F1246243861aEE307060C5baEbBbd4fB7
USDC: 0xFd1f4747cBfbBb1E5bE7f290dfeD4b5BD3C4477b
USDT: 0x1571f7693152086588A63b04a92AaF4A3c70ab4d
WBTC: 0x5968451B5Bff8fbD5446C71172450302880Dd95c
ZRX: 0xFf00975111a045099B3A13610DA9Da1842b8D491
```

Then Flashloans:

```
MockFlashLoanReceiver: 0x792c2aadB101Ee3C3F3D9487F2E899DdE8A34e27
```

The LendingPoolcore links to a library called the core library. a library is a stateless contract that provides helper functions. A very common library in use is OpenZeppelin's SafeMath which provides basic arithmetic functions that don't allow unsigned interger under- or overflows

```
CoreLibrary: 0x98Bc70e12eCd10Bf7dd52c27003703A6F96bBFF5
```

When an ERC20 token is added to Aave for deposits and loans, it is referred to as a reserve. Each reserve requires it's own ReserveInterestRateStrategy contract.
This contract informs Aave how to set borrowing and savings rates. I've deployed one for the Kovan Dai token as an example:

```
ReserveInterestRateStrategy for Dai:
0x2963bF52c63935CDa43aD774e1E2B126A81594FE
Kovan Dai: 0xc4375b7de8af5a38a93548eb8453a498222c4ff2
```