Chanterelle Documentation

Release v0.8.2.1

Martin Allen, Ilya Ostrovskiy

May 01, 2018

Contents:

CHAPTER 1

chanterelle.json

A Chanterelle project is primarily described in `chanterelle.json`, which should be placed in the root of your project. A sample project is defined below, based on the parking-dao application.

```
{ "name": "parking-dao",
  "version": "0.0.1",
  "source-dir": "contracts",
  "artifacts-dir": "build/contracts",
  "modules": [ "FoamCSR"
             , "ParkingAuthority"
             , "SimpleStorage"
             , "User"
             , "ParkingAnchor"
             , "Nested.SimpleStorage"
             ],
  "dependencies": ["zeppelin-solidity"],
  "libraries": {
      "FixedLib": "0x1337133713371337133713371337133713371337",
      "AutocompiledLib": {
          "address": "0xf00dcafe0ea7beef808080801234567890ABCDEF",
          "code": {
              "file": "src/MyLibraries/AutocompiledLib.sol"
          }
      },
      "InjectedLib": {
          "address": "0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef",
          "code": {
              "bytecode": "0x73deadbeefdeadbeefdeadbeefdeadbeefdeadbeef301460606040...
↪"
          }
      },
      "FetchedLib": {
          "address": "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4",
          "via": ["mainnet", "rinkeby"]
      }
  },
```

```
  "networks": {
        "some-private-net": {
            "url": "http://chain-1924-or-1925.roll-the-dice-see-what-you-get.com:8545/
↪",
            "chains": "1924,1925",
        },
        "a-different-net": {
            "url": "http://mystery-chain.whose-id-always-chang.es:8545/",
            "chains": "*",
        }
  },
  "solc-output-selection": [],
  "purescript-generator": {
      "output-path": "src",
      "module-prefix": "Contracts",
      "expression-prefix": ""
  }
}
```

1.1 Specification

Note: All filepaths are relative to the `chanterelle.json` file, which is considered the project root.

`chanterelle.json` - specification:

- `name` - `Required`: The name of your project (currently unused, for future use with package management)
- `version` - `Required`: The currrent version of your project (currently unused, for future use with package management)
- `source-dir` - `Required`: Where your Solidity contracts are located.
- `artifacts-dir` - `Optional`: The directory where the contract artifacts (ABI, bytecode, deployment info, etc) will be written. Defaults to `build`.
- `modules` - `Required`: A list of all solidity contracts you wish to compile (see *Modules*)
- `dependencies` - `Optional`: External Solidity (source-code) libraries/dependencies to use when compiling (see *Dependencies*).
- **`libraries` - `Optional`: Solidity libraries to link against when compiling.**
  - A library may be defined as just an address. This is known as a "Fixed Library". In this case, that address will be fed to solc when compiling contracts that depend on it. However, generating genesis blocks will be unavailable as there is no means to get the code for the given library.
  - A library may alternatively be defined as an address and a means to fetch the library. The address will be fed into solc as with a Fixed Library, however, when generating a genesis block, Chanterelle will attempt to compile the library or fetch it from any specified networks.
  - If a library is injected as raw bytecode or fetched from a network, that code must begin with a library guard (`0x73<addr>3014`). Chanterelle uses this to automagically inject the correct library address into the Genesis block.
  - Supported networks to fetch from include: `"mainnet"`, `"ropsten"`, `"rinkeby"`, `"kovan"`, and `"localhost"`, as well as any networks defined in the `"networks"` field of your project spec.
- **`networks` - `Optional`: Additional networks to fetch libraries from.**
  - Each network has a required `"url"` field, which tells Chanterelle how to reach a node on that network

  - Each network has a required `"chains"` field, which tells Chanterelle which network IDs to accept from that node. The value may either be a comma-separated list of network ID numbers (still has to be a string for just one network), or `"*"` to accept any network ID.

- `solc-output-selection` - Additional outputs to request from solc (currently unsupported, but see *solc documentation*)
- `purescript-generator` - `Required`: Options for purescript-web3-generator (see below)

`purescript-generator` - options:

- `output-path` - `Required`: Where to place generated PureScript source files, for example this is your PureScript project source directory.
- `module-prefix` - `Optional`: What module name to prefix to your generated PureScript bindings. Note that the generated files will be stored relative to the output path (e.g. if set to `Contracts` as above, code will be generated into `src/Contracts`). Defaults to `Contracts`.
- `expression-prefix` - `Optional`: Prefix *all* generated functions with the specified prefix. This is useful if you are depending on external smart contract libraries that name their solidity events or functions that are invalid purescript names.

CHAPTER 2

Modules

Chanterelle uses a notion of modules to determine which units of Solidity code to compile and create PureScript bindings for. Those coming from a Haskell background may notice the parallel to Cabal's exposed-modules. Simply adding a contract file within the source-dir does not mean it will be compiled, nor will there be a generate PureScript file. Instead, one must explicitly specify it in the project's *chanterelle.json*.

Chanterelle uses module namespacing for Solidity files similar to what's found in PureScript or Haskell, though here we enforce that the module name must mirror the filepath. A module named `Some.Modular.Contract` is expected to be in `contracts/Some/Modular/Contract.sol`, and its PureScript binding will have the same module name as well.

Solidity build artifacts are put into the `build/` directory by default (see `artifacts-dir` in *chanterelle.json*). The full artifact filepath corresponds to the its relative location in the `source-dir`. For example, in the parking-dao the `ParkingAuthority` Solidity module is located at `contracts/ParkingAuthority.sol`. Thus the build artifact will be written to `build/ParkingAuthority.json`.

If a module-prefix is specified in the purescript-generator section, that module prefix will be prepended to the generated PureScript modules' names. In the parking-dao example, the module `FoamCSR` is expected to be located in `contracts/FoamCSR.sol` and will generate a PureScript module `Contracts.FoamCSR` with filepath `src/Contracts/FoamCSR.purs`. Likewise, `Nested.SimpleStorage` is expected to be located in `contracts/Nested/SimpleStorage.sol` and the generated PureScript module name will be `Contracts.Nested.SimpleStorage` with filepath `src/Contracts/Nested/SimpleStorage.purs`.

CHAPTER 3

Dependencies

As the Ethereum ecosystem has not conclusively settled on a Solidity package management structure yet, we support referencing any modules installed in `node_modules` as additional include paths for use in your Solidity imports.

In the parking-dao example project, we have `zeppelin-solidity` as a listed dependency. By listing this dependency, chanterelle will format the `solc` input so that any imports starting with `zeppelin-solidity/` will be fetched from `/path/to/project/node_modules/zeppelin-solidity/`.

In the future we aim to have a more clever system in place for handling this usage scenario.

CHAPTER 4

Compiling

Currently, Chanterelle does not have a dedicated command line tool for invoking its functionality. Instead, one writes a brief PureScript program to invoke the various features of Chanterelle.

Generally, you'd want to have at least two subprojects, one for compiling and one for deploying/testing. This is because the deployer and test suite will surely depend on PureScript bindings generated during the compilation phase, and thus cannot be part of the same project. An example of this can be seen in the Parking DAO example.

4.1 Invoking the compiler

A sample application to invoke the compiler is presented below. This is nearly identical to the Parking DAO compile script, with the exception that this script also invokes the Genesis block generator. One may leave out the `runGenesisGenerator` bit if this functionality is not required. One may want to store this script in a directory outside where their PureScript build system (such as Pulp) would keep code. One such location is `compile/Compile.purs` (as opposed to say, `src/compile/Compile.purs`).

```
module CompileMain where

import Prelude
import Chanterelle (compileMain)
import Chanterelle.Genesis (runGenesisGenerator)
import Control.Monad.Aff.Console (CONSOLE)
import Control.Monad.Eff (Eff)
import Control.Monad.Eff.Exception (EXCEPTION)
import Control.Monad.Eff.Now (NOW)
import Node.FS.Aff (FS)
import Node.Process (PROCESS)
import Network.Ethereum.Web3 (ETH)

main :: forall eff.
      Eff
        ( console :: CONSOLE
        , fs :: FS
```

```
        , process :: PROCESS
        , exception :: EXCEPTION
        , now :: NOW
        , eth :: ETH
        | eff
        )
        Unit
main = do
  compileMain
  runGenesisGenerator "./base-genesis-block.json" "./injected-genesis-block.json"
```

We can then invoke this script as follows

```
pulp build --src-path compile -m CompileMain --to compile.js && \
node compile.js --log-level info; \
rm -f compile.js
```

This will compile and purescript-web3 codegen all the modules specified in `chanterelle.json` as well as generate a genesis block whose contents are those of `./base-genesis-block.json` with injected libraries appended into `allocs` and written out to `./injected-genesis-block.json`.

Note that we do not use `pulp run` as we then have no means to pass command line arguments to the compiler.

4.2 Compiler arguments

Currently the following command line arguments are supported for the compiler phase when ran with `compileMain`:

- `--log-level`: One of `debug`, `info`, `warn`, or `error`. Defaults to `info`. This option changes the level of logging to the console.

CHAPTER 5

Genesis Blocks

Chanterelle supports generating genesis blocks for use with custom blockchains which depend on externally deployed libraries on other chains. It utilizes the `libraries` field in the *chanterelle.json*.

The genesis generator takes the path to a `geth`-compatible genesis block as input (relative to the project root or absolute), and writes the filled-in block to the output path specified, overwriting it if it exists.

As was seen in the compilation example, the genesis generator is invoked with `runGeneratorMain`, customarialy as part of your compilation phase.

As will be noted in the *libraries* chapter, the genesis generator is unable to run if any Fixed libraries are specified in the project spec.

We typically recommend using the genesis generator in conjunction with Cliquebait.

CHAPTER 6

Libraries

Chanterelle supports supplying Solidity libraries to `solc` during compile-time. Future versions will support injecting libraries during the Deployment phase.

6.1 Overview

Each library is a mapping of a library name to one or more parameters describing it. These parameters are utilized during the compilation of contracts as well as when generating genesis blocks. Each library specified will have its address passed into `solc` so that it may automatically be linked into the compiled bytecode.

6.2 Fixed Libraries

The most basic form of library descriptor is simply a library name and an address. This is seldom used in practice, and will prevent the genesis generator from running.

6.3 Autocompiled Libraries

These are the most common form of library you'll likely use. In this case, rather than a string representing the library address, one specified an object containing the library's address as well as where to find the Solidity code for that library.

When generating genesis blocks, the generator will compile the library and use the resulting bytecode.

6.4 Injected Libraries

Injected libraries consist of raw EVM bytecode that represents the code that should exist at the library's address (i.e., as would be received from `eth.getCode("0xaddress")`. Note that most Solidity libraries have checks to ensure

that they are not called directly, and have this written as part of their deployment address when they are first deployed to the blockchain. To handle this, Chanterelle will automatically substitute that section of the bytecode to ensure the library behaves as expected. If a library does implement this check in the standard manner (by having the first bytes be `0x73<addr>3014`), the genesis generator **will** fail.

6.5 Fetched Libraries

Chanterelle may be configured to attempt to fetch the code for a library from an existing network. Should the code be unavailable on all the specified networks, the genesis generator will fail.

The `"via"` field may be one of:

- `"*"`: Attempt to fetch the library from all networks defined in the project spec
- `"**"`: Attempt to fetch the library from both the networks defined in the project spec as well as from the predefined networks: `mainnet`, `ropsten`, `rinkeby`, `kovan`, or `localhost`.
- `["net_name", ...]`: Attempt to fetch the library from any of the named networks. These may include both the project-specific networks as well as the predefined networks. Note that this must be an array even if only one network is specified.

CHAPTER 7

Deployments

7.1 Configuration

Every contract deployment requires an explicit configuration. Specifically, the configuration is an object of the following type:

```
type ContractConfig args =
    { filepath :: String
    , name :: String
    , constructor :: Constructor args
    , unvalidatedArgs :: V (Array String) (Record args)
    }
```

The `filepath` field is the filepath to the solc build artifact relative the the `chanterelle.json` file.

The `name` field is there to name the deployment throughout the logging. (This could dissappear assuming its suffient to name the deployment according to the build artifact filename.)

The type `Constructor args` is a type synonym:

```
type Constructor args = forall eff. TransactionOptions NoPay -> HexString -> Record␣
↪args -> Web3 eff HexString
```

In other words, `Constructor args` is the type a function taking in some `TransactionOptions NoPay` (constructors are not payable transactions), the deployment bytepurescriptcode, and a record of type `Record args`. It will format the transaction and submit it via an `eth_sendTransaction` RPC call, returning the transaction hash as a `HexString`.

The `unvalidatedArgs` field has type `V (Array String) (Record args)` where `V` is a type coming from the purescript-validation library. This effectively represents *either* a type of `Record args` *or* a list of error messages for all arguments which failed to validate.

It's possible that your contract requires no arguments for deployment, and in that case chanterelle offers some default values. For example, if the filepath of the build artifact for `VerySimpleContract.sol` is `build/VerySimpleContract.json`, you might end up with something like

```
verySimpleContractConfig :: ContractConfig NoArgs
verySimpleContractConfig =
  { filepath: "build/VerySimpleContract.json"
  , name: "VerySimpleContract"
  , constructor: constructorNoArgs
  , unvalidatedArgs: noArgs
  }
```

Let's consider the simplest example of a contract configuration requiring a constructor with arguments. Consider the following smart contract:

```
contract SimpleStorage {

  uint256 count public;

  event CountSet(uint256 _count);

  function SimpleStorage(uint256 initialCount) {
    count = initialCount;
  }

  function setCount(uint256 newCount)) {
    count = newCount;
    emit CountSet(newCount));
  }

}
```

Depending on your project configuration, when running `chanterelle compile` you should end up with something like the following artifacts:

1. The solc artifact `build/SimpleStorage.json`
2. The generated PureScript file `src/Contracts/SimpleStorage.purs`

In the PureScript module `Contracts.SimpleStorage`, you will find a function

```
constructor :: forall e. TransactionOptions NoPay -> HexString -> {initialCount ::␣
↪UIntN (D2 :& D5 :& DOne D6)} -> Web3 e HexString
```

Blurring your eyes a little bit, it's easy to see that this indeed matches up to the constructor defined in the Solidity file. We could then define the deployment configuration for `SimpleStorage` as

```
import Contracts.SimpleStorage as SimpleStorage

simpleStorageConfig :: ContractConfig (initialCount :: UIntN (D2 :& D5 :& DOne D6))
simpleStorageConfig =
    { filepath: "build/SimpleStorage.json"
    , name: "SimpleStorage"
    , constructor: SimpleStorage.constructor
    , unvalidatedArgs: validCount
    }
  where
    validCount = uIntNFromBigNumber s256 (embed 1234) ?? "SimpleStorage: initialCount␣
↪must be valid uint256"
```

Here you can see where validation is important. Clearly `1234` represents a valid `uint`, but you can easily imagine scenarios where this might save us a lot of trouble– too many characters in an address, an improperly formatted string, an integer is out of a bounds, etc.

7.2 Deploy Scripts

Deploy scripts are written inside the `DeployM` monad, which is a monad that gives you access to a web3 connection, controlled error handling, and whatever effects you want. The primary workhorse is the `deployContract` function:

```
deployContract :: TransactionOptions NoPay -> ContractConfig args -> DeployM
↪{deployAddress :: Address, deployArgs :: Record args}
```

This function takes your contract deployment configuration as defined above and sends the transaction. If no errors are thrown, it will return the address where the contract as deployed as well as the deploy arguments that were validated before the transaction was sent. It will also automatically write to the solc artifact in the `artifacts-dir`, updating the `networks` object with a key value pair mapping the networkId to the deployed address.

Error hanlding is built in to the `DeployM` monad. Unless you want to customize your deployment with any attempt to use some variant of try/catch, any error encountered before or after a contract deployment will safely terminate the script and you should get an informative message in the logs. It will not terminate while waiting for transactions to go through unless the timeout threshold is reached. You can configure the duration as a command line argument.

7.3 Deployment Example

Consider this example take from the parking-dao example project:

```
module MyDeployScript where

import ContractConfig (simpleStorageConfig, foamCSRConfig, parkingAuthorityConfig)

type DeployResults = (foamCSR :: Address, simpleStorage :: Address, parkingAuthority␣
↪:: Address)

deployScript :: forall eff. DeployM eff (Record DeployResults)
deployScript = do
  deployCfg@(DeployConfig {primaryAccount}) <- ask
  let bigGasLimit = unsafePartial fromJust $ parseBigNumber decimal "4712388"
      txOpts = defaultTransactionOptions # _from ?~ primaryAccount
                                         # _gas ?~ bigGasLimit
  simpleStorage <- deployContract txOpts simpleStorageConfig
  foamCSR <- deployContract txOpts foamCSRConfig
  let parkingAuthorityConfig = makeParkingAuthorityConfig {foamCSR: foamCSR.
↪deployAddress}
  parkingAuthority <- deployContract txOpts parkingAuthorityConfig
  pure { foamCSR: foamCSR.deployAddress
       , simpleStorage: simpleStorage.deployAddress
       , parkingAuthority: parkingAuthority.deployAddress
       }
```

After setting up the `TransactionOptions`, the script first deploys the `SimpleStorage` contract and then the `FoamCSR` contract using their configuration. The `ParkingAuthority` contract requires the address of the `FoamCSR` contract as one of it's deployment arguments, so you can see us threading it in before deploying. Finally, we simple return all the addresses of the recently deployed contracts to the caller.

Note that if we simply wanted to terminate the deployment script after the contract deployments there then there's no point in returning anything at all. However, deployment scripts are useful outside of the context of a standalone script. For example you can run a deployment script before a test suite and then pass the deployment results as an environment to the tests. See the section on testing for an example.

7.4 Invocation

Much like with the *compilation phase*, the deployment phase is invoked with a minimal PureScript boilerplate. This script, however, invokes the `deployScript` you defined previously, and may either reside with the rest of your source or more methodically in a separate `deploy/` subproject. The latter is demonstrated below

```
module DeployMain (main) where

import Prelude

import Chanterelle (deployMain)
import Control.Monad.Eff (Eff)
import Control.Monad.Eff.Console (CONSOLE)
import Control.Monad.Eff.Exception (EXCEPTION)
import Control.Monad.Eff.Now (NOW)
import Network.Ethereum.Web3 (ETH)
import Node.FS.Aff (FS)
import Node.Process (PROCESS)
import MyDeployScript (deployScript) as MyDeployScript

main :: forall e. Eff (now :: NOW, console :: CONSOLE, eth :: ETH, fs :: FS, process␣
↪:: PROCESS, exception :: EXCEPTION | e) Unit
main = deployMain MyDeployScript.deployScript
```

We can then invoke this script as follows:

```
pulp build --src-path deploy -I src -m DeployMain --to deploy.js && \
node deploy.js --log-level info; \
rm -f deploy.js
```

One may note the similarities to the invocation of the compiler script, however the build has an additional `-I src` as your deploy script will mostly likely depend on artifacts that are codegen'd into your main source root as well.

7.5 Deployer arguments

Currently the following command line arguments are supported for the deployment phase when ran with `deployMain`:

- `--log-level`: One of `debug`, `info`, `warn`, or `error`. Defaults to `info`. This option changes the level of logging to the console.

CHAPTER 8

Testing

8.1 Configuration

You can use whatever purescript testing framework you want, but for illustrative purposes we will use purescript-spec.

There are various testing utility functions in `Chanterelle.Test`, probably the most important is `buildTestConfig`.

```
type TestConfig r =
  { accounts :: Array Address
  , provider :: Provider
  | r
  }


buildTestConfig
  :: String
  -> Int
  -> DeployM eff (Record r)
  -> Aff (console :: CONSOLE, eth :: ETH, fs :: FS | eff) (TestConfig r)
```

This function takes in some test configuration options and a deploy script, and outputs a record containing all of the unlocked accounts on the test node, a connection to the node, and whatever the output of your deployment script is. This output is then meant to be threaded through as an environment to the rest of your test suites.

Note, unlike the deploy process meant for actual deployments, `buildTestConfig` will not write anything to the file system about the result of your deployment. In other words, test deployments are ephemeral.

8.2 Example Test Suite

Here's an example test suite for our `SimpleStorage` contract:

```
simpleStorageSpec
  :: forall e r.
     TestConfig (simpleStorage :: Address | r)
  -> Spec ( fs :: FS
          , eth :: ETH
          , avar :: AVAR
          , console :: CONSOLE
          | e
          ) Unit
simpleStorageSpec {provider, accounts, simpleStorage} = do

  describe "Setting the value of a SimpleStorage Contract" do

    it "can set the value of simple storage" $ do
      var <- makeEmptyVar
      let filterCountSet = eventFilter (Proxy :: Proxy SimpleStorage.CountSet)␣
↪simpleStorage
      _ <- forkWeb3 provider $
        event filterCountSet $ \e@(SimpleStorage.CountSet cs) -> do
          liftEff $ log $ "Received Event: " <> show e
          _ <- liftAff $ putVar cs._count var
          pure TerminateEvent
      let primaryAccount = unsafePartialBecause "Accounts list has at least one␣
↪account" $ fromJust (accounts !! 0)
          n = unsafePartial fromJust <<< uIntNFromBigNumber s256 <<< embed $ 42
          txOptions = defaultTransactionOptions # _from ?~ primaryAccount
                                                # _to ?~ simpleStorage
                                                # _gas ?~ embed 90000
      hx <- assertWeb3 provider $ SimpleStorage.setCount txOptions {_count: n}
      liftEff <<< log $ "setCount tx hash: " <> show hx
      val <- takeVar var
      val `shouldEqual` n
```

The flow of the test is as follows:

1. We create an `AVar` to communicate between the testing thread and the event monitoring thread.
2. We then fork an event monitoring thread for our `CountSet` event, placing the first received value in the `AVar` and terminating the monitor. Notice that the address for the filter is taken from the supplied `TestConfig`.
3. We create then our `TransactionOptions` and submit a transaction to change the count using the `setCount` function from the generated PureScript module.
4. We call `takeVar` which blocks until the var is filled, then make sure the value we received is the one we put in.

Admittingly this example is pretty trivial– of course we're going to get back the value we put in. However, this pattern is pretty universal, namely take the supplied test config to help you template the transactions, call some functions, monitor for some event, then make sure the values are what you want.