Chanterelle Documentation

Release stable, v6.0.0

Martin Allen, Ilya Ostrovskiy

Jul 14, 2021

Contents:

CHAPTER 1

Installation

Chanterelle exists as both a library and an command-line executable. The library enables your applications to take advatange of Chanterelle's rapid development pipeline, while the command-line executable provides easy access to Chanterelle's core functionality.

1.1 Installing Globally

Installing the Chanterelle command-line interface globally allows you to easily access Chanterelle's compilation and code generation interface to prevent cyclical dependency problems when bootstrapping a development environment.

The command-line application first attempts to use a project-local version of Chanterelle when invoking commands, but has the ability to fall back to a "global" instance, which is necessary when a project is first created. Chanterelle tries to use a local installation first to prevent incompatibilities that may occur if there is a mismatch between the global version and the project-local version.

In order for Chanterelle to be able to fall back to a global installation, one must be compiled after installing Chanterelle via NPM. This is done via the `chanterelle global-postinstall` subcommand.

You can install the HEAD of *master* globally by running:

```
npm install -g f-o-a-m/chanterelle  # Install bleeding-edge Chanterelle CLI
chanterelle global-postinstall      # Bootstrap the global installation
```

Or if you would like to install a specific version, you may specify it via NPM:

```
npm install -g f-o-a-m/chanterelle#v3.0.0 # Install Chanterelle CLI v3.0.0
chanterelle global-postinstall            # Bootstrap the global installation
```

1.2 Installing Locally (per-project)

You will likely also want to install Chanterelle local to particular project:

```
npm install --save f-o-a-m/chanterelle      # Add the Chanterelle CLI to NPM path for␣
↪NPM package scripts
```

If the Chanterelle CLI is invoked within a project containing a local installation that has been compiled, it will use the version within that project as opposed to the global one.

1.3 What is this cyclic dependency stuff all about?

Chanterelle always first attempts to use the version installed within your project. When you've written a deployment script, your PureScript compiler will automatically compile the necessary components of Chanterelle to enable it to run as a CLI command independent of a global installation. This is an important step, as it prevents version mismatches from affecting the integrity of your deployment scripts. To this end, the Chanterelle CLI command always attempts to first use your project-local copy of Chanterelle, However, your project needs to compile successfully before that is available. Your deployment script will likely fail to compile as it would depend on PureScript bindings to the smart contracts which are being deployed – bindings which Chanterelle generates – and if Chanterelle can't compile because your project can't compile – then you'd be stuck in an endless cyclic dependency. To resolve this, Chanterelle allows you to compile an independent global version to fall back to, which would allow you to compile your contracts and generate any PureScript necessary to proceed with compilation.

1.4 Fair enough, but why do I need to run this `global-postinstall` subcommand?

Great question! Chanterelle itself is written in PureScript, and as such it depends on the PureScript compiler. The `global-postinstall` merely compiles the Chanterelle codebase, as it would if you had a project-local version. This is not done as a package postinstall script as, very often in global package setups, NPM might not give sufficient permissions to install the PureScript compiler package that Chanterelle depends on or otherwise install dependencies. To maximize the flexibilty of the global installation feature, and avoid running into user-specific permissions Chanterelle separates out this step such that it is independent of being installed via NPM.

To avoid running into a myriad of permissions issues, we recommend using *NVM <https://github.com/nvm-sh/nvm>* for managing globally available NPM packages.

CHAPTER 2

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": {
      "ALibrary": "src/MyLibraries/AutocompiledLib.sol",
      "AnotherLib": {
          "root": "node_modules/some-npm-lib/contracts",
          "file": "AnotherLib.sol"
      },
  },
  "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": [],
```

(continues on next page)

(continued from previous page)

```
  "solc-optimizer": {
      "enabled": false,
      "runs: 200
  },
  "solc-version": "<default>",
  "purescript-generator": {
      "output-path": "src",
      "module-prefix": "Contracts",
      "expression-prefix": ""
  }
}
```

2.1 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 compile, which can be deployed and linked against.**
  - All libraries will automatically be compiled and output to `<artifacts-dir>/libraries` as part of the compile stage.
  - Unlike modules, no PureScript bindings are generated for libraries, as they are intended to be used by other Solidity contracts.
- **`networks` - `Optional`: Reserved for future use with package management and more elaborate deployment functionalit**
  - 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-optimizer` - `Optional`: Optimizer options to pass to solc. Defaults are what's shown in the example.**
  - Supports the `enabled` and `runs` fields, which are passed on to `solc`.
- `solc-output-selection` - `Optional`: Additional outputs to request from solc. Sometimes helpful for debugging Chanterelle itself. (currently unsupported, but see *solc documentation*)
- **`solc-version` - `Optional`: Use a different version of the Solidity compiler.**

  - The default is whatever `solc` npm module is available in your build environment.
  - It is recommended to use a version of `solc` >=0.5 in your package.json, and override to a lower version in `chanterelle.json`. This is because Chanterelle calls out to node.js `solc`, and newer versions of the package have better compatibility in the input and output format.
  - Chanterelle ships with solc "^0.5" by default.

- `purescript-generator` - `Required`: Options for purescript-web3-generator (see below)

`purescript-generator` - options:

- `output-path` - `Required`: Where to place generated PureScript source files. Generally, this would be your PureScript project's 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 contracts or libraries that have Solidity events or functions whose names would be invalid PureScript identifiers.

CHAPTER 3

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 generated 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 4

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. We see the potential for future EthPM 1.0 and 2.0 support as it appears to be the direction many new Solidity developments are looking towards.

In the parking-dao example project, we have `zeppelin-solidity` as a listed dependency. By listing this dependency, Chanterelle will provide a remapping to the Solidity compiler 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 5

Compiling

```
chanterelle build
# which is shorthand for
chanterelle compile && chanterelle codegen
```

This will compile and generate PureScript bindings for all the modules specified in `chanterelle.json`. Additionally, libraries will also be compiled and placed in a special `libraries` subdirectory, but no PureScript bindings will be generated, as libraries are intended to be called by other Solidity contracts instead of Web3 applications. Nonetheless, you will likely need to link your Solidity code to libraries, and so artifacts are generated to allow you to keep track of libraries.

Chanterelle will only recompile modules and libraries that are "stale". A stale module is one whose build artifact has been modified before its corresponding source file, or one whose artifact has last been updated before `chanterelle.json`. If you attempt to compile and see nothing changing, and no output in your terminal about compiling, it probably means there's nothing to do, and chanterelle is clever about it.

CHAPTER 6

Libraries

Chanterelle supports compiling and linking Library contracts during deployment

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

CHAPTER 7

Deployments

7.1 Configuration

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

```
type DeployReceipt args =
  { deployAddress :: Address
  , deployArgs :: Record args
  , deployHash :: HexString
  }

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 = TransactionOptions NoPay -> HexString -> Record args -> Web3␣
↪HexString
```

In other words, `Constructor args` is the type a function taking in some `TransactionOptions NoPay` (constructors are not payable transactions), the deployment bytecode as a PureScript HexString, 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`. This constructor function is generated by Chanterelle as part of the codegen step. Note that this function is taking in the raw bytecode as an argument. This is because your contract

may have unlinked libraries and it doesn't make sense to generate it into the PureScript binding. When you call `deployContract` with your `ContractConfig`, Chanterelle will automatically feed in the appropriate bytecode.

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 :: TransactionOptions NoPay -> HexString -> {initialCount :: UIntN (D2 :& D5 :& DOne D6)} -> Web3 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"
```

(continues on next page)

(continued from previous page)

```
    , 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␣
↪(DeployReceipt 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)

deploy :: DeployM Unit
deploy = void deployScript

type DeployResults = (foamCSR :: Address, simpleStorage :: Address, parkingAuthority␣
↪:: Address)
deployScript :: DeployM (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}
```

(continues on next page)

(continued from previous page)

```
  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 Libraries

Library deployments are very similar to those of contracts. Libraries exist as code that simply exists on the blockchain and gets invoked by other contracts. As such, library contracts do not have constructors nor do they take any arguments, and all that is necessary to specify them is the file path to the artifact and the name of the Library contract.

```
type LibraryConfig args =
    { filepath :: String
    , name :: String
    | args
    }
```

You may find that this looks strangely similar to the type definition for `ContractConfig`, and are also probably curious as to what purpose the `args` serves. Well, to keep Chanterelle internals and general as possible, the `ContractConfig` is actually a subset of `LibraryConfig`, and the `args` in `LibraryConfig` exists to add additional fields to the record type internally. In fact, the previous definition of `ContractConfig` was a bit of a lie, it's really `type ContractConfig args = LibraryConfig (constructor :: Constructor args, unvalidatedArgs :: V (Array String) (Record args))`. What this is saying is that the bare minimum for deploying any kind of solidity code with Chanterelle is the path to the artifact and the name of the code. Either way, once you've defined a configuration for your library, you may deploy it using:

```
type LibraryMeta = (libraryName :: String, libraryAddress :: Address)

deployLibrary :: TransactionOptions NoPay -> LibraryConfig () -> DeployM␣
↪(DeployReceipt LibraryMeta)
```

This looks eerily similar to `deployContract`, except instead of a `ContractConfig args`, it takes a `LibraryConfig ()`. The type prevents you from trying to deploy a contract as a Library (with no constructor or arguments), and similar to the `deployContract` function which returned the validated arguments passed into the constructor, `deployLibrary` returns some metadata about the Library's name and the address it was deployed to. This is meant to be used in conjunction with the `linkLibrary` function:

```
linkLibrary:: ContractConfig args -> Record LibraryMeta -> DeployM ArtifactBytecode
```

What this is saying is "given an artifact configuration, and a record containing library metadata, link the library into the contract bytecode. Note that `Record LibraryMeta` is exactly the same type as that of the `deployArgs :: Record args` in `DeployReceipt args`. This means that if you had a contract named `MyContract` that needs a libraries named `MyLibrary` and `MyOtherLibrary` linked into it to function, you can do something akin to:

```
deployScript = do
  deployCfg@(DeployConfig {primaryAccount}) <- ask
  let bigGasLimit = unsafePartial fromJust $ parseBigNumber decimal "4712388"
      txOpts = defaultTransactionOptions # _from ?~ primaryAccount
                                         # _gas ?~ bigGasLimit
  myLibrary <- deployLibrary txOpts myLibraryConfig
  myOtherLibrary <- deployLibrary txOpts myOtherLibraryConfig
  _ <- linkLibrary myContractConfig myLibrary.deployArgs
  _ <- linkLibrary myContractConfig myOtherLibrary.deployArgs
  myContract <- deployContract txOpts myContractConfig
```

The `linkLibrary` function returns the linked bytecode for situations in which you wish to inspect how your bytecode changes as libraries gets linked in. You do not have to hold on to it or otherwise use it for anything.

7.5 Invocation

depending on your setup you should make sure *MyDeployScript* module is built. in most cases you can access corresponding js file in *./output/MyDeployScript/index.js* which should be passed to *chanterelle deploy* command like this:

```
chanterelle deploy ./output/MyDeployScript/index.js
```

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 (Record r)
  -> Aff (TestConfig r)
```

This function takes in some test configuration options and a deploy script, runs the 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. Furthermore, while the Chanterelle artifacts are being read from your filesystem to fetch their bytecode for deployment, any deployments previously performed will be ignored, even if deployment information with the same network ID as your test node exists in the artifact.

In other words, test deployments are entirely ephemeral and oblivious to any other deployments.

8.2 Example Test Suite

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

```
simpleStorageSpec
  :: forall r.
     TestConfig (simpleStorage :: Address | r)
  -> Spec 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 taking 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.