Etherlime Documentation

Limechain

Apr 29, 2020

Developer Documentation


chat
on gitter

CHAPTER 1

What is etherlime?

etherlime is an ethereum development and deployment framework based on ethers.js.

This framework provides alternative to the other web3.js based frameworks and allows for ultimate control by the developer. It also adds much needed verboseness in the deployment process so that you can be aware of what is really going on (as opposed to the general shooting in the dark technique).

This framework was born out of necessity, hardships and trouble in the development and deployment of ethereum smart contracts. We are trying to ease the pain of deployment, compilation and unit testing and add much needed stability to the process. In our mind ethers is much more stable alternative than web3.js for the moment therefore this framework is born.

Etherlime can be used as library and/or command line tool. The command line tools give you great boosts but you can skip them fully and use plain old node.js including the etherlime library.

1.1 Milestones

- [Ready] Being able to deploy compiled contracts on local and infura nodes <—- **Done**
- [Ready] Being able to compile contracts to the desired formats for deployment <—- **Done**
- [Ready] Being able to run unit tests on the compiled contracts <—- **Done**
- [Ready] Being able to run unit tests with code coverage <—- **Done**
- [Ready] Being able to debug transactions <—- **Done**
- [Not Ready] Being able to verify contracts <—- **Next**

1.2 Community

Join our community group

1.2.1 Quick Start

Installing

```
npm i -g etherlime
```

Running embedded ganache-cli

```
etherlime ganache
```

Allows the use of EtherlimeGanacheDeployer

Deploying with etherlime

Initialize etherlime

```
etherlime init
```

This will create `deployment` directory with `deploy.js` file inside. You can use this file to write your deployment procedure.

Deployer Example

```
const etherlime = require('etherlime');

const TestContract = require('../build/TestContract.json'); // Path to your etherlime␣
↪compiled contract json file

const deploy = async (network, secret) => {

    const deployer = new etherlime.EtherlimeGanacheDeployer();

    const result = await deployer.deploy(TestContract, {}); // Add params separated␣
↪with ,
}

module.exports = { deploy }
```

Verifying Smart Contract Example

```
const etherlime = require('etherlime');

const TestContract = require('../build/TestContract.json'); // Path to your etherlime␣
↪compiled contract json file

const deploy = async (network, secret, apiKey) => {
    deployer.defaultOverrides = { apiKey };
    const deployer = new etherlime.InfuraPrivateKeyDeployer(secret, network, "INFURA_
↪API_KEY");
```

(continues on next page)

(continued from previous page)

```
    const result = await deployer.deployAndVerify(TestContract, {}); // Add params␣
↪separated with ,
}

module.exports = { deploy }

Result of ``etherlime deploy`` with ``deployAndVerify`` method would be something␣
↪like this: |Verifier
result|
```

Deploying

Run the following in order to execute the deployment file mentioned above:

```
etherlime deploy
```

The deployment process is verbose and gives you real-time info about the performed actions. In addition there is a report of the actions when the deployment finishes (as not all of us monitor the deployment process constantly);

Result of `etherlime deploy` would be something like this:

History of your deploys

In order to see a list of what you've deployed you can run the following command:

```
etherlime history
```

1.2.2 Etherlime Library API

Deployer

Deployer functionality

The main functionality the deployer exposes is (obviously) the ability to deploy compiled contract.

This is achieved through the `deploy(contract, [libraries], [params])` function.

deploy(contract, [libraries], [params])

Parameters:

- `contract` - descriptor object for contract to be deployed. More details below
- `libraries` - key-value object containing all libraries which will be linked to the contract.
- `params` - the constructor params you'd need to pass on deploy (if there are any)

The contract is descriptor object that needs to have atleast the following three fields:

- `contractName` - the name of the contract
- `abi` - the abi interface of the contract
- `bytecode` - the compiled bytecode

The easiest way to get such descriptor is to compile your solidity files via *etherlime compile*

The libraries object should be in the following format:

```
{
    libraryName0: '0xAddressOfLibrary0',
    libraryName1: '0xAddressOfLibrary1'
}
```

If the contract to be deployed doesn't contains any libraries, `{}`, `undefined`, `null`, `false` or `0` can be passed. For convenience we have made the deploy function to work even without this parameter passed.

Example

Linking libraries

```
const contractUsingQueueAndLinkedList = require('...');

const libraries = {
    Queue: '0x655341AabD39a5ee0939796dF610aD685a984C53,
    LinkedList: '0x619acBB5Dafc5aC340B6de4821835aF50adb29c1'
}

await deployer.deploy(contractUsingQueueAndLinkedList, libraries);
```

Skipping linking on contract without arguments

```
const contractWithoutLibraries = require('...');

await deployer.deploy(contractWithoutLibraries);
```

Skipping linking on contract with arguments

```
const contractWithoutLibraries = require('...');

await deployer.deploy(contractWithoutLibraries, false, param1, param2);
```

deployAndVerify(contract, [libraries], [params])

The main functionality the deployAndVerify exposes is (obviously) the ability to deploy and then verify compiled contract on Etherscan. This method exposes the same features as *deploy* method, but in addition automatically verifies the deployed smart contract using Etherscan API with Etherscan API Key.

In order to use the *deployAndVerify* method of the deployer, an Etherscan API Key is used. You can create your Etherscan API Key here.

Parameters:

- `contract` - descriptor object for contract to be deployed. More details below
- `libraries` - key-value object containing all libraries which will be linked to the contract.
- `params` - the constructor params you'd need to pass on deploy (if there are any)

The deployment method reads the API Key form the deployer *defaultOverrides* object.

Passing API Key to the deployer:

- Passing the API Key to the *defaultOverrides* object:

```
deployer.defaultOverrides = { gasLimit: 4700000, gasPrice: 3000000000,␣
↪etherscanApiKey: '3DQYBPZZS77YDR15NKJHURVTV9WI2KH6UY' };
```

- Setting the API Key through the deployer *setVerifierApiKey* setter:

```
deployer.setVerifierApiKey('3DQYBPZZS77YDR15NKJHURVTV9WI2KH6UY')
```

- **Passing the API Key from *etherlime deploy* command with optional parameter *etherscanApiKey*:**
  etherlime deploy –secret="Your private key" –network="rinkeby" –etherscanApiKey="3DQYBPZZS77YDR15NKJHURVTV9WI2KH6UY"

```
const deploy = async (network, secret, etherscanApiKey) => {
const deployer = new etherlime.InfuraPrivateKeyDeployer(secret, network, "INFURA_API_
↪KEY");
deployer.defaultOverrides = { gasLimit: 4700000, gasPrice: 3000000000,␣
↪etherscanApiKey };
};
```

Network is automatically detected based on the network that the deployer is set to deploy. The supported networks are:

- `mainnet`
- `ropsten`
- `rinkeby`
- `kovan`
- `goerli`

estimateGas(contract, [libraries], [params])

Estimates the gas that this transaction is going to cost you.

Parameters:

- `contract` - descriptor object for contract to be deployed
- `libraries` - key-value object containing all libraries which will be linked to the contract.
- `params` - the constructor params you'd need to pass on deploy (if there are any)

The contract is descriptor object is the same as above.

Example

```
const estimate = await deployer.estimateGas(TestContract, randomParam1, randomParam2);
// returns something like "2470692"
```

Deployers

InfuraPrivateKeyDeployer

```
InfuraPrivateKeyDeployer(privateKey, network, apiKey, [defaultOverrides])
```

Parameters:

- `privateKey` - The private key to the deployment wallet/signer instance
- `network` - network as found in `ethers.providers.networks`
- `apiKey` - your Infura API key
- `defaultOverrides` - [Optional] object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

```
const etherlime = require('etherlime');

const TestContract = require('./TestContract.json');

const defaultConfigs = {
    gasPrice: 20000000000,
    gasLimit: 4700000,
    chainId: 0 // Suitable for deploying on private networks like Quorum
}

const deploy = async (network, secret) => {

    const deployer = new etherlime.InfuraPrivateKeyDeployer('Your Private Key Goes␣
↪Here', 'ropsten', 'Your Infura API Key', defaultConfigs);

    const result = await deployer.deploy(TestContract,
↪'0xda8a06f1c910cab18ad187be1faa2b8606c2ec86', 1539426974);
}
```

Setters

***deployer* . setPrivateKey (privateKey)**

- `privateKey` - The private key to the deployment wallet/signer instance

***deployer* . setNetwork (network)**

- `network` - network as found in `ethers.providers.networks`

***deployer* . setApiKey (apiKey)**

- `apiKey` - your Infura API key

***deployer* . setDefaultOverrides (defaultOverrides)**

- `defaultOverrides` - object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

***deployer* . setSigner (signer)**

- `signer` - ethers.Wallet instance

***deployer* . setProvider (provider)**

- `provider` - ethers.provider instance

***deployer* . setVerifierApiKey (etherscanApiKey)**

- `etherscanApiKey` - Etherscan API Key

Example

```
const deployer = new etherlime.InfuraPrivateKeyDeployer(privateKey, network, apiKey,␣
↪defaultConfigs);
const newNetwork = 'ropsten';
    deployer.setNetwork(newNetwork);
```

JSONRPCPrivateKeyDeployer

```
JSONRPCPrivateKeyDeployer(privateKey, nodeUrl, [defaultOverrides])
```

Parameters:

- `privateKey` - The private key to the deployment wallet/signer instance
- `nodeUrl` - the url to the node you are trying to connect (local or remote)
- `defaultOverrides` - [Optional] object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

```
const etherlime = require('etherlime');

const TestContract = require('./TestContract.json');

const defaultConfigs = {
    gasPrice: 20000000000,
    gasLimit: 4700000,
    chainId: 0 // Suitable for deploying on private networks like Quorum
}

const deploy = async (network, secret) => {

    const deployer = new etherlime.JSONRPCPrivateKeyDeployer('Your Private Key Goes␣
↪Here', 'http://localhost:8545/', defaultConfigs);

    const result = await deployer.deploy(TestContract);
}
```

Setters

***deployer* . setPrivateKey (privateKey)**

- `privateKey` - The private key to the deployment wallet/signer instance

***deployer* . setNodeUrl (nodeUrl)**

- `nodeUrl` - the url to the node you are trying to connect (local or remote)

***deployer* . setDefaultOverrides (defaultOverrides)**

- `defaultOverrides` - object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

***deployer* . setSigner (signer)**

- `signer` - ethers.Wallet instance

***deployer* . setProvider (provider)**

- `provider` - ethers.provider instance

***deployer* . setVerifierApiKey (etherscanApiKey)**

- `etherscanApiKey` - Etherscan API Key

Example

```
const deployer = new etherlime.JSONRPCPrivateKeyDeployer(privateKey, nodeUrl,␣
↪defaultOverrides);
    const newNodeUrl = http://localhost:9545;
    deployer.setNodeUrl(newNodeUrl);
```

EtherlimeGanacheDeployer

```
EtherlimeGanacheDeployer([privateKey], [port], [defaultOverrides])
```

Parameters:

- `privateKey` - [Optional] The private key to the deployment wallet/signer instance. Defauts to the first one in the *etherlime ganache*
- `port` - [Optional] the port you've ran the etherlime ganache on. Defaults to 8545.
- `defaultOverrides` - [Optional] object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

This deployer only works with etherlime ganache

```
const etherlime = require('etherlime');

const TestContract = require('./TestContract.json');

const defaultConfigs = {
    gasPrice: 20000000000,
    gasLimit: 4700000,
    chainId: 0 // Suitable for deploying on private networks like Quorum
}

const deploy = async (network, secret) => {

    const deployer = new etherlime.EtherlimeGanacheDeployer();

    const result = await deployer.deploy(TestContract);
}
```

Setters

***deployer* . setPrivateKey (privateKey)**

- `privateKey` - The private key to the deployment wallet/signer instance

***deployer* . setPort (port)**

- `port` - the port you've ran the etherlime ganache on.

***deployer* . setDefaultOverrides (defaultOverrides)**

- `defaultOverrides` - object overriding the deployment settings for `gasPrice` , `gasLimit` and `chainId`.

***deployer* . setNodeUrl (nodeUrl)**

- `nodeUrl` - the url to the node you are trying to connect (local or remote)

***deployer* . setSigner (signer)**

- `signer` - ethers.Wallet instance

***deployer* . setProvider (provider)**

- `provider` - ethers.provider instance

***deployer* . setVerifierApiKey (etherscanApiKey)**

- `etherscanApiKey` - Etherscan API Key

Example

```
const deployer = new etherlime.EtherlimeGanacheDeployer();
const port = 9545;
    deployer.setPort(port);
```

Deployed Contract Wrapper

Wrappers

One of the advancements of the etherlime is the result of the deployment - the `DeployedContractWrapper`

The `DeployedContractWrapper` is a powerful object that provides you with `ethers.Contract` amongst other functionalities. This allows you to start using your deployed contract right away as part of your deployment sequence (f.e. you can call initialization methods)

In addition it exposes you `verboseWaitForTransaction(transaction, transactionLabel)` function. This function can be used to wait for transaction to be mined while giving you verbose output of the state. In addition it allows you to specify a label for the transaction you are waiting for, so that you can get a better understanding of what transaction is being waited for. This comes in handy when deployment scripts start to grow.

```
const contractWrapper = await deployer.deploy(ICOTokenContract);
const transferTransaction = await contractWrapper.transferOwnership(randomAddress);
const result = await contractWrapper.verboseWaitForTransaction(transferTransaction,
↪'Transfer Ownership');
```

If you are working with EtherlimeGanacheDeployer you will have the `from` method at your disposal. It will allow you to call certain methods from other default accounts.

```
const deployer = new etherlime.EtherlimeGanacheDeployer();
const contractWrapper = await deployer.deploy(SomeContract);
const tx = await contractWrapper.from(0 /* could be string address or ethers.Wallet␣
↪instance*/).someFunction(params);
const result = await contractWrapper.verboseWaitForTransaction(tx);
```

Working with previously deployed contracts

Sometimes you want to work with already deployed contract. You can do this two ways:

etherlime.ContractAt

`etherlime.ContractAt(contract, contractAddress, [signer], [providerOrPort])`

Etherlime has a convenience method allowing you to quickly wrap contracts. Passing the contract descriptor and the address it is deployed `ContractAt` will wire up an instance of the wrapper connected to etherlime ganache on the default port and default account. Optionally you can provide an account and port to connect to etherlime ganache. Alternatively if you want to connect to another provider you can pass it as last parameter, but then you must pass a signer too which is already connected to the same provider.

```
const deployedContract = etherlime.ContractAt(ContractDescriptor,␣
↪deployedContractAddress);

const tx = await deployedContract.someMethod(randomParam);
const result = await deployedContract.verboseWaitForTransaction(tx);
```

The deployer instance

The deployer object allows you to wrap such an deployed contract by it's address and continue using the power of the wrapper object. The function you can use to achieve this is `wrapDeployedContract(contract, contractAddress)`.

```
const deployedContractWrapper = deployer.
↪wrapDeployedContract(SomeContractWithInitMethod, alreadyDeployedContractAddress);

const initTransaction = await deployedContractWrapper.init(randomParam,␣
↪defaultConfigs);
const result = await deployedContractWrapper.
↪verboseWaitForTransaction(initTransaction, 'Init Contract');
```

1.2.3 Etherlime CLI

Installing & Help

Syntax

```
npm i -g etherlime
```

Install the global etherlime to allow you to run `etherlime` commands.

Help

```
etherlime help
```

Run this command to give you all possible commands of `etherlime` + help info

Version

```
etherlime --version
```

Running this command will give you the current installed `etherlime` version

etherlime init

Syntax

```
etherlime init [output] [zk]
```

Parameters:

- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.
- `zk` - [Optional] Defines whether to include in project a zk-proof folder with primary ready to use circuit for compiling. Defaults to `false`.

Running this command will install `etherlime` in the directory you've run it and will create `deployment` directory with `deploy.js` prepared for you to use. You can use this file to write your deployment procedure. It also create `test` directory where you can write your tests. It comes with an `exampleTest.js` file which you can use as a start point. The init command generate and `package.json` for you which you can use for your npm modules.

etherlime ganache

Syntax

```
etherlime ganache [port] [output] [fork] [gasPrice] [gasLimit] [mnemonic] [count]
```

Parameters:

- `port` - [Optional] By specifying `--port` you can specify port to run the etherlime ganache. Default: 8545
- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.
- `fork` - [Optional] By specifying `--fork` you can fork from another currently running Ethereum network at a given block or at a last current block. The input to the optional parameter should be the HTTP location and port of the running network, e.g http://localhost:8545 and in addition you can specify a block number to fork from, using an `@` sign: http://localhost:8545@3349038
- `gasPrice` - [Optional] By specifying `--gasPrice` you can specify the default gas price for transactions. Default: 2000000000 wei (2 Gwei)
- `gasLimit` - [Optional] By specifying `--gasLimit` you can specify the default block gas limit. Default: 6721975
- `mnemonic` - [Optional] By specifying `--mnemonic` you can generate additional account/accounts to the accounts that are coming with `etherlime ganache` command. Please note: Running this command will modify your local `setup.json`.
- `count` - [Optional] By specifying `--count` you can specify how many accounts to generate based on the mnemomic specified with `--mnemomnic`. Defaults to: 1 and works only if `--mnemonic` is passed.

For easier integration and usage of `EtherlimeGanacheDeployer` and running local deployments you can use the embedded ganache-cli. It comes with fixed 10 accounts and a lot of ETH (191408831393027885698 to be precise)

etherlime compile

Running this command will compile all smart contracts along with imported sources. The command comes with integrated solidity and vyper compiler and would automatically fetch all files with '.sol' and '.vy' extensions and would record the compiled json object in './build' folder. Note! To enable the vyper compiler you need to have running docker.

Syntax

```
etherlime compile [dir] [runs] [solcVersion] [docker] [list] [all] [quiet] [output]␣
↪[buildDirectory] [workingDirectory] [deleteCompiledFiles]
```

Parameters:

- `dir` - [Optional] By specifying `dir` you can set the root directory where to read the contracts and place the build folder. By default `dir` is set to the current working directory `./`
- `runs` - [Optional] By specifying `runs` between 1 and 999 you enabled the optimizer and set how many times the optimizer will be run. By default the optimizer is not enabled.
- `solcVersion` - [Optional] By specifying `solcVersion` you can set the version of the solc which will be used for compiling the smart contracts. By default it use the solc version from the node_modules folder.
- `docker` - [Optional] When you want to use a docker image for your solc you should set `docker=true` in order `solcVersion` to accept the passed image.
- `list` - [Optional] By specifying `list` you can list the available solc versions. The following values can be used: `docker`, `releases`, `prereleases` and `latestRelease`. By default only 10 version are listed
- `all` - [Optional] By specifying `all` together with `list` you will be able to list all available solc versions.
- `quiet` - [Optional] Disable verboseness during compilation. By the default `quiet` is set to false.
- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.
- `buildDirectory` - [Optional] Defines the directory for placing builded contracts.
- `workingDirectory` - [Optional] Defines the folder to use for reading contracts from, instead of the default one: `./contracts`. Here can be specified also a single solidity file for compiling e.g: `/contracts/LimeFactory.sol`.
- `deleteCompiledFiles` - [Optional] Delete the files in the compilation contract directory before compiling. By the default `deleteCompiledFiles` is set to false.
- `exportAbi` - [Optional] In addition to the json build files, etherlime build *abis* folder with files containing the abi of every contract

The `solcVersion` can accept the following values:

- <undefined> - passing undefined or simply don't using the solcVersion argument will use the solc version from the local node_modules
- <version> - you can pass directly the version of the solc. Example: `--solcVersion=0.4.24`

- <image> - the image which will be used to load the solc into the docker. Example: `nightly-0.4.25-a2c754b3fed422b3d8027a5298624bcfed3744a5`
- <path> - you can pass the absolute path to a local solc
- <native> - when you set the solc version argument to `native` the compiler is using the solc globally installed on your machine

Here is example of result:

```
Georgis-MBP:academy-lecture-2 georgespasov$ etherlime compile
Compiling ./contracts/Billboard.sol...
Compilation finished successfully
```

---

etherlime deploy

Syntax

```
etherlime deploy [file] [network] [secret] [-s] [compile] [runs] [output] [apiKey]
```

Parameters:

- `file` - [Optional] By specifying `--file` you can use another file as long as you keep the structure of the file (exporting an `async deploy` function with `network` and `secret` params)
- `network` - [Optional] By specifying `--network` you can specify the network param to be passed to your deploy method
- `secret` - [Optional] By specifying `secret` you can specify the secret param to be passed to your deploy method. Comes in very handy for passing private keys.
- `-s` - [Optional] Silent - silences the verbose errors
- `compile` - [Optional] Enable compilation of the smart contracts before their deployment. By default the deployment is done with a compilation
- `runs` - [Optional] Enables the optimizer and runs it the specified number of times
- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.
- `apiKey` - [Optional] You can pass Etherscan API KEY in order to use it in the deployment script for verifying smart contracts on Etherscan.

Running this command will deploy the file specified (defaults to `./deployment/deploy.js`) The deployment process is verbose and gives you real-time info about the performed actions. In addition there is a report of the actions when the deployment finishes (as not all of us monitor the deployment process constantly):

```
Deploying contract: TestContract
Waiting for transaction to be included in a block and mined: 0xc9343d409f76fd41c5c4538570e2ad995480b00f807a6fa33eec4e1111497998
Contract TestContract deployed at address: 0x19622627778 77A29600b7c0fFC4e773b4e7Fb19d
Your deployment script finished successfully!
Here is your report:
```

Event Time	Executor	Name or Label	Tx Hash	Status	Gas Price	Gas Used	Result
28 Jun, 14:06:50	InfuraPrivateKeyDeployer	TestContract	0xc9343d409f76fd41c5c4538570e2ad995480b00f807a6fa33eec4e1111497998	Success	20000000000	2470692	0x19622627778 77A29600b7c0fFC4e773b4e7Fb19d

---

etherlime history

Syntax

```
etherlime history [limit] [output]
```

Parameters:

- `limit` - [Optional] By specifying `-limit` you can set the max number of historical records to be shown. Default is 5.
- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.

Using this command will print you historical list of execution reports

etherlime test

Syntax

```
etherlime test [path] [timeout] [skip-compilation] [gas-report] [runs] [solc-version]␣
↪[output] [port]
```

Parameters:

- `path` - [Optional] By specifying `path` you can set a path to a selected directory or you can set the path directly to the javascript file which contains your tests. By default the `path` points to `./test`.
- `timeout` - [Optional] This parameter defines the test timeout in milliseconds. Defaults to 2000 ms.
- `skip-compilation` - [Optional] This parameter controls wether a compilation will be ran before the tests are started. Default: false.
- `gas-report` - [Optional] Enables Gas reporting future that will show Gas Usage after each test. Default: false.
- `runs` - [Optional] By specifying `runs` between 1 and 999 you enabled the optimizer and set how many times the optimizer will be run. By default the optimizer is not enabled.
- `solc-version` - [Optional] By specifying `solc-version` you can set the version of the solc which will be used for compiling the smart contracts. By default it use the solc version from the node_modules folder.
- `output` - [Optional] Defines the way that the logs are shown. Choices: `none` - silences the output of logs, `normal` - see verbose logs in the console and `structured` - structured output in a file meant for inter program communication.
- `port` - [Optional] The port that the etherlime ganache is runing. Used for wiring up the default accounts correctly. Defaults to 8545

Global Objects

We've augmented the test runner with the following things you can use:

- In your unit tests you can use the global `accounts` object. It contains the secretKey (private key) and instance of ethers.Wallet of the account.

- The assert object has `assert.revert(promiseOfFailingTransaction)` function for testing reverting transactions

Available Utils

On your disposal there is a global available utils object. Here are the methods it exposes:

- `utils.timeTravel(provider, seconds)` method allowing etherlime ganache to move `seconds` ahead. You need to pass your provider from the EtherlimeGanacheDeployer
- `utils.setTimeTo(provider, timestamp)` method allowing etherlime ganache to move to the desired `timestamp` ahead. You need to pass your provider from the EtherlimeGanacheDeployer
- `utils.mineBlock(provider)` method telling the etherlime ganache to mine the next block. You need to pass your provider from the EtherlimeGanacheDeployer
- `utils.hasEvent(receipt, contract, eventName)` allowing the user to check if the desired event was broadcasted in the transaction receipt. You need to pass the Transaction receipt, the contract that emits it and the name of the Event.
- `utils.parseLogs(receipt, contract, eventName)` allowing the user get parsed events from a transaction receipt. You need to pass the Transaction receipt, the contract that emits it and the name of the Event. Always returns an event.

Examples

General Example

```
const etherlime = require('etherlime');
const Billboard = require('../build/Billboard.json');

describe('Example', () => {
        let owner = accounts[3];
        let deployer;

        beforeEach(async () => {
                deployer = new etherlime.EtherlimeGanacheDeployer(owner.secretKey);
        });

        it('should set correct owner', async () => {
                const BillboardContract = await deployer.deploy(Billboard, {});
                let _owner = await BillboardContract.owner();

                assert.strictEqual(_owner, owner.signer.address, 'Initial contract␣
↪owner does not match');
        });
});
```

execute function from another account

```
const etherlime = require('etherlime');
const ethers = require('ethers');
const Billboard = require('../build/Billboard.json');
```

(continues on next page)

(continued from previous page)

```
describe('Example', () => {
        let aliceAccount = accounts[3];
        let deployer;

        beforeEach(async () => {
                deployer = new etherlime.EtherlimeGanacheDeployer(aliceAccount.
↪secretKey);
                const BillboardContract = await deployer.deploy(Billboard, {});
        });

        it('should execute function from another account', async () => {
                let bobsAccount = accounts[4].signer;
                const transaction = await BillboardContract
                        .from(bobsAccount /* Could be address or just index in␣
↪accounts like 3 */)
                        .buy('Billboard slogan', { value: ONE_ETHER });
                assert.equal(transaction.from, bobsAccount.address);
        });
});
```

accounts

```
const Billboard = require('../build/Billboard.json');
const etherlime = require('etherlime');

describe('Billboard', () => {
        let owner = accounts[5];

        it('should initialize contract with correct values', async () => {
                const deployer = new etherlime.EtherlimeGanacheDeployer(owner.
↪secretKey);
                const BillboardContract = await deployer.deploy(Billboard, {});

                // Do something with the contract
        });
});
```

assert.revert

```
it('should throw if throwing method is called', async () => {
        assert.revert(contract.throwingMethod());
});
```

Check if the desired event was broadcasted in the transaction receipt

```
const etherlime = require('etherlime');
const Billboard = require('../build/Billboard.json');
const assert = require('chai').assert;
```

(continues on next page)

(continued from previous page)

```
describe('Billboard', () => {
    let owner = accounts[5];

    it('should emit event', async () => {
        const deployer = new etherlime.EtherlimeGanacheDeployer(owner.secretKey);
        const BillboardContract = await deployer.deploy(Billboard, {});

        const buyTransaction = await BillboardContract.buy('Billboard slogan', {␣
↪value: 10000 });

        const transactionReceipt = await BillboardContract.
↪verboseWaitForTransaction(buyTransaction);

        const expectedEvent = 'LogBillboardBought';

        assert.isDefined(transactionReceipt.events.find(emittedEvent => emittedEvent.
↪event === expectedEvent, 'There is no such event'));
    });
});
```

etherlime coverage

Syntax

```
etherlime coverage [path] [timeout] [port] [runs] [solcVersion] [buildDirectory]␣
↪[workingDirectory] [shouldOpenCoverage]
```

Parameters:

- `path` - [Optional] By specifying `path` you can set a path to a selected directory or you can set the path directly to the javascript file which contains your tests. By default the `path` points to `./test`.
- `timeout` - [Optional] This parameter defines the test timeout in milliseconds. Defaults to 2000 ms.
- `port` - [Optional] The port to run the solidity coverage testrpc (compatible with etherlime ganache deployer). Default: 8545.
- `runs` - [Optional] By specifying number runs you can enable the optimizer of the compiler with the provided number of optimization runs to be executed. Compilation is always performed by solidity coverage.
- `solcVersion` - [Optional] By specifying `solcVersion` you can choose a specific solc version to be used for compilation and coverage reports.
- `buildDirectory` - [Optional] By specifying `buildDirectory` you can choose which folder to use for reading builded contracts from, instead of the default one: `./build`.
- `workingDirectory` - [Optional] By specifying `workingDirectory` you can choose which folder to use for reading contracts from, instead of the default one: `./contracts`.
- `html` - [Optional] By specifying `html` you can choose either to open automatically with you default browser the html coverage report located in: `./coverage`. Defaults to `false`.

etherlime debug

In order to debug transaction, you will need the following:

- The transaction hash of a transaction on your desired blockchain.
- The source code of contract that transaction is executed from.

Syntax

```
etherlime debug <txHash> [port]
```

Parameters:

- `txHash` - Transaction hash of the transaction on your desired blockchain.
- `port` - [Optional] The port that the etherlime ganache is runing. Defaults to 8545.

Using the command will start the debugger interface with the following information:

- List of addresses involved or created during the cycle of the transaction passed in.
- List of available commands for using the debugger.
- The entry point of the transaction, including code preview and the source file.

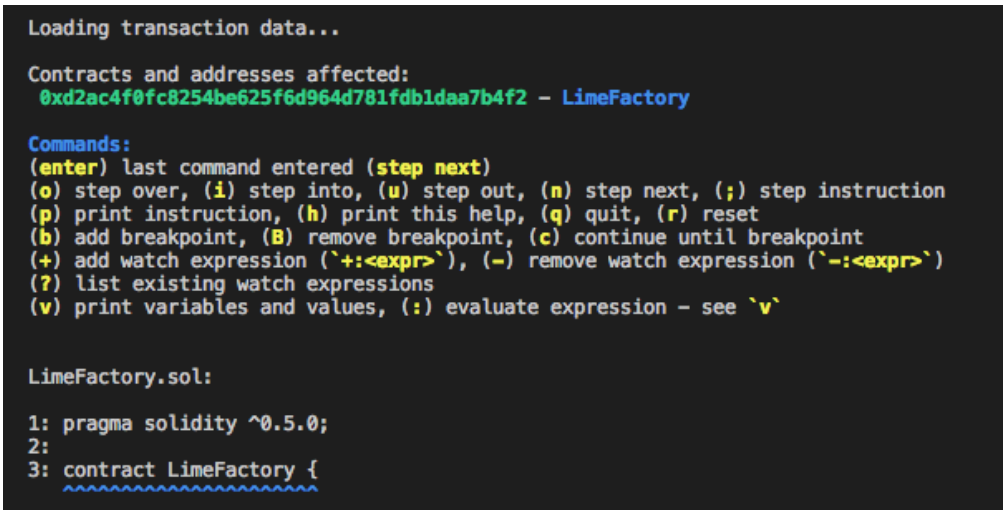
Available Commands

The enter key is sending to the debugger the last command that is entered. After the initial start of the debugger, the enter key is set to step to the next logical source code element (the next statement or expression that is evaluated by the EVM). You can use n or enter initially.

- `(o) step over` Steps over the current line, relative to the position of the statement or expression currently being evaluated in the Solidity source file. Use this command if you don't want to step into a function call or contract creation on the current line, or if you'd like to quickly jump to a specific point in the source file
- `(i) step into` Steps into the function call or contract creation currently being evaluated. Use this command to jump into the function and quickly start debugging the code that exists there.
- `(u) step out` Steps out of the currently running function. Use this command to quickly get back to the calling function, or end execution of the transaction if this was the entry point of the transaction.
- `(n) step next` Dteps to the next logical statement or expression in the source code. For example, evaluating sub expressions will need to occur first before the virtual machine can evaluate the full expression. Use this command if you'd like to analyze each logical item the virtual machine evaluates.
- `(;) step instruction` Allows you to step through each individual instruction evaluated by the virtual machine. This is useful if you're interested in understanding the low level bytecode created by the Solidity source code. When you use this command, the debugger will also print out the stack data at the time the instruction was evaluated.
- `(p) print instruction` Prints the current instruction and stack data, but does not step to the next instruction. Use this when you'd like to see the current instruction and stack data after navigating through the transaction with the logical commands described above.
- `(h) print help` Print the list of available commands.
- `(q) quit` Print the list of available commands.
- `(r) reset` Reset the debugger to the beginning of the transaction.
- `(b) add a breakpoint` Set breakpoints for any line in any of your source files (see examples below). These can be given by line number; by relative line number; by line number in a specified source file; or one may simply add a breakpoint at the current point in the code.

- `(B) remove a breakpoint` Remove any of your existing breakpoints.
- `(B all) remove all breakpoints` Remove all of your existing breakpoints.
- `(c) continue to breakpoint` Cause execution of the code to continue until the next breakpoint is reached or the last line is executed.
- `(+:) add watch expression` Add a watch on a provided expression, for example: `+:limes`
- `(-:) remove watch expression` Remove a watch on a provided expression, for example: `-:limes`
- `(?) list existing watch expressions` Display a list all the current watch expressions.
- `(v) display variables` Display the current variables and their values.

Here is example of runned debugger with txHash:



---

etherlime shape

Syntax

```
etherlime shape [name]
```

Parameters:

- `name` - Specifies the name of the framework or library that the project will be build up. Choices: `angular` - shapes boilerplate containing ready to use dApp with Angular front-end and Etherlime project. `react` - shapes boilerplate containing ready to use dApp with React front-end and Etherlime project.

References:

Follow up the steps to set up your project here:

- Angular: https://github.com/LimeChain/etherlime-shape-angular/blob/master/README.md
- React: https://github.com/LimeChain/etherlime-shape-react/blob/master/README.md
- Monoplasma Demo: https://github.com/LimeChain/etherlime-shape-monoplasma

Running this command will create integrated blockchain project with all modules and settings needed.

etherlime flatten

Syntax

```
etherlime flatten [file] [solcVersion]
```

Parameters:

- `file` - The name of the contract from "./contract" folder that you want to be flattened.
- `solcVersion` - [Optional] By specifying `solcVersion` you can set the version of the solc which will be used for compiling the smart contracts. By default it uses the solc version from your node_modules or the default one from etherlime.

Running this command will flatten the given smart contract and will record all Solidity code in one file along with imported sources. It will create "./flat" folder where you can find the flattened contract.

etherlime ide

Syntax

```
etherlime ide [port]
```

Parameters:

- `port` - [Optional] By specifying `--port` you can specify port your ganache is running on. Default: 8545

Running this command will run web-based Solidity IDE that works with the file system. It will allow you to easily edit, compile, deploy and interact with your smart contracts. You must have a running ganache to start the IDE with loaded accounts.

etherlime zk

In order to start a project with Zero Knowledge Proof, please refer to etherlime init command.

Available Commands:

Circuit Compilation

- `etherlime zk compile` Running this command will compile a circuit file located in `zero-knowledge-proof/circuits` and generates a new folder `compiled-circuits`.

Establish Trusted Setup

- `etherlime zk setup` Running this command will establish a trusted setup based on compiled circuit and generates a folder `trusted setup` with `proving_key` and `verification_key`. The command reads the compiled circuit from `zero-knowledge-proof/compiled-circuits`.

Generate ZK Proof

- `etherlime zk proof [signal] [circuit] [provingKey]` Running this command will generates a proof based on compiled circuit, public signal input and proving key. A new folder `generated-proof` is generated with `proof` and `public_signals`. This proof can be used for off-chain Zero-Knowledge-Proof verification.

  Parameters:

  - `signal` - [Optional] Specifies the file with public signals input to be used for generating a proof. Defaults to `input.json` read from `zero-knowledge-proof/input` folder.
  - `circuit` - [Optional] Specifies the compiled circuit for checking of matched signals. Defaults to: `circuit.json` read from `zero-knowledge-proof/compiled-circuits` folder.
  - `provingKey` - [Optional] Specifies the prooving key to be used for generating a proof. Defaults to: `circuit_proving_key.json` read from `zero-knowledge-proof/trusted-setup` folder.

Verify Proof (Off-chain)

- `etherlime zk verify [publicSignals] [proof] [verifierKey]` Running this command will generates a verifier based on public signals file that comes out of the proof command, the proof itself and verifier key. A new folder `verified-proof` is generated with `output.json` file.

  Parameters:

  - `publicSignals` - [Optional] Specifies the file with signals to be used for generating verifying a proof. Defaults to `circuit_public_signals.json` read from `zero-knowledge-proof/generated-proof` folder.
  - `proof` - [Optional] Specifies the compiled proof that would be used for generating a proof based on it. Defaults to: `circuit_proof.json` read from `zero-knowledge-proof/generated-proof` folder.
  - `verifierKey` - [Optional] Specifies the verifier key to be used for generating a proof. Defaults to: `circuit_verification_key.json` read from `zero-knowledge-proof/trusted-setup` folder.

  `output.json` file has two params:

  - `verified` - whatever the proof is verified or not
  - `timestamp` - identifier for the time that event occurs

Generate Smart Contract for On-Chain Verification

- `etherlime zk generate [verifierKey]` Generates a verifier smart contract based on verification key which can be used for on-chain verification. The smart contract is written in contracts folder and it is ready to be compiled and deployed with `etherlime compile` and `etherlime deploy`. The verifier smart contract has a public view method `verifyProof` that can be called for on-chain verification. You can generate the call parameters with `etherlime zk-generate-call` cli command.

  Parameters:

  - `verifierKey` - [Optional] Specifies the verifier key to be used for generating a verifier smart contract. Defaults to: circuit_verification_key.json read from `zero-knowledge-proof/generated-proof` folder.

Generate output call based for On-chanin Verification

- `etherlime zk call [publicSignals] [proof]` Running this command will generates a call based on proof and public signals. A new folder `generated-call` is generated with `generatedCall.json` file. This generated call can be used for on-chain verification, for calling public view method `verifyProof` of the generated verifier contract with this data.

  Parameters:

  - `publicSignals` - [Optional] Specifies the file with signals to be used for generating verifying a proof. Defaults to `circuit_public_signals.json` read from `zero-knowledge-proof/generated-proof` folder.
  - `proof` - [Optional] Specifies the compiled proof that would be used for generating a proof based on it. Defaults to: `circuit_proof.json` read from `zero-knowledge-proof/generated-proof` folder.

1.2.4 Migration from Truffle to Etherlime

Install & Initialize Etherlime

```
npm i -g etherlime
```

Install the global etherlime to allow you to run `etherlime` commands.

```
etherlime init
```

The command will add to your project structure the following parts:

- ./contracts/LimeFactory.sol
- ./deployment/deploy.js
- ./test/exampleTest.js

Note! These are added just to give you an example. You can remove them.

Write new scripts for deployment using the template provided

- **require etherlime module**
- **require the compiled contract** from ./build folder not the contract itself

with Truffle

```
const LimeFactory = artifacts.require("./LimeFactory.sol");
```

with Etherlime

```
const etherlime = require('etherlime')
const LimeFactory = require('../build/LimeFactory.json');
```

- **set the deployer and then deploy the contract**

Local deployment with Etherlime

```
const etherlime = require('etherlime')
const LimeFactory = require('../build/LimeFactory.json');
const InterfaceFactory = require('../build/InterfaceFactory.json')

const deployer = new etherlime.EtherlimeGanacheDeployer();
const limeFactory = await deployer.deploy(LimeFactory);

//example how to wrap deployed contract and to pass its address
const contractInstance = await etherlime.ContractAt(InterfaceFactory, limeFactory.
↪contractAddress)
```

Find more examples for deployment here.

Modify tests

In order to modify the tests from Truffle to Etherlime, slight changes are needed to be done:

with Truffle

```
const LimeFactory = artifacts.require("./LimeFactory.sol");

contract('LimeFactory tests', async (accounts) => {

    let owner = accounts[0];

    beforeEach(async function() {
        limeFactory = await LimeFactory.new();
    });

    it('should do something', () => {

    })
}
```

with Etherlime

```
// step1: require Etherlime module
const etherlime = require('etherlime')

// step2: require compiled contract from ./build not the .sol file (as in deployment␣
↪scripts)
const LimeFactory = require('../build/LimeFactory.json')


// step4: replace 'contract' descriptor to 'describe' then remove (accounts) param in␣
↪async function
describe('LimeFactory tests', async () => {

    // step5: initialize account
    let owner = accounts[0];

    // step6: set the deployer in before/beforeEach and fix the deployment scripts as␣
↪we did before
```

(continues on next page)

(continued from previous page)

```
    beforeEach(async function() {

        deployer = new etherlime.EtherlimeGanacheDeployer(owner.secretKey);
        limeFactory = await deployer.deploy(LimeFactory);

    });

    it('should do something', () => {

    })
})
```

Flexibility

- **in case you want to use an address of an account, you must extend it to** `let owner = accounts[0].signer.address`
- **when a contract's method is called, the default sender is set to accounts[0]. If you want to execute it from another account, replace** `{from: anotherAccount}` object with `.from(anotherAccount).`

with Truffle

```
await limeFactory.createLime(newLime' 0, 10, 12, {from: accounts[1]})
```

with Etherlime

```
await limeFactory.from(2).createLime('newLime' 0, 10, 12);

// as a param you may also use:
await limeFactory.from(accounts[1]).createLime('newLime' 0, 10, 12);
await limeFactory.from(accounts[1].signer).createLime('newLime' 0, 10, 12);
await limeFactory.from(accounts[1].signer.address).createLime('newLime' 0, 10, 12);
await limeFactory.from(customSigner).createLime('newLime' 0, 10, 12);
```

- **when you need to execute payable function, pass the value as an object** `contract.somePayableFunction(arg1, arg2, {value: 100})`
- **don't use ".call" when calling view functions.**
- **to timeTravel - replace web3 increaseTime with global options** `utils.timeTravel(provider, seconds)`

Assertions and available utils

For more convenience Etherlime provides some additional assertions and global utils object:

assert it is an address

```
it('should be valid address', async () => {
    assert.isAddress(limeFactory.contractAddress, "The contract was not deployed");
})
```

assert a function revert

```
it('should revert if try to create lime with 0 carbohydrates', async () => {
    let carbohydrates = 0;
    await assert.revert(limeFactoryInstance.createLime("newLime2", carbohydrates, 8,␣
↪2), "Carbohydrates are not set to 0");
});
```

test an event

with Truffle:

```
let expectedEvent = 'FreshLime';
let result = await limeFactory.createLime('newLime' 8, 10, 12);
assert.lengthOf(result.logs, 1, "There should be 1 event emitted from new product!");
assert.strictEqual(result.logs[0].event, expectedEvent, `The event emitted was $
↪{result.logs[0].event} instead of ${expectedEvent}`);
```

with Etherlime

```
let expectedEvent = 'FreshLime'
let transaction = await limeFactory.createLime('newLime' 8, 10, 12);
const transactionReceipt = await limeFactory.verboseWaitForTransaction(transaction)

// check the transaction has such an event
let isEmitted = utils.hasEvent(transactionReceipt, LimeFactory, expectedEvent);
assert(isEmitted, 'Event FreshLime was not emitted');

// parse logs
let logs = utils.parseLogs(transactionReceipt, LimeFactory, expectedEvent);
assert.equal(logs[0].name, 'newLime, "LimeFactory" with name "newLime" was not created
↪');
```

Find more test examples here.

Final steps:

- **delete** `./migrations` folder
- **delete** `truffle.js/truffle-config.js` file
- **delete** `truffle` from `package.json`
- **delete** `node_modules`
- **run** `npm install`
- **open a fresh terminal tab and enter** `etherlime ganache`
- **run** `etherlime test`

CHAPTER 2

License

Completely MIT Licensed. Including ALL dependencies.