ride Documentation

Release 1.0.0

© 2022, Jourlez

January 12, 2023

Chapter 1

User guide

1.1 User Requirements

1.1.1 Install Ansible

Ansible will take care of setting your development environment, but for that you need to install Ansible first.

Fedora

```
sudo dnf install ansible
```

RHEL

```
sudo yum install ansible
```

CentOS

```
sudo yum install epel-release
sudo yum install ansible
```

Ubuntu

```
sudo apt install ansible
```

Pip

```
sudo pip3 install ansible
```

1.1.2 Install Ansible Roles

```
ansible-galaxy install -r ./ansible/requirements.yml
```

1.1.3 Run Ansible Playbook

```
ansible-playbook ./ansible/install.yml --ask-become-pass
```

At this point Ansible will take a few minutes to set everything up so please be patient.

1.1.4 Verify Private Node and Explorer

You can verify that the node is running through REST API at http://localhost:6869.

On top of that you can also see an empty chain of you local node rapidly being built using your instance of the Explorer at http://localhost:3000.

1.2 How to Run the Tests

1. The command to run tests is the following:

```
surfboard test
```

2. If you have several scenarios (for instance, you need a separate deployment script), you can run a particular scenario by running

```
surfboard test my-scenario.js
```

Surfboard will pick up all the tests files in ./test/ folder and run the scenario over a node through the endpoint configured in surfboard.config.json . After several seconds, you will see the test results.

3. There are several parameters you can use while testing:

- `-v`, `--verbose` logs all transactions and node responses
- `--env=`*env* which environment should be used for test
- `--variables=`*variables*

  env variables can be set for usage in tests via env.{variable_name}. E.g.: MY_SEED="seed phraze",DAPP_ADDRESS=xyz, AMOUNT=1000

1.3 Examples

1.3.1 Deposit and Withdraw

This example will guide you through all the necessary steps for developing a simple decentralised application(dApp) in which anyone can deposit as many tokens to the dApp as they want, but they can only withdraw their own waves.

Navigate to the ./ride/examples/01/ folder which is where your Ride Project will be located in. Then run the following command:

```
surfboard init
```

If you get a warning saying the directory is not empty continue anyways by selecting yes, this part is very important. The above will produce an output similar to this one:

```
~/ride/examples/01$ surfboard init
?
```

The seed and address on the previous message are there because while creating you new Ride project, Surfboard also created a new wallet for you on the Testnet node of the Blockchain. Make sure to save them both since you will require them for the next steps.

Source Code

```
.
```

```
├── surfboard.config.json
├── ride
│   └── wallet.ride
├── scripts
│   └── wallet.deploy.js
└── test
    └── wallet.ride-test.js
```

- surfboard.config.json is the configuration file for your Ride project.
- ride/ contains all of your Ride Scripts(Smart Contracts/dApps).
- test/ has all your test files, in this case powered by javascript's Mocha framework.
- scripts/ holds scripts used for deploying your smart contracts or dApps.

surfboard.config.json

```
{
    "ride_directory": "ride",
    "test_directory": "test",
    "envs": {
        "custom": {
            "API_BASE": "http://localhost:6869/",
            "CHAIN_ID": "R",
            "SEED": "waves private node seed with waves tokens",
            "timeout": 60000
        },
        "testnet": {
            "API_BASE": "https://nodes-testnet.wavesnodes.com/",
            "CHAIN_ID": "T",
            "SEED": "chapter position silver stage later verify account organ ride
bronze emotion scissors",
            "timeout": 60000
        }
    },
    "defaultEnv": "custom",
    "mocha": {
        "timeout": 60000
    }
}
```

The envs (environment) entries which are: custom and testnet are important since these are used to set the default deployment environment for your dApps. Each environment is configured by:

- API_BASE: The REST API endpoint of the node that will be used for running a dApp as well as the CHAIN_ID of the network.
- SEED: The seed phrase for account with tokens, which will be the source of all the tokens in your test. As you noticed, the SEED for the testnet entry matches the seed of the account Surfboard automatically created for you.

wallet.ride

```
# In this example multiple accounts can deposit their funds and safely take them
back.
# User balances are stored in the dApp state as mapping `address=>waves`.

{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}

@Callable(i)
func deposit() = {
   # deposit function can be invoked from UI by user to top up the balance
```

```
   let pmt = extract(i.payment)  # creating variable with all data about a payment
attached to the invokation
   if (isDefined(pmt.assetId)) then throw("can hodl waves only at the moment")  #
waves-only threshold
   else {
        let currentKey = toBase58String(i.caller.bytes)  # determining caller key
        let currentAmount = match getInteger(this, currentKey) {  # reading
current user's balance from the account state
            case a:Int => a
            case _ => 0  # taking zero as a balance value if this is the first
time user deposits money
        }
        let newAmount = currentAmount + pmt.amount  # counting new balance as "old
balance + payment value"
        WriteSet([DataEntry(currentKey, newAmount)])  # updating user's balance in
the account state
   }
}

@Callable(i)
func withdraw(amount: Int) = {
    # withdraw function can be invoked by user to "cash out" a part of his balance
    let currentKey = toBase58String(i.caller.bytes)
    let currentAmount = match getInteger(this, currentKey) {
        case a:Int => a
        case _ => 0
    }
    let newAmount = currentAmount - amount
    if (amount < 0)  # is user requesting a positive amount?
            then throw("Can't withdraw negative amount")
    else if (newAmount < 0)  # does user have enough balance for this withdraw?
            then throw("Not enough balance")
            else ScriptResult(
                    WriteSet([DataEntry(currentKey, newAmount)]),  # saving new
balance to the account state
                    TransferSet([ScriptTransfer(i.caller, amount, unit)])  #
transfering the withdraw amount to user
                )
    }

@Verifier(tx)
func verify() = false  # this script can NOT be updated because of this verifier
function
```

There are two @Callable functions available for invocation via InvokeScriptTransaction.

- deposit(), which requires attached payment.
- withdraw(amount: Int), which transfers tokens back to the caller upon successful execution.

Throughout the dApp lifecycle, there will be a mapping(address → amount) maintained:

Table 1.1. Mapping(address → amount)

Action	Resulting State
<initial>	<empty>
Alice deposits 5 Waves	<alice-address> → 500000000
Bob deposits 2 Waves	<alice-address> → 500000000; <bob-address> → 200000000
Bob withdraws 7 Waves	<DENIED!>
Alice withdraws 4 Waves	<alice-address> → 100000000; <bob-address> → 200000000

wallet.deploy.js

```
// Wallet.ride deploy script. To run execute `surfboard run path/to/script`

// wrap out script with async function to use fancy async/await syntax
(async () => {
    // Functions, available in tests, also available here
    const script = compile(file('wallet.ride'));

    // You can set env varibles via cli arguments. E.g.: `surfboard run
path/to/script  --variables 'dappSeed=seed phrase,secondVariable=200'`
    const dappSeed = env.dappSeed;
    if (dappSeed == null){
        throw new Error(`Please provide dappSedd`)
    }
    //const dappSeed = env.SEED; // Or use seed phrase from surfboard.config.json
    const ssTx = setScript({
        script,
        // additionalFee: 400000 // Uncomment to raise fee in case of redeployment
    }, dappSeed);
    await broadcast(ssTx);
    await waitForTx(ssTx.id);
    console.log(ssTx.id);
})();
```

wallet.ride-test.js

```
const wvs = 10 ** 8;

describe('wallet test suite', async function () {

    this.timeout(100000);

    before(async function () {
        await setupAccounts(
            {foofoofoofoofoofoofoofoofoofoofoo: 10 * wvs,
                 barbarbarbarbarbarbarbarbarbar: 2 * wvs,
                  wallet: 0.05 * wvs});
        const script = compile(file('wallet.ride'));
        const ssTx = setScript({script}, accounts.wallet);
        await broadcast(ssTx);
        await waitForTx(ssTx.id)
        console.log('Script has been set')
    });

    it('Can deposit', async function () {

        const iTxFoo = invokeScript({
            dApp: address(accounts.wallet),
            call: {function: "deposit"},
            payment: [{assetId: null, amount: 0.9 * wvs}]
        }, accounts.foofoofoofoofoofoofoofoofoofoofoo);


        const iTxBar = invokeScript({
            dApp: address(accounts.wallet),
            call: {function: "deposit"},
            payment: [{assetId: null, amount: 1.9 * wvs}]
        }, accounts.barbarbarbarbarbarbarbarbar)


        await broadcast(iTxFoo);
        await broadcast(iTxBar);
```

```
        await waitForTx(iTxFoo.id);
        await waitForTx(iTxBar.id);
    })

    it('Cannot withdraw more than was deposited', async function () {
        const iTxFoo = invokeScript({
            dApp: address(accounts.wallet),
            call: {
                function: "withdraw",
                args: [{type:'integer', value: 2 * wvs}]
            },

        }, accounts.foofoofoofoofoofoofoofoofoofoofoo);

        expect(broadcast(iTxFoo)).to.be.rejectedWith("Not enough balance")
    })

    it('Can withdraw', async function () {
        const iTxFoo = invokeScript({
            dApp: address(accounts.wallet),
            call: {
                function: "withdraw",
                args: [{ type: 'integer', value: 0.9 * wvs }]
            },

        }, accounts.foofoofoofoofoofoofoofoofoofoofoo);
        await broadcast(iTxFoo)
    })
})
```

There's a before function and three tests. Let's review them:

- The before function funds a few accounts via MassTransferTransaction using setupAccounts (foo, bar and wallet), compiles the script and creates a set-script transaction of the compiled script which assigns it to the account under the name of wallet and broadcasted to the blockchain and then mined into a block via waitForTx.
- The "Can deposit" test verifies correct deposits are being processed successfully; signs and broadcasts an InvokeScriptTransaction invoking deposit() function with an invokescript, for two accounts that were setup in the before function which are foo and bar. Check out the invokescript parameters
- The "Can't withdraw more than was deposited" test verifies that no-one can "steal" another account's tokens
- The "Can withdraw" test verifies that correct withdrawals are being processed successfully.

Tests

When you run the tests Surfboard will pick up all the test files inside the ./test/ folder and run them over a node through the endpoint configured in surfboard.config.json. After several seconds, you will see an output like this one:

```
~/ride/examples/01$ surfboard test
Starting test with "custom" environment
Root address: 3M4qwDomRabJKLZxuXhwfqLApQkU592nWxF

 wallet test suite
Generating accounts with nonce: d1ef4b3
Account generated: foofoofoofoofoofoofoofoofoofoofoo#d1ef4b3 -
3M9AKTNRKmFkwGQmXiTUZz9cE8haU3mNdVP
Account generated: barbarbarbarbarbarbarbarbarbar#d1ef4b3 -
3M28pYYfjiUXFWvmHwdQnbgLYP2jkDDF3iL
Account generated: wallet#d1ef4b3 - 3MBAK4oVXxWStk6JQJwcLK9riXhwFdujksw
```

```
Accounts successfully funded

Script has been set
    ?
```

Now you can look more closely at what happened using your Explorer instance by just browsing blocks or pasting one of the addresses above to the search box. Remember to check the Blockchain Network to make sure you have selected the correct one(Custom in this case).

To test against a different environment like testnet, just use the following command:

```
surfboard test --env=testnet
```

If you want to explore what information is sent to and from the nodes during the calls of broadcast(…) in a test, run a test with -v (stands for "verbose") to make Surfboard produce extensive logs:

```
surfboard test -v
```

Deployment

You can also deploy your dApp to the wallet created for you by the surfboard init command. To do that, run the command below:

```
surfboard run ./scripts/wallet.deploy.js --env=testnet --variables dappSeed="dApp
seed"
```

When you run the above command, you will get an error(which is expected) because as you already know in order to deploy a dApp you would need to fund the wallet that will be scripted. You can request funding for your account using the faucet and typing the address of the account Surfboard generated for you.

Now that your account has funds run the command once more and it will output the transaction id associated with the deployment carried out by the wallet.ride-test.js file which you can look up using the Explorer instance. The output will look something like this:

```
~/ride/examples/01$ surfboard run ./scripts/wallet.deploy.js --env=testnet
--variables dappSeed="dApp seed"
6TBUs1ZSTDB8LZeDE8jGTwhGMLCvPKe8pBM1FWnHzYe7
```

Keeper Integration

If you want to go the extra mile this is your chance: you can integrate the project you have been working on with Keeper. Go to your browser of preference and open the official website for the tool and follow the installation instructions based on the type of browser you're using.

Open Keeper and click Get Started, choose a password and then Continue. On the bottom left instead of Mainnet select Custom as the Blockchain Network and set the Node Address to the one below which as you will notice is the address you assigned to your private node, then Save and Apply.

```
http://localhost:6869
```

Import a seed with tokens for the Blockchain Network. For simplicity, just use the staking seed of your node:

```
waves private node seed with waves tokens
```

Choose a name for the account, any name will do. Then start an instance of a 1-page serverless web appliction using npm with the following command:

```
http-server-node --port 5050 --entryPoint
../../docs/_static/01_ride/files/02_index.html
```

Check the instance you just started at:

```
http://localhost:5050
```

Enter the account address of the account generated for you by the command surfboard test under the name of wallet during testing into the script address blank text field.

Type an amount and select Deposit which will cause Keeper to request your permission to sign an InvokeScriptTransaction with an attached payment of the amount you typed.

Approve the transaction so it will be created and broadcasted to the network. You can use the transaction "View transaction" button at the bottom to get the transaction id and that way you can look up all the details using the Explorer instance. Remember to check the Blockchain Network to make sure you have selected the correct one(Custom in this case).

Note You could also use a service that allows you to automatically generate an interface for a dApp using the names and data of the arguments of your callable functions. Just sign in using Keeper and again enter the account address of the account generated for you by the command surfboard test under the name of wallet during testing into the smart contract blank text field.

Now you have a good foundation to start developing using Ride, feel free to modify this example as you like and try to play a little bit more with it so you can learn more on your own.

Developer Guide/Contributing

2.1 Developer Requirements

Follow the same process as specified in Section 1.1.

2.2 Project Structure

```
.
├── ansible/
│   ├── install.yml
│   ├── requirements.yml
│   ├── group_vars/
│   │   └── all.yml
│   └── roles/
│       ├── deploy-node-explorer/tasks/
│       │   ├── deploy-node-explorer
│       │   └── main.yml
│       └── update-system/tasks/
│           ├── main.yml
│           └── update-system.yml
├── docs/
│   ├── buildDocs.sh
│   ├── conf.py
│   ├── index.rst
│   ├── make.bat
│   ├── Makefile
│   ├── _build/
│   ├── _static/
│   ├── _templates/
│   └── locales/
└── examples/
    └── 01
```

ansible folder:

- install.yml is the main playbook in charge of running all the roles used for project setup.
- requirements.yml contains all the required roles for the main playbook to work properly.
- group_vars/all.yml has all the variables used in the main playbook.
- roles/ is the folder containing each and every one of the roles that will be used by the main playbook.

docs folder:

- buildDocs.sh is your build script which will be executed in a docker container on GitHub's cloud.
- conf.py is a Python script holding the configuration of the Sphinx project. It contains the project name and release you specified to sphinx-quickstart, as well as some extra configuration keys.
- index.rst is the root document of the project, which serves as welcome page and contains the root of the "table of contents tree" (or toctree).
- make.bat and Makefile are convenience scripts to simplify some common Sphinx operations, such as rendering the content.
- _static/ will contain custom stylesheets and other static files like images.
- _templates/ will contain custom html template files overriding jinja templates.
- locales/ will contain all the translated versions of the documentation.

2.3 Document Generation

2.3.1 HTML

```
sphinx-build -b html docs/ docs/_build/html
```

2.3.2 PDF

```
sphinx-build -b rinoh docs/ docs/_build/rinoh -D language="en"
```

2.3.3 EPUB

```
sphinx-build -b epub docs/ docs/_build/epub -D language="en"
```

2.4 Internationalization (i18n)

Run the command below from the 'docs/' directory. This tells sphinx to parse your reST files and automatically find a bunch of strings-to-be-translated and give them a unique msgid that will be used during translation.

```
make gettext
```

After this, .pot files will be generated for you. You can verify it using the command below.

```
ls _build/gettext/
```

Next, let's tell sphinx to prepare some Spanish destination-language '.po' files from your above-generated source-lananguage '.pot' files. Execute the following command to prepare your Spanish-specific translation files.

```
sphinx-intl update -p _build/gettext -l es
```

The above execution create '.po' files: one for each of your '.pot' source-language files, which correlate directly to each of your two '.rst' files. For example, for the new Spanish-specific 'docs/locales/es/LC_MESSAGES/index.po' file, as you see it has the same contents as the source '.pot' file.

These language-specific '.po' files are where you actually do the translating. If you're a large project, then you'd probably want to use a special program or service to translate these files.

After translating everything you can build your html static content in the language you want, for example Spanish:

```
sphinx-build -b html . _build/html/en -D language='es'
```