Chanterelle Documentation

Release latest

Martin Allen, Ilya Ostrovskiy

Sep 22, 2023

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