hs-web3 Documentation

Release 0.8.0.0

Alexander Krupenkin

Apr 23, 2023

User documentation

1 Getting started **3**
- 1.1 Installation . . . 3
- 1.2 Quick start . . . 3

2 Ethereum node API **5**
- 2.1 Providers . . . 5
- 2.2 API Reference . . . 6

3 Polkadot node API **7**
- 3.1 API Reference . . . 7

4 Polkadot Storage **9**

5 Polkadot Extrinsic **11**

6 Ethereum accounts **13**
- 6.1 Account managing . . . 13
- 6.2 Transaction sending . . . 14
- 6.3 Safe transactions . . . 15

7 Smart contracts **17**
- 7.1 Contract ABI . . . 17
- 7.2 ABI encoding . . . 18
- 7.3 Contract deployment . . . 18

8 Ipfs Client API **21**
- 8.1 Api Type . . . 21
- 8.2 Monad Runner . . . 21
- 8.3 Call Functions . . . 22

9 Ethereum Name Service **23**

10 Contributing **25**
- 10.1 Did you find a bug? . . . 25
- 10.2 Did you write a patch that fixes a bug? . . . 25

11 Testing **27**

hs-web3 is a Haskell library for interacting with Ethereum. It implements Generic JSON-RPC client for most popular Ethereum nodes: parity-ethereum and go-ethereum.

CHAPTER 1

Getting started

Note: **hs-web3** is a Haskell library. Of course you should have some knowledge about Haskell and platform tools like a *cabal* or *stack*. If you have not - Real World Haskell and Learn You a Haskell for Great Good is a good point to begin.

1.1 Installation

Simplest way is using Stackage with Nix integration.

```
stack install web3 --nix
```

Dependencies for building from source without Nix:

- zlib
- optional: solidity

1.2 Quick start

Lets import library entrypoint modules using `ghci`:

```
> import Network.Ethereum.Web3
> import qualified Network.Ethereum.Api.Eth as Eth
```

Note: I recomend to import *Network.Ethereun.Api.Eth* as **qualified**, because it has name similar to their prefix in JSON-RPC API.

Looks anything in `Eth` API:

```
> :t Eth.blockNumber
Eth.blockNumber :: JsonRpc m => m Quantity
```

To run it use `runWeb3` function:

```
> :t runWeb3
runWeb3 :: MonadIO m => Web3 a -> m (Either Web3Error a)

> runWeb3 Eth.blockNumber
Right 6601059
```

Note: Function `runWeb3` run default provider at **http://localhost:8545**, for using custom providers try to use `runWeb3'`.

CHAPTER 2

Ethereum node API

Any Ethereum node can export their Generic JSON-RPC. For connection with node **hs-web3** use internal tiny JSON-RPC client.

Note: Tiny client library placed at `Network.JsonRpc.TinyClient`. It exports special monad `JsonRpc` and function `remote` to define JSON-RPC methods. When developing tiny client I was inspired HaXR library.

2.1 Providers

To handle connection with Ethereum node some thing named **provider** is required. `Provider` data type define the endpoint of Ethereum node API. Module that export this type placed at `Network.Ethereum.Api.Provider`.

```
data Provider = HttpProvider String
```

Note: Currently **hs-web3** support HTTP(S) providers only.

Another interesting thing in this module is `Web3` type.

```
newtype Web3 a = ...
instance Monad Web3
instance JsonRpc Web3
```

As you can see `Web3` is monad that can handle JSON-RPC. It's very important because it can be used for any Ethereum node communication.

Note: `Web3` is a state monad with `JsonRpcClient` type as a state. This is mean that you can modify JSON-RPC server URI in runtime using MTL lenses for example.

Finally provider module exports `runWeb3` and party functions.

```
runWeb3' :: MonadIO m => Provider -> Web3 a -> m (Either Web3Error a)

runWeb3 :: MonadIO m => Web3 a -> m (Either Web3Error a)
runWeb3 = runWeb3' def
```

Note: Function `runWeb3` run default provider at **http://localhost:8545**.

Lets try to call custom Ethereum node URI with `runWeb3'` function using `ghci`.

```
> import Network.Ethereum.Api.Provider
> import qualified Network.Ethereum.Api.Eth as Eth
> runWeb3' (HttpProvider "http://localhost:9545") Eth.blockNumber
```

It can be useful to define function with custom Ethereum node endpoint location.

```
myNode :: Web3 a -> Either Web3Error a
myNode = runWeb3' (HttpProvider "http://my-host-name:8545")
```

2.2 API Reference

Currently implemented the following Ethereum APIs in modules:

Method prefix	Implementation
`eth_*`	Network.Ethereum.Api.Eth
`net_*`	Network.Ethereum.Api.Net
`web3_*`	Network.Ethereum.Api.Web3
`personal_*`	Network.Ethereum.Api.Personal

All modules use descriptive types according to official Ethereum specification. It placed at Network.Ethereum.Api.Types.

Note: See classic API reference at Hackage web3 page.

CHAPTER 3

Polkadot node API

As same as Ethereum nodes Polkadot node exports HTTP/WebSockets *JSON-RPC* API. For connection with node **hs-web3** use internal tiny JSON-RPC client.

Lets try to call Polkadot node with `runWeb3'` function using `ghci`.

```
> import Network.Web3.Provider
> import qualified Network.Polkadot.Api.System as System
> runWeb3' (WsProvider "127.0.0.1" 9944) $ System.name
Right "Parity Polkadot"
```

It can be useful to define function with Polkadot node endpoint location.

```
myNode :: Web3 a -> Either Web3Error a
myNode = runWeb3' (Wsprovider "127.0.0.1" 9944)
```

3.1 API Reference

Currently implemented the following Polkadot APIs in modules:

Method prefix	Implementation
`account_*`	Network.Polkadot.Api.Account
`author_*`	Network.Polkadot.Api.Author
`babe_*`	Network.Polkadot.Api.Babe
`chain_*`	Network.Polkadot.Api.Chain
`childstate_*`	Network.Polkadot.Api.Childstate
`contracts_*`	Network.Polkadot.Api.Contracts
`engine_*`	Network.Polkadot.Api.Engine
`grandpa_*`	Network.Polkadot.Api.Grandpa
`offchain_*`	Network.Polkadot.Api.Offchain
`payment_*`	Network.Polkadot.Api.Payment
`rpc_*`	Network.Polkadot.Api.Rpc
`state_*`	Network.Polkadot.Api.State
`system_*`	Network.Polkadot.Api.System

All modules use descriptive types located at Network.Polkadot.Api.Types.

Note: See classic API reference at Hackage web3 page.

CHAPTER 4

Polkadot Storage

Blockchains that are built with Substrate expose a remote procedure call (RPC) server that can be used to query runtime storage. In Haskell Web3 the standard Web3 provider could be used.

Lets try to query Polkadot storage with `runWeb3'` function using `ghci`.

```
> import Network.Web3.Provider
> import Network.Polkadot
> runWeb3' (WsProvider "127.0.0.1" 9944) (query "timestamp" "now" [] :: Web3 (Either␣
↪String Moment))
Right (Right 1610689972001)
```

The `query` function arguments is **section** (or module), **method** and list of arguments (for maps and double maps).

```
query :: (JsonRpc m, Decode a) => Text -> Text -> [Argument] -> m a
```

Where `a` type should be SCALE decodable.

Note: More usage details available in Polkadot example app.

CHAPTER 5

Polkadot Extrinsic

Extrinsic is a piece of data from external world that proposed to be a part of blockchain. Generally exist two kinds of extrinsics: `unsigned` (inherents) and `signed` (transactions).

Lets try to send Polkadot transaction with `runWeb3'` function using `ghci`.

```
> import Network.Web3.Provider
> import Network.Polkadot
```

The first, let's create new one account.

```
> me <- generate :: IO Ed25519
> multi_signer me
"5D7c97BufUFEqrGGn2nyw5HhgMTzQT2YkBZ33mojWwBijFLQ"
```

Where `Ed25519` generated and its Base58 encoded address printed out. I've use `multi_signer` wrapper because of `MultiAddress` format used in Polkadot.

The next, let's make a call structure that encodes function arguments and parameters required for Polkadot runtime dispatcher.

```
> let Right alice = from_ss58check "5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY"
> Right myCall <- runWeb3' (WsProvider "127.0.0.1" 9944) $ new_call "Balances"
↪"transfer" (MaId alice, Compact 200000000000000)
> myCall
Call 0 5 (MaId 0xd43593c715fdd31c61141abd04a99fd6822c8558854ccde39a5684e7a56da27d,␣
↪Compact 200000000000000)
```

Where `alice` is transfer destination account on chain, `from_ss58check` decodes it from Base58 and pack into `AccountId` type. It also should be wrapped in call arguments into `MultiAddress` type using `MaId` constructor. The `new_call` function gets module name, function name, arguments tuple and returns encodable structure for Polkadot runtime dispatcher.

Next step is signing the call and other extrinsic related staff like lifetime, nonce and etc. Fortunately, Haskell Web3 has `sign_and_send` function that makes it automatically.

```
> Right myTx <- runWeb3' (WsProvider "127.0.0.1" 9944) $ sign_and_send me myCall 0
> myTx
0x9034fb2c7e46b5de6e681565a657cefc32fb2aa93c21aad03acc20b79fb31e68
```

The `sign_and_send` function gets crypto pair to sign extrinsic, the call structure and tips amount (zero is acceptable in general case). If everything ok then you will get transaction hash as result.

Note: More usage details available in Polkadot example app.

CHAPTER 6

Ethereum accounts

Note: Ethereum whitepaper mention two types of accounts: smart contract and external owned account (EOA). But EOA only can send transaction to network. In this page EOA managing and generalized transaction sending is described.

hs-web3 support a few kinds of EOA described in table below.

Type	Description
Default	typically first of node accounts list, **should be unlocked**
Personal	available via `personal_*` JSON-RPC, **password required**
PrivateKey	derived from secp256k1 private key, use JSON-RPC *sendRawTransaction*

All of them has an instance for Account typeclass.

```
class MonadTrans t => Account a t | t -> a where
   -- | Run computation with given account credentials
   withAccount :: JsonRpc m => a -> t m b -> m b

   -- | Send transaction to contract, like a 'write' command
   send :: (JsonRpc m, Method args) => args -> t m TxReceipt

   -- | Call constant method of contract, like a 'read' command
   call :: (JsonRpc m, Method args, AbiGet result) => args -> t m result
```

The `Account` is a multi-parameter typeclass that define most important EOA actions.

6.1 Account managing

The first parameter of `Account` typeclass is an account internal params presended as independent data type.

```haskell
-- | Unlockable node managed account params
data Personal = Personal
   { personalAddress    :: !Address
   , personalPassphrase :: !Passphrase
   } deriving (Eq, Show)
```

In this example `Personal` data contains of two params: personal account address and password. For using account credentials `Account` typeclass provide special function `withAccount`.

```haskell
-- | Run computation with given account credentials
withAccount :: JsonRpc m => a -> t m b -> m b
```

`withAccount` function takes two arguments: account initialization parameters (some credentials like a password or private key) and computation to run it in given account context. Finally it returns `JsonRpc` computation that can be runned using any web3 provider.

```haskell
runWeb3 $ do

   -- Run with default account context
   withAccount () $ ...

   -- Run with personal account context
   withAccount (Personal "0x..." "password") $ ...
```

6.2 Transaction sending

The second parameter of `Account` typeclass is transaction parametrization monad. This monad do one thing - prepare transaction parameters before call.

Note: Transaction sending diagram by layer looks like `provider -> account -> transaction`, provider at low level, account at middle layer and transaction former at high level.

`withParam` is a special function, it behaviour is very similar to `withStateT` function. It used to set parameters of transaction locally and revert params after out of scope.

```haskell
withParam :: Account p (AccountT p)
          => (CallParam p -> CallParam p)
          -> AccountT p m a
          -> AccountT p m a
```

The first argument of `withParam` function is state transition function, second - the computation to run in context of changed state. `CallParam` helps to parametrize transaction sending, lenses is very useful for this purpose.

```haskell
runWeb3 $
   withAccount () $
      withParam (to .~ alice) $
         ...
```

Where lens `to` is used for setting transaction recipient address. All transaction parametrization lenses presended in table below.

Note: By default transaction gas limit estimated according to transaction input but it also can be set manually.

Finally for sending transactions `Account` typeclass provide two functions:

```
-- | Send transaction to contract, like a 'write' command
send :: (JsonRpc m, Method args) => args -> t m TxReceipt

-- | Call constant method of contract, like a 'read' command
call :: (JsonRpc m, Method args, AbiGet result) => args -> t m result
```

Note: Functions above can be run in account context only and transaction parameters should be set before.

6.3 Safe transactions

Default behaviour of `send` function is send transaction and waiting for transaction receipt. It does mean that transaction is already in blockchain when execution flow get back. But finalization in Ethereum is probabilistic. For this reason waiting for some count of confirmation is a good practics for safe transaction sending.

Note: Vitalik Buterin blog post describe how much confirmation is required for high probability of transaction finality. For using this value import `safeConfirmations` from `Network.Ethereum.Account.Safe` module.

Module `Network.Ethereum.Account.Safe` implements function `safeSend`. It very similar to `send` but take count of transaction confirmation as first argument.

```
send = safeSend 0
```

CHAPTER 7

Smart contracts

If Ethereum is a World Computer then smart contract is a program for that. **hs-web3** provide functions and abstractions to compile, deploy and interact with smart contracts.

Note: Currently **Solidity** is de facto standard for Ethereum smart contract development. Please read intro to get more knowledge about Solidity smart contracts.

7.1 Contract ABI

One of the most important thing that Solidity introduce is a contract Application Binary Interface. ABI is a standard for smart contract communication, both from outside the Ethereum and for contract-to-contract interaction. In hs-web3 Quasiquotation is used to parse contract JSON ABI or load from file. TemplateHaskell driven generator creates ABI encoding instances and contract method helpers automatically.

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE DeriveGeneric          #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings      #-}
{-# LANGUAGE QuasiQuotes            #-}
module ERC20 where

import           Network.Ethereum.Contract.TH

[abiFrom|ERC20.json|]
```

Using Solidity contract ABI generator creates helper functions like a `transfer` and `balanceOf`.

```
transfer :: (JsonRpc m, Account a t, Functor (t m)) => Address -> UIntN 256 -> t m␣
↪TxReceipt

balanceOf :: (JsonRpc m, Account a t, Functor (t m)) => Address -> t m (UIntN 256)
```

Note: Use `-ddump-splices` to see generated code during compilation or in GHCi.

Helper functions wraps building and sending transaction with given argument and function selector. This behaviour is very similar to web3.js contract object.

7.2 ABI encoding

To build transaction input from Solidity method call special encoding is used. **hs-web3** implements Solidity ABI encoding for primitive and composed types. Codecs are placed at `Data.Solidity.Abi.Codec`.

```
encode :: (AbiPut a, ByteArray ba) => a -> ba

decode :: (ByteArrayAccess ba, AbiGet a) => ba -> Either String a
```

Note: When I develop codecs I was inspired by cereal library. As result `AbiGet` and `AbiPut` classes are analogue to cereal `Serialize`.

Primitive solidity types are placed at `Data.Solidity.Prim`, this module exports types like an `Address` or `UIntN`.

```
> import Data.Solidity.Prim
> import Data.Solidity.Abi.Codec
> encode (42 :: UIntN 128) :: HexString
HexString "0x0000000000000000000000000000000000000000000000000000000000000000000000000000002a"
> encode (42 :: IntN 256, "Hello" :: Text) :: HexString
HexString
↪"0x0000000000000000000000000000000000000000000000000000000000000002a0000000000000000000000000000000000
↪"
```

7.3 Contract deployment

To deploy smart contract special function with name `new` is used.

```
-- | Create new smart contract on blockchain
new :: (Account p t, JsonRpc m, Method a, Monad (t m))
    => a
    -- ^ Contract constructor
    -> t m (Maybe Address)
    -- ^ Address of deployed contract when transaction success
```

This function use `Method` instance of contract constructor (`*Contract` data type) to encode transaction input and send it without destination to create new contract.

```
Just address <- runWeb3 $ withAccount () $ withParam id $ new SimpleStorageContract
```

CHAPTER 8

Ipfs Client API

As many Ethereum Dapps use Ipfs for data storage, an Ipfs Client Api has been included.

Note: The api client is placed at `Network.Ipfs.Api`. `Network.Ipfs.Api.Ipfs` exports the api functions.

8.1 Api Type

The api type is defined in `Network.Ipfs.Api.Api`. The client uses the Servant Library.

```
ipfsApi :: Proxy IpfsApi
ipfsApi =  Proxy

_cat :<|> _ls ... :<|> _shutdown = client ipfsApi
```

The aeson definitions of the data types returned by the api functions are also present in `Network.Ipfs.Api.Api`.

8.2 Monad Runner

`Network.Ipfs.Api.Ipfs` exports `runIpfs` monad runner and api functions.

```
runIpfs' :: BaseUrl -> Ipfs a -> IO ()

runIpfs :: Ipfs a -> IO ()
runIpfs = runIpfs' (BaseUrl Http "localhost" 5001 "/api/v0")
```

Note: As you can see `runIpfs` uses the default BaseUrl at **http://localhost:5001/api/v0/** .

You can create a BaseUrl instance for any other IPFS Api provider and pass it to **runIpfs'**.

Example of calling functions (Non-Stream) with runIpfs :

```
main = I.runIpfs $ do
        ret <- I.cat <Cid>
```

Cid should be of the type Text.

8.3 Call Functions

There are three type of call functions:

Type	Description
call	Regular Call function.
multipartCall	Call function for 'multipart/form-data'.
streamCall	Call function for Streams.

As streamCall returns IO(), the API functions using it has to be called with **liftIO** (Specified in Function Documentation).

```
main = I.runIpfs $ do
        ret3 <- liftIO $ I.repoVerify
```

CHAPTER 9

Ethereum Name Service

ENS offers a secure & decentralised way to address resources both on and off the blockchain using simple, human-readable names.

Note: Experimental ENS on Ethereum mainnet added in release `0.8`.

For resolving addresses from ENS names please use `resolve` function from Network.Ethereum.Ens.

```
import qualified Network.Ethereum.Ens as Ens
import           Network.Ethereum.Web3
import           Lens.Micro ((.~))

main = runWeb3 $ withAccount () $ do
   alice <- Ens.resolve "aliceaccount.eth"

   withParam (to .~ alice) $
      withParam (value .~ (1 :: Ether)) $
         send ()
```

CHAPTER 10

Contributing

10.1 Did you find a bug?

- **Ensure the bug was not already reported** by searching on GitHub under Issues.
- If you're unable to find an open issue addressing the problem, open a new one. Be sure to include a **title and clear description**, as much relevant information as possible.

Also you can open an issue if you have a proposal for an improvements.

10.2 Did you write a patch that fixes a bug?

- Open a new GitHub pull request with the patch.
- Ensure the PR description clearly describes the problem and solution. Include the relevant issue number if applicable.

Thanks!

CHAPTER 11

Testing

Testing the **web3** is split up into two suites: **unit** and **live**.

- The unit suite tests internal library facilities.
- The live tests that the library adequately interacts with a Web3 provider.

One may simply run `stack test` to run both suites, or `stack test web3:unit` or `stack test web3:live` to run the test suites individually.

Note: The live suite also requires a Web3 provider with Ethereum capabilities, as well as an unlocked account with ether to send transactions from.