Populus Documentation

Release 2.0.0-alpha.1

Piper Merriam

Feb 01, 2017

Contents

Populus is a smart contract development framework for the Ethereum blockchain.

CHAPTER 1

Contents

1.1 Quickstart

1.1.1 System Dependencies

Populus depends on the following system dependencies.

- Solidity : For contract compilation
- Go Ethereum: For running test chains and contract deployment.

In addition, populus needs some system dependencies to be able to install the PyEthereum library.

Debian, Ubuntu, Mint

```
sudo apt-get install libssl-dev
```

Fedora, CentOS, RedHat

```
sudo yum install openssl-devel
```

OSX

```
brew install pkg-config libffi autoconf automake libtool openssl
```

1.1.2 Installation

Populus can be installed using `pip` as follows.

```
$ pip install populus
```

By default populus will use standard library tools for io operations like threading and subprocesses. Populus can be configured to instead use `gevent`. To install with gevent support:

```
$ pip install populus[gevent]
```

To enable `gevent` set the environment variable `THREADING_BACKEND=gevent`.

Installation from source can be done from the root of the project with the following command.

```
$ python setup.py install
```

1.1.3 Initializing a new project

Populus can initialize your project using the `$ populus init` command.

```
$ populus init
Wrote default populus configuration to `./populus.json`.
Created Directory: ./contracts
Created Example Contract: ./contracts/Greeter.sol
Created Directory: ./tests
Created Example Tests: ./tests/test_greeter.py
```

Your project will now have a `./contracts` directory with a single Solidity source file in it named `Greeter.sol`, as well as a `./tests` directory with a single test file named `test_greeter.py`.

1.1.4 Compiling your contracts

Before you compile our project, lets take a look at the `Greeter` contract that is generated as part of the project initialization.

```
pragma solidity ^0.4.0;

contract Greeter {
    string public greeting;

    function Greeter() {
        greeting = "Hello";
```

```
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() constant returns (string) {
        return greeting;
    }
}
```

`Greeter` is simple contract that is initialized with a default greeting of the string `'Hello'`. It exposes the `greet` function which returns whatever string is set as the greeting, as well as a `setGreeting` function which allows the greeting to be changed.

You can now compile the contract using `$ populus compile`

```
$ populus compile
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

1.1.5 Testing your contract

Now that you have a basic contract you'll want to test that it behaves as expected. The project should already have a test module named `test_greeter.py` located in the `./tests` directory that looks like the following.

```
def test_greeter(chain):
    greeter = chain.get_contract('Greeter')

    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter = chain.get_contract('Greeter')

    set_txn_hash = greeter.transact().setGreeting('Guten Tag')
    chain.wait.for_receipt(set_txn_hash)

    greeting = greeter.call().greet()
    assert greeting == 'Guten Tag'
```

You should see two tests, one that tests the default greeting, and one that tests that we can set a custom greeting. You can run tests using the `py.test` command line utility which was installed when you installed populus.

```
$ py.test tests/
collected 2 items
```

```
tests/test_greeter.py::test_greeter PASSED
tests/test_greeter.py::test_custom_greeting PASSED
```

You should see something akin to the output above with three passing tests.

1.2 Overview

1.2.1 Introduction

The primary interface to populus is the command line command `$ populus`.

1.2.2 Command Line Options

```
$ populus
Usage: populus [OPTIONS] COMMAND [ARGS]...

  Populus

Options:
  -c, --config FILENAME  Specify a populus configuration file to be used.
  -h, --help             Show this message and exit.

Commands:
  chain          Manage and run ethereum blockchains.
  compile        Compile project contracts, storing their...
  deploy         Deploys the specified contracts to a chain.
  init           Generate project layout with an example...
  makemigration  Generate an empty migration.
  migrate        Run project migrations
```

1.2.3 Project Layout

By default Populus expects a project to be layed out as follows.

```
- project root
    - populus.json
    - build (automatically created during compilation)
    |   - contracts.json
    - contracts
    |   - MyContract.sol
    |   - ....
    - tests
```

```
        - test_my_contract.py
        - test_some_other_tests.py
        - ....
        - ....
```

Initialize

```
$ populus init --help
Usage: populus init [OPTIONS]

  Generate project layout with an example contract.

Options:
  -h, --help  Show this message and exit.
```

Running `$ populus init` will initialize the current directory with the default project layout that populus uses.

- `./contracts/`
- `./contracts/Greeter.sol`
- `./tests/test_greeter.py`

1.3 Tutorial

Learn how to use populus by working your way through the following tutorials.

1.3.1 Contents

Part 1: Basic Testing

Introduction

The following tutorial picks up where the quickstart leaves off. You should have a single solidity contract named `Greeter` located in `./contracts/Greeter.sol` and a single test module `./tests/test_greeter.py` that contains two tests.

Modify our Greeter

Lets add way for the Greeter contract to greet someone by name. We'll do so by adding a new function `greet(bytes name)` which you can see below. Update your solidity source to match this updated version of the contract.

```
pragma solidity ^0.4.0;

contract Greeter {
    string public greeting;

    function Greeter() {
        greeting = "Hello";
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() constant returns (string) {
        return greeting;
    }

    function greet(bytes name) constant returns (bytes) {
        // create a byte array sufficiently large to store our greeting.
        bytes memory namedGreeting = new bytes(
            name.length + 1 + bytes(greeting).length
        );

        // push the greeting onto our return value.
        // greeting.
        for (uint i=0; i < bytes(greeting).length; i++) {
            namedGreeting[i] = bytes(greeting)[i];
        }

        // add a space before pushing the name on.
        namedGreeting[bytes(greeting).length] = ' ';

        // loop over the name and push all of the characters onto the
        // greeting.
        for (i=0; i < name.length; i++) {
            namedGreeting[bytes(greeting).length + 1 + i] = name[i];
        }
        return namedGreeting;
    }
}
```

Testing our changes

Now we'll want to test our contract. Lets add another test to `./tests/test_greeter.py` so that the file looks as follows.

```
def test_greeter(chain):
    greeter = chain.get_contract('Greeter')

    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter = chain.get_contract('Greeter')

    set_txn_hash = greeter.transact().setGreeting('Guten Tag')
```

```
    chain.wait.for_receipt(set_txn_hash)

    greeting = greeter.call().greet()
    assert greeting == 'Guten Tag'

def test_named_greeting(chain):
    greeter = chain.get_contract('Greeter')

    greeting = greeter.call().greet('Piper')
    assert greeting == 'Hello Piper'
```

You can run tests using the `py.test` command line utility which was installed when you installed populus.

```
$ py.test tests/
collected 3 items

tests/test_greeter.py::test_greeter PASSED
tests/test_greeter.py::test_custom_greeting PASSED
tests/test_greeter.py::test_named_greeting PASSED
```

You should see something akin to the output above with three passing tests.

Part 2: Local Chains

Introduction

In part 1 of the tutorial we modified our `Greeter` contract and expanded the test suite to cover the new functionality.

In this portion of the tutorial we will explore the ability for populus to both run nodes for you as well as connect to running nodes.

Setting up a *local* chain

The first thing we will do is setup a local chain. Create a file in the root of your project named `populus.json` with the following contents

```
{
  "chains": {
    "horton": {
      "chain": {
        "class": 'populus.chain.LocalGethChain'
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider"
        }
```

```
      }
    }
  }
}
```

We have just setup the minimal configuration necessary to run a local chain named `horton`. You can run this chain now in yoru terminal with the following command.

```
$ populus chain run horton
```

You should see **alot** of very verbose output from the running geth node. If you wait and watch you will also see blocks being mined.

Deploying the contract

Now that we have a local chain we can deploy our `Greeter` contract using the `populus deploy` command. When prompted select the listed account.

```
$ populus deploy --chain horton Greeter
Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).

Greeter
Accounts
-----------------
0 - 0xf142ff9061582b7b5f2f39f1be6445947a1f3feb

Enter the account address or the number of the desired account␣
↪[0xf142ff9061582b7b5f2f39f1be6445947a1f3feb]: 0
Deploying Greeter
Deploy Transaction Sent:␣
↪0xd3e6ad1ee455b37bd18703a6686575e9471101fbed7aa21808afd0495e026fe6
Waiting for confirmation...

Transaction Mined
=================
Tx Hash      : 0xce71883741bf4a86e2ca5dd0be5e99888e09888b8a40361a9fb1df81210abe10
Address      : 0x89c2a280a483f45a3d140ef752ffe9c6cd4b57fa
Gas Provided : 433940
Gas Used     : 333940


Verifying deployed bytecode...
Verified contract bytecode @ 0x89c2a280a483f45a3d140ef752ffe9c6cd4b57fa matches␣
↪expected runtime bytecode
Deployment Successful.
```

Note: Your output will differ in that the ethereum address and transaction hashes won't be the same.

It's worth pointing out some *special* properties of local chains.

- They run with all APIs enabled (RPC, IPC, WebSocket)
- They run with the coinbase unlocked.
- They mine blocks using a single CPU.

- Their `datadir` is located in the `./chains` directory within your project.
- The coinbase account is alotted a **lot** of ether.

Having to select which account to deploy from each time you deploy on a chain is tedious. Lets modify our configuration to specify what the *default* deploy address should be. Change your configuration to match this.

```
{
  "chains": {
    "horton": {
      "chain": {
        "class": 'populus.chain.LocalGethChain'
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider"
        },
        "eth": {
          "default_account": "0xf142ff9061582b7b5f2f39f1be6445947a1f3feb"
        }
      }
    }
  }
}
```

You can test this now by deploying the greeter contract again using the same command from above. If everything is configured correctly you should no longer be prompted to select an account.

1.4 Compiling

Running `$ populus compile` will compile all of the project contracts found in the `./contracts/` directory. The compiled assets are then written to `./build/contracts.json`.

Note: Populus currently only supports compilation of Solidity contracts.

1.4.1 Basic Compilation

Basic usage to compile all of the contracts and libraries in your project can be done as follows.

```
$ populus compile
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

1.4.2 Watching

This command can be used with the flag `--watch/-w` which will automatically recompile your contracts when the source code changes.

```
$ populus compile --watch
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
Change detected in: contracts/Greeter.sol
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

1.4.3 Build Output

Output is serialized as `JSON` and written to `build/contracts.json` relative to the root of your project. It will be a mapping of your contract names to the compiled assets for that contract.

```
{
    "Greeter": {
        "abi": [
            {
                "constant": true,
                "inputs": [
                    {
                        "name": "name",
                        "type": "bytes"
                    }
                ],
                "name": "greet",
                "outputs": [
                    {
                        "name": "",
                        "type": "bytes"
                    }
                ],
                "type": "function"
            },
            {
                "constant": false,
                "inputs": [
                    {
```

```
                        "name": "_greeting",
                        "type": "string"
                    }
                ],
                "name": "setGreeting",
                "outputs": [],
                "type": "function"
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greet",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "type": "function"
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greeting",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "type": "function"
            },
            {
                "inputs": [],
                "type": "constructor"
            }
        ],
        "code": "0x...",
        "code_runtime": "0x...",
        "meta": {
            "compilerVersion": "0.3.5-9da08ac3",
            "language": "Solidity",
            "languageVersion": "0"
        },
        "source": null
    }
}
```


1.5 Testing

1.5.1 Introduction

The Populus framework provides some powerful utilities for testing your contracts. Testing in Populus is powered by the python testing framework `py.test`.

All tests are run against an in-memory blockchain from `pyethereum.tester`.

The convention for tests is to place them in the `./tests/` directory in the root of your project. In order for `py.test` to find your tests modules their module name must start with `test_`.

Running Tests With Pytest

To run the full test suite of your project:

```
$ py.test tests/
```

Or to run a specific test

```
$ py.test tests/test_greeter.py
```

1.5.2 Fixtures

The test fixtures provided by populus are what make testing easy. In order to use a fixture in your tests all you have to do add an argument with the same name to the signature of your test function.

Project

- `project`

The Project object for your project.

```
def test_project_things(project):
    # directory things
    assert project.project_dir == '/path/to/my/project'

    # raw compiled contract access
    assert 'MyContract' in project.compiled_contracts
```

Unmigrated Chain

Warning: This fixture has been removed as part of the deprecation of the migrations API. You should instead use the `chain` fixture.

- `unmigrated_chain`

The `'tester'` test chain. This chain will not have had migrations run.

```
def test_greeter(unmigrated_chain):
    greeter = unmigrated_chain.get_contract('Greeter')

    assert greeter.call().greet() == "Hello"

def test_deploying_greeter(unmigrated_chain):
    GreeterFactory = unmigrated_chain.get_contract_factory('Greeter')
    deploy_txn_hash = GreeterFactory.deploy()
    ...
```

Chain

- `chain`

The same chain from the `unmigrated_chain` fixture except it has had all migrations run ion it.

```
def test_greeter(chain):
    greeter = chain.get_contract('Greeter')

    assert greeter.call().greet() == "Hello"

def test_deploying_greeter(chain):
    GreeterFactory = chain.get_contract_factory('Greeter')
    deploy_txn_hash = GreeterFactory.deploy()
    ...
```

Web3

- `web3`

A Web3.py instance configured to connect to `chain` fixture.

```
def test_account_balance(web3, chain):
    initial_balance = web3.eth.getBalance(web3.eth.coinbase)
    wallet = chain.get_contract('Wallet')

    withdraw_txn_hash = wallet.transact().withdraw(12345)
    withdraw_txn_receipt = chain.wait.for_receipt(withdraw_txn_hash)
    after_balance = web3.eth.getBalance(web3.eth.coinbase)

    assert after_balance - initial_balance == 1234
```

Contracts

Warning: This fixture has been renamed to `base_contract_factories`. In future releases of populus this fixture will be removed or repurposed.

- `contracts`

Base Contract Factories

- `base_contract_factories`

The contract factory classes for your project. These will all be associated with the Web3 instance from the `web3` fixture.

```
def test_wallet_deployment(web3, base_contract_factories):
    WalletFactory = base_contract_factories.Wallet

    deploy_txn_hash = WalletFactory.deploy()
```

Note: For contracts that have library dependencies, you should use the `Chain.get_contract_factory(...)` api. The contract factories from the `base_contract_factories` fixture will not be returned with linked bytecode. The ones from `Chain.get_contract_factory()` are returned fully linked.

Accounts

- `accounts`

The `web3.eth.accounts` property off of the `web3` fixture

```
def test_accounts(web3, accounts):
    assert web3.eth.coinbase == accounts[0]
```

1.5.3 Custom Fixtures

The built in fixtures for accessing contracts are useful for simple contracts, but this is often not sufficient for more complex contracts. In these cases you can create you own fixtures to build on top of the ones provided by Populus.

One common case is a contract that needs to be given constructor arguments. Lets make a fixture for a token contract that requires a constructor argument to set the initial supply.

```
import pytest

@pytest.fixture()
def token_contract(chain):
    TokenFactory = chain.get_contract_factory('Token')
    deploy_txn_hash = TokenFactory.deploy(arguments=[
        1e18,  # initial token supply
    )
    contract_address = chain.wait.for_contract_address(deploy_txn_hash)
    return TokenFactory(address=contract_address)
```

Now, you can use this fixture in your tests the same way you use the built-in populus fixtures.

```
def test_initial_supply(token_contract):
    assert token_contract.call().totalSupply() == 1e18
```

1.6 Deploy

1.6.1 Introduction

The deployment functionality exposed by Populus is meant for one-off deployments of simple contracts. The deployment process includes some, or all of the following steps.

1. Selection of which chain should be deployed to.
2. Running the given chain.
3. Compilation of project contracts.
4. Derivation of library dependencies.
5. Library linking.
6. Individual contract deployment.

Note: The command line deployment command cannot be used to deploy contracts which require constructor arguments.

1.6.2 Deploying A Contract

Deployment is handled through the `$ populus deploy` command.

Lets deploy a simple Wallet contract. First we'll need a contract in our project `./contracts` directory.

```
// ./contracts/Wallet.sol
contract Wallet {
        mapping (address => uint) public balanceOf;

        function deposit() {
                balanceOf[msg.sender] += 1;
        }

        function withdraw(uint value) {
                if (balanceOf[msg.sender] < value) throw;
                balanceOf[msg.sender] -= value;
                if (!msg.sender.call.value(value)()) throw;
        }
}
```

We can deploy this contract to a local test chain like this.

```
    $ populus deploy Wallet -c local_a
    Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0␣
↪because of library dependencies).

    Wallet
    Deploying Wallet
    Deploy Transaction Sent:␣
↪0x29e90f07314db495989f03ca931088e1feb7fb0fc13286c1724f11b2d6b239e7
    Waiting for confirmation...

    Transaction Mined
    =================
    Tx Hash      : 0x29e90f07314db495989f03ca931088e1feb7fb0fc13286c1724f11b2d6b239e7
    Address      : 0xb6fac5cb309da4d984bb6145078104355ece96ca
    Gas Provided : 267699
    Gas Used     : 167699


    Verifying deployed bytecode...
```

```
    Verified contract bytecode @ 0xb6fac5cb309da4d984bb6145078104355ece96ca matches␣
↪expected runtime bytecode
Deployment Successful.
```

Above you can see the output for a basic deployment.

1.7 Migrations

Warning: The migrations API is pending deprecation.

1.8 Project

1.8.1 Introduction

The Project API is the common entry point to all aspects of your populus project.

1.8.2 Basic Usage

- `Project(config_file_path=None)`

When instantaited with no arguments, the project will look for a `populus.json` file found in the current working directory and load that if found.

```
from populus.project import Project
# loads local `populus.json` file (if present)
project = Project()

# loads the specified config file
other_project = Project('/path/to/other/populus.json')
```

The project object is the entry point for almost everything that populus can do.

```
>>> project.project_dir
'/path/to/your-project`
>>> project.contracts_dir
'./contracts'
>>> project.config
{....}  # Your project configuration.
>>> project.compiled_contracts
{
  'Greeter': {
    'code': '0x...',
```

```
    'code_runtime': '0x...',
    'abi': [...],
    ...
  },
  ...
}
>>> with p.get_chain('temp') as chain:
...     print(chain.web3.eth.coinbase)
...
0x4949dce962e182bc148448efa93e73c6ba163f03
```

1.8.3 Chain API

- `get_chain(chain_name, chain_config=None)`

  Returns a `populus.chain.Chain` instance. You may provide `chain_config` in which case the chain will be configured using the provided configuration rather than the declared configuration for this chain from your configuration file.

  The returned `Chain` instance can be used as a context manager.

1.9 Configuration

1.9.1 Introduction

Populus is designed to be highly configurable through the project configuration file. By default, populus will load the file name `populus.json` from the root of your project.

The `$ populus init` command will write the full default configuration.

What you can Configure

This config file controls many aspects of populus that are configurable. Currently the config file controls the following things.

- Project root directory
- Contract source file location
- Compiler settings
- Available chains and how web3 connects to them.

Compiler Configuration

The following configuration options are available to control how populus compiles your project contracts.

```
{
  "compilation": {
    "contracts_dir": "./path/to/contract-source-files",
    "settings": {
      "optimize": true,
      "optimize_runs": 100
    }
  }
}
```

Contract Source Directory

The directory that project source files can be found in.

- key: `compilation.contracts_dir`
- value: Filesystem path
- default: `'./contracts'`

Compiler Settings

Enable or disable compile optimization.

- key: `compilation.settings.optimize`
- value: Boolean
- default: `True`

Chains

The `chains` key within the configuration file declares what chains populus has access to and how to connect to them. Populus comes pre-configured with the following chains.

- `'mainnet'`: Connects to the public ethereum mainnet via `geth`.
- `'ropsten'`: Connects to the public ethereum ropsten testnet via `geth`.
- `'tester'`: Uses an ephemeral in-memory chain backed by pyethereum.
- `'testrpc'`: Uses an ephemeral in-memory chain backed by pyethereum.
- `'temp'`: Local private chain whos data directory is removed when the chain is shutdown. Runs via `geth`.

```
{
  "chains": {
    "my-chain": {
      ... // The chain settings.
    }
  }
}
```

Individual Chain Settings

Each key and value in the `chains` portion of the configuration corresponds to the name of the chain and the settings for that chain. Each chain has two primary sections, `web3` and `chain` configuration settings.

```
{
  "chains": {
    "my-chain": {
      "chain": {
        "class": "populus.chain.LocalGethChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider"
        }
      }
    }
  }
}
```

The above chain configuration sets up a new local private chain within your project. The chain above would set it's data directory to `<project-dir>/chains/my-chain/`.

To simplify configuration of chains you can use the `ChainConfig` object.

```
>>> from populus.config import ChainConfig
>>> chain_config = ChainConfig()
>>> chain_config.set_chain_class('local')
>>> chain_config['web3'] = web3_config  # see below for the Web3Config object
>>> project.config['chains.my-chain'] = chain_config
```

The `set_chain_class()` method can take any of the following values.

- **These strings**
  - `chain_config.set_chain_class('local') => 'populus.chain.LocalGethChain'`
  - `chain_config.set_chain_class('external') => 'populus.chain.ExternalChain'`
  - `chain_config.set_chain_class('tester') => 'populus.chain.TesterChain'`
  - `chain_config.set_chain_class('testrpc') => 'populus.chain.TestRPCChain'`
  - `chain_config.set_chain_class('temp') => 'populus.chain.TemporaryGethChain'`
  - `chain_config.set_chain_class('mainnet') => 'populus.chain.MainnetChain'`
  - `chain_config.set_chain_class('testnet') => 'populus.chain.TestnetChain'`
  - `chain_config.set_chain_class('ropsten') => 'populus.chain.TestnetChain'`
- **Full python paths to the desired chain class.**
  - `chain_config.set_chain_class('populus.chain.LocalGethChain') => 'populus.chain.LocalGethChain'`
  - `chain_config.set_chain_class('populus.chain.ExternalChain') => 'populus.chain.ExternalChain'`
  - `...`
- **The actual chain class.**
  - `chain_config.set_chain_class(LocalGethChain) => 'populus.chain.LocalGethChain'`
  - `chain_config.set_chain_class(ExternalChain) => 'populus.chain.ExternalChain'`
  - `...`

Chain Class Settings

Determines which chain class will be used for the chain.

- key: `chains.<chain-name>.chain.class`
- value: Dot separated python path to the chain class that should be used.
- required: Yes

Available options are:

- `populus.chain.ExternalChain`

  A chain that populus does not manage or run. This is the correct class to use when connecting to a node that is already running.

- `populus.chain.TestRPCChain`

  An ephemeral chain that uses the python `eth-testrpc` package to run an in-memory ethereum blockchain. This chain will spin up an HTTP based RPC server.

- `populus.chain.TesterChain`

  An ephemeral chain that uses the python `eth-testrpc` package to run an in-memory ethereum blockchain. This chain **must** be used in conjunction with a web configuration using the provider `EthereumTesterProvider`.

- `populus.chain.LocalGethChain`

  A geth backed chain which will setup it's own data directory in the `./chains` directory in the root of your project.

- `populus.chain.TemporaryGethChain`

  An ephemeral chain backed by `geth` which uses a temporary directory as the data directory which is removed when the chain is shutdown.

- `populus.chain.TestnetChain`

  A `geth` backed chain which connects to the public Ropsten test network.

- `populus.chain.MainnetChain`

  A `geth` backed chain which connects to the main public network.

Web3

Configuration for the Web3 instance that will be used with this chain. See *Web3 Configuration* for more details.

- key: `chains.<chain-name>.web3`
- value: Web3 Configuration
- required: Yes

1.9.2 Web3 Configuration

Configuration for setting up a Web3 instance.

```
{
  "provider": {
    "class": "web3.providers.ipc.IPCProvider",
    "settings": {
      "ipc_path": "/path/to/geth.ipc"
    }
  }
  "eth": {
    "default_account": "0xd3cda913deb6f67967b99d67acdfa1712c293601",
  }
}
```

In order to simplify configuring Web3 instances you can use the `Web3Config` class.

```
>>> from populus.config import Web3Config
>>> web3_config = Web3Config()
>>> web3_config.set_provider('ipc')
>>> web3_config.provider_kwargs['ipc_path'] = '/path/to/geth.ipc'
>>> web3.config.default_account = '0x0000000000000000000000000000000000000001'
>>> project.config['chains.my-chain.web3'] = web3_config
>>> project.write_config()  # optionally persist the configuration to disk
```

Provider Class

Specifies the import path for the provider class that should be used.

- key: `provider.class`
- value: Dot separated python path
- required: Yes

Provider Settings

Specifies the `**kwargs` that should be used when instantiating the provider.

- key: `provider.settings`
- value: Key/Value mapping

Default Account

If present the `web3.eth.defaultAccount` will be populated with this address.

- key: `eth.default_account`
- value: Ethereum Address

1.9.3 Configuration API

The project configuration can be accessed as a property on the `Project` object via `project.config`. This object is a dictionary-like object with some added convenience APIs.

Project configuration is represented as a nested key/value mapping.

Getting and Setting

The `project.config` object exposes the following API for getting and setting configuration values. Supposing that the project configuration file contained the following data.

```
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
```

```
  'g': {
    'h': {
      'i': 'j',
      'k': 'l'
    }
  }
}
```

The config object supports retrieval of values in much the same manner as a dictionary. For convenience, you can also access *deep* nested values using a single key which is dot-separated combination of all keys.

```
>>> project.config.get('a')
{
  'b': {
    'c': 'd',
    'e': 'f'
  }
}
>>> project.config['a']
{
  'b': {
    'c': 'd',
    'e': 'f'
  }
}
>>> project.config.get('a.b')
{
  'c': 'd',
  'e': 'f'
}
>>> project.config['a.b']
{
  'c': 'd',
  'e': 'f'
}
>>> project.config.get('a.b.c')
'd'
>>> project.config['a.b.c']
'd'
>>> project.config.get('a.b.x')
None
>>> project.config['a.b.x']
KeyError: 'x'
>>> project.config.get('a.b.x', 'some-default')
'some-default'
```

The config object also supports setting of values in the same manner.

```
>>> project.config['m'] = 'n'
>>> project.config
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
  'g': {
```

```
    'h': {
      'i': 'j',
      'k': 'l'
    }
  },
  'm': 'n'
}
>>> project.config['o.p'] = 'q'
>>> project.config
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
  'g': {
    'h': {
      'i': 'j',
      'k': 'l'
    }
  },
  'm': 'n'
  'o': {
    'p': 'q'
  }
}
```

Config objects support existence queries as well.

```
>>> 'a' in project.config
True
>>> 'a.b' in project.config
True
>>> 'a.b.c' in project.config
True
>>> 'a.b.x' in project.config
False
```

Config References

Sometimes it is useful to be able to re-use some configuration in multiple locations in your configuration file. This is where references can be useful. To reference another part of your configuration use an object with a single key of `$ref`. The value should be the full key path that should be used in place of the reference object.

```
{
  'a': {
    '$ref': 'b.c'
  }
  'b': {
    'c': 'd'
  }
}
```

In the above, the key `a` is a reference to the value found under key `b.c`

```
>>> project.config['a']
['d']
>>> project.config.get('a')
['d']
```

1.9.4 Defaults

Populus ships with many defaults which can be overridden as you see fit.

Built-in defaults

Populus ships with the following *default* configuration.

```
{
  "version": "1",
  "chains": {
    "mainnet": {
      "chain": {
        "class": "populus.chain.MainnetChain"
      },
      "web3": {
        "$ref": "web3.GethIPC"
      }
    },
    "ropsten": {
      "chain": {
        "class": "populus.chain.TestnetChain"
      },
      "web3": {
        "$ref": "web3.GethIPC"
      }
    },
    "temp": {
      "chain": {
        "class": "populus.chain.TemporaryGethChain"
      },
      "web3": {
        "$ref": "web3.GethIPC"
      }
    },
    "tester": {
      "chain": {
        "class": "populus.chain.TesterChain"
      },
      "web3": {
        "$ref": "web3.Tester"
      }
    },
    "testrpc": {
      "chain": {
        "class": "populus.chain.TestRPCChain"
      },
      "web3": {
        "$ref": "web3.TestRPC"
      }
```

```
    }
  },
  "compilation": {
    "contracts_dir": "./contracts",
    "settings": {
      "optimize": true
    }
  },
  "web3": {
    "GethIPC": {
      "provider": {
        "class": "web3.providers.ipc.IPCProvider"
      }
    },
    "InfuraMainnet": {
      "eth": {
        "default_account": "0x0000000000000000000000000000000000000001"
      },
      "provider": {
        "class": "web3.providers.rpc.HTTPProvider",
        "settings": {
          "endpoint_uri": "https://mainnet.infura.io"
        }
      }
    },
    "InfuraRopsten": {
      "eth": {
        "default_account": "0x0000000000000000000000000000000000000001"
      },
      "provider": {
        "class": "web3.providers.rpc.HTTPProvider",
        "settings": {
          "endpoint_uri": "https://ropsten.infura.io"
        }
      }
    },
    "TestRPC": {
      "provider": {
        "class": "web3.providers.tester.TestRPCProvider"
      }
    },
    "Tester": {
      "provider": {
        "class": "web3.providers.tester.EthereumTesterProvider"
      }
    }
  }
}
```

It is recommended to use the `$ populus init` command to populate this file as it contains useful defaults.

Pre-Configured Web3 Connections

The following pre-configured configurations are available. To use one of the configurations on a chain it should be referenced like this:

```
{
  "chains": {
    "my-custom-chain": {
        "web3": {"$ref": "web3.GethIPC"}
    }
  }
}
```

GethIPC

Web3 connection which will connect to geth using an IPC socket.

- key: `web3.GethIPC`

InfuraMainnet

Web3 connection which will connect to the mainnet ethereum network via Infura.

- key: `web3.InfuraMainnet`

InfuraRopsten

Web3 connection which will connect to the ropsten ethereum network via Infura.

- key: `web3.InfuraRopsten`

TestRPC

Web3 connection which will use the `TestRPCProvider`.

- key: `web3.TestRPC`

Tester

Web3 connection which will use the `EthereumTesterProvider`.

- key: `web3.Tester`

1.10 Chains

1.10.1 Introduction

Populus has the ability to run a variety of blockchains for you, both programatically and from the command line.

Transient Chains

Populus can run two types of transient chains.

- `tester`

  A test EVM backed blockchain.

- `testrpc`

  Runs the `eth-testrpc` chain which implements the full JSON-RPC interface backed by a test EVM.

- `temp`

  Runs a blockchain backed by the go-ethereum `geth` client. This chain will use a temporary directory for it's chain data which will be cleaned up and removed when the chain shuts down.

Local Chains

Local chains can be setup within your `populus.json` file. Each local chain stores its chain data in the `populus.Project.blockchains_dir` and persists it's data between runs.

Local chains are backed by the go-ethereum `geth` client.

Public Chains

Populus can run both the main and ropsten public chains.

- `mainnet`

  With `$ populus chain run mainnet` populus will run the the go-ethereum client for you connected to the main public ethereum network.

- `ropsten`

  With `$ populus chain run ropsten` populus will run the the go-ethereum client for you connected to the ropsten testnet public ethereum network.

1.10.2 Running from the command line

The `$ populus chain` command handles running chains from the command line.

```
$ populus chain
Usage: populus chain [OPTIONS] COMMAND [ARGS]...

  Manage and run ethereum blockchains.

Options:
  -h, --help  Show this message and exit.

Commands:
  reset  Reset a chain removing all chain data and...
  run    Run the named chain.
```

1.10.3 Running programatically from code

The `populus.Project.get_chain(chain_name, chain_config=None)` method returns a `populus.chain.Chain` instance that can be used within your code to run any populus chain.

Lets look at a basic example of using the `temp` chain.

```
>>> from populus import Project
>>> project = Project()
>>> with project.get_chain('temp') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...
...
coinbase: 0x16e11a86ca5cc6e3e819efee610aa77d78d6e075
>>>
>>> with project.get_chain('temp') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...
...
coinbase: 0x64e49c86c5ad1dd047614736a290315d415ef28e
```

You can see that each time a `temp` chain is instantiated it creates a new data directory and generates new keys.

The `testrpc` chain operates in a similar manner in that each time you run the chain the EVM data is fully reset. The benefit of the `testrpc` server is that it starts quicker, and has mechanisms for manually resetting the chain.

Here is an example of running the `tester` blockchain.

```
>>> from populus import Project
>>> project = Project()
>>> with project.get_chain('tester') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     chain.mine()
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     snapshot_id = chain.snapshot()
...     print('Snapshot:', snapshot_id)
...     chain.mine()
...     chain.mine()
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     chain.revert(snapshot_id)
...     print('blockNumber:', chain.web3.eth.blockNumber)
...
coinbase: 0x82a978b3f5962a5b0957d9ee9eef472ee55b42f1
blockNumber: 1
```

```
blockNumber: 2
Snapshot: 0
blockNumber: 4
blockNumber: 2
```

The `testrpc` chain can be run in the same manner.

1.10.4 Access To Contracts

All chain objects present the following API for interacting with your project contracts.

- `get_contract_factory(contract_name, link_dependencies=None, validate_bytecode=True)`

  Returns the contract factory for the contract indicated by `contract_name` from the chain's `compiled_contracts`.

  If provided, `link_dependencies` should be a dictionary that maps library names to their on chain addresses that will be used during bytecode linking.

  If truthy (the default), `validate_bytecode` indicates whether the bytecode for any library dependencies for the given contract should be validated to match the on chain bytecode.

  If your project has no project migrations then the data used for these contract factories will come directly from the compiled project contracts.

  If your project has migrations then the data used to build your contract factories will be populutated as follows.:

  1. The newest migration that has been run which deploys the requested contract.
  2. The newest migration which contains this contract in it's `compiled_contracts` property
  3. The compiled project contracts.

- `get_contract(contract_name, link_dependencies=None, validate_bytecode=True)`

  Returns the contract instance indicated by the `contract_name` from the chain's `compiled_contracts`.

  The `link_dependencies` argument behaves the same was as specified in the `get_contract_factory` method.

  The `validate_bytecode` argument behaves the same way as specified in the `get_contract_factory` with the added condition that the bytecode for the requested contract will also be checked.

  Note: When using a `TestRPCChain` the `get_contract` method will lazily deploy your contracts for you. This lazy deployment will only work for simple contracts which do not require constructor arguments.

- `is_contract_available(contract_name, link_dependencies=None, validate_bytecode=True, raise_on_error=False)`

  Returns `True` or `False` as to whether the contract indicated by `contract_name` from the chain's `compiled_contracts` is available through the `Chain.get_contract` API.

  The `link_dependencies` argument behaves the same was as specified in the `get_contract_factory` method.

The `validate_bytecode` argument behaves the same way as specified in the `get_contract_factory` with the added condition that the bytecode for the requested contract will also be checked.

If `raise_on_error` is truthy, then the method will raise an exception instead of returning `False` for any of the failure cases.

1.10.5 Waiting for Things

Each chain object exposes the following API through a property `chain.wait`. The `timeout` parameter determines how long this will block before raising a `Timeout` exception. The `poll_interval` determines how long it should wait between polling. If `poll_interval == None` then `random.random()` will be used to determine the poling interval.

- `wait.for_contract_address(txn_hash, timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds returning the contract address from the transaction receipt for the given `txn_hash`.

- `wait.for_receipt(txn_hash, timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds returning the transaction receipt for the given `txn_hash`.

- `wait.for_block(block_number=1, timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds waiting until the highest block on the current chain is at least `block_number`.

- `wait.for_unlock(account=web3.eth.coinbase, timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds waiting until the account specified by `account` is unlocked. If `account` is not provided, `web3.eth.coinbase` will be used.

- `wait.for_peers(peer_count=1, timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds waiting for the client to have at least `peer_count` peer connections.

- `wait.for_syncing(timeout=120, poll_interval=None)`

  Blocks for up to `timeout` seconds waiting the chain to begin syncing.

1.11 Release Notes

1.11.1 1.5.0

- Remove gevent dependency
- Mark migrations API for deprecation.
- Mark unmigrated_chain testing fixture for deprecation.
- Mark `contracts` fixture for deprecation. Replaced by `base_contract_factories` fixture.
- Deprecate and remove old `populus.ini` configuration scheme.
- Add new configuration API.

1.11.2 1.4.2

- Upstream version bumps for web3 and ethtestrpc
- Change to use new web3.providers.tester.EthereumTesterProvider for test fixtures.

1.11.3 1.4.1

- Stop-gap fix for race-condition error from upstream: https://github.com/pipermerriam/web3.py/issues/80

1.11.4 1.4.0

- Contract source directory now configurable via populus.ini file.
- Updates to upstream dependencies.

1.11.5 1.3.0

- Bugfix for geth data_dir directory on linux systems.

1.11.6 1.2.2

- Support solc 0.4.x

1.11.7 1.2.1

- Support legacy JSON-RPC spec for `eth_getTransactionReceipt` in wait API.

1.11.8 1.2.0

- All function in the `chain.wait` api now take a `poll_interval` parameter which controls how aggressively they will poll for changes.
- The `project` fixture now caches the compiled contracts across test runs.

1.11.9 1.1.0

This release begins the first deprecation cycle for APIs which will be removed in future releases.

- Deprecated: Entire migrations API
- New configuration API which replaces the `populus.ini` based configuration.
- Removal of `gevent` as a required dependency. Threading and other asynchronous operations now default to standard library tools with the option to enable the gevent with an environment variable `THREADING_BACKEND==gevent`

1.11.10 1.0.0

This is the first release of populus that should be considered stable.

- Remove `$ populus web` command
- Remove `populus.solidity` module in favor of `py-solc` package for solidity compilation.
- Remove `populus.geth` module in favor of `py-geth` for running geth.
- Complete refactor of pytest fixtures.
- Switch to `web3.py` for all blockchain interactions.
- Compilation: - Remove filtering. Compilation now always compiles all contracts. - Compilation now runs with optimization turned on by default. Can be disabled with `--no-optimizie`. - Remove use of `./project-dir/libraries` directory. All contracts are now expected to reside in the `./project-dir/contracts` directory.
- New `populus.Project` API.
- New Migrations API: - `$ populus chain init` for initializing a chain with the Registrar contract. - `$ populus makemigration` for creating migration files. - `$ populus migrate` for executing migrations.
- New configuration API: - New commands `$ populus config`, `$ populus config:set` and `$ populus config:unset` for managing configuratino.
- New Chain API: - Simple programatic running of project chains. - Access to `web3.eth.contract` objects for all project contracts. - Access to pre-linked code based on previously deployed contracts.

1.11.11 0.8.0

- Removal of the `--logfile` command line argument. This is a breaking change as it will break when used with older installs of `geth`.

1.11.12 0.7.5

- Bugfix: `populus init` now creates the `libraries` directory
- Bugfix: `populus compile --watch` no longer fails if the `libraries` directory isn't present.

1.11.13 0.7.4

- Bugfix for the `geth_accounts` fixture.
- Bugfix for project initialization fixtures.
- Allow returning of `indexed` event data from Event.get_log_data
- Fix EthTesterClient handling of TransactionErrors to allow continued EVM interactions.
- Bugfix for long Unix socket paths.
- Enable whisper when running a geth instance.
- Better error output from compile errors.
- Testing bugfixes.

1.11.14 0.7.3

- Add `denoms` pytest fixture
- Add `accounts` pytest fixture
- Experimental synchronous function calls on contracts with `function.s(...)`
- Bugfixes for function group argument validation.
- Bugfixes for error handling within EthTesterClient
- Inclusion of Binary Runtime in compilation
- Fixes for tests that were dependent on specific solidity versions.

1.11.15 0.7.2

- Make the ethtester client work with asynchronous code.

1.11.16 0.7.1

- Adds `ipc_client` fixture.

1.11.17 0.7.0

- When a contract function call that is supposed to return data returns no data an error was thown. Now a custom exception is thrown. This is a breaking change as previously for addresses this would return the empty address.

1.11.18 0.6.6

- Actually fix the address bug.

1.11.19 0.6.5

- Fix bug where addresses were getting double prefixed with `0x`

1.11.20 0.6.3

- Bugfix for Event.get_log_data
- Add `get_code` and `get_accounts` methods to EthTesterClient
- Add `0x` prefixing to addresses returned by functions with multiple return values.

1.11.21 0.6.3

- Shorted path to cli tests to stay under 108 character limit for unix sockets.
- Adds tracking of contract addresses deployed to test chains.
- New `redeploy` feature available within `populus attach` as well as notification that your contracts have changed and may require redeployment.

1.11.22 0.6.2

- Shorted path to cli tests to stay under 108 character limit for unix sockets.
- Allow passing `--verbosity` tag into `populus chain run`
- Expand documentation with example use case for populus deploy/chain/attach commands.

1.11.23 0.6.1

- Change the *default* gas for transactions to be a percentage of the max gas.

1.11.24 0.6.0

- **Improve `populus deploy` command.**
  - Optional dry run to test chain
  - Prompts user for confirmation on production deployments.
  - Derives gas needs based on dry-run deployment.
- Addition of `deploy_coinbase` testing fixture.
- Renamed `Contract._meta.rpc_client` to be `Contract._meta.blockchain_client` to be more appropriately named since the `EthTesterClient` is not an RPC client.
- Renamed `rpc_client` argument to `blockchain_client` in all relevant functions.
- Moved `get_max_gas` function onto blockchain clients.
- Moved `wait_for_transaction` function onto blockchain clients.
- Moved `wait_for_block` function onto blockchain clients.
- Bugfix when decoding large integers.
- Reduced `gasLimit` on genesis block for test chains to `3141592`.
- Updated dependencies to newer versions.

1.11.25 0.5.4

- Additional support for *library* contracts which will be included in compilation.
- `deployed_contracts` automatically derives deployment order and dependencies as well as linking library addresses.
- `deployed_contracts` now comes with the transaction receipts for the deploying transaction attached.
- Change to use `pyethash` from pypi

1.11.26 0.5.3

- New `populus attach` command for launching interactive python repl with contracts and rpc client loaded into local scope.
- Support for auto-linking of library contracts for the `deployed_contracts` testing fixture.

1.11.27 0.5.2

- Rename `rpc_server` fixture to `testrpc_server`
- Introduce `populus_config` module level fixture which holds all of the default values for other populus module level fixtures that are configurable.
- Add new configuration options for `deployed_contracts` fixture to allow declaration of which contracts are deployed, dependency ordering and constructor args.
- Improve overall documentation around fixtures.

1.11.28 0.5.1

- Introduce the `ethtester_client` which has the same API as the eth_rpc_client.Client class but interacts directly with the `ethereum.tester` module
- Add ability to control the manner through which the `deployed_contracts` fixture communicates with the blockchain via the `deploy_client` fixture.
- Re-organization of the contracts module.
- Support for multiple contract functions with the same name.
- Basic support for extracting logs and log data from transactions.

1.11.29 0.5.0

- Significant refactor to the `Contract` and related `Function` and `Event` objects used to interact with contracts.
- Major improvements to robustness of `geth_node` fixture.
- `deployed_contracts` testing fixture no longer provides it's own rpc server. Now you must either provide you own, or use the `geth_node` or `rpc_server` alongside it in tests.
- `geth_node` fixture now writes to a logfile located in `./chains/<chain-name>/logs/` for both cli and test case runs.

1.11.30 0.4.3

- Add support for address function args with a 0x prefix.

1.11.31 0.4.2

- Add `init` command for initializing a populus project.

1.11.32 0.4.1

- Missing `index.html` file.

1.11.33 0.4.0

- **Add blockchain management via `populus chain` commands which wraps `geth` library.**
  - `populus chain run <name>` for running the chain
  - `populus chain reset <name>` for resetting a chain
- **Add html/css/js development support.**
  - Development webserver via `populus web runserver`
  - Conversion of compiled contracts to web3 contract objects in javascript.

1.11.34 0.3.7

- Add support for decoding multiple values from a solidity function call.

1.11.35 0.3.6

- Add support for decoding `address``` return types from contract functions.

1.11.36 0.3.5

- Add support for contract constructors which take arguments via the new `constructor_args` parameter to the `Contract.deploy` method.

1.11.37 0.3.4

- Fix bug where null bytes were excluded from the returned bytes.

1.11.38 0.3.3

- Fix a bug in the `sendTransaction` methods for contract functions that did not pass along most of the `**kwargs`.
- Add new `Contract.get_balance()` method to contracts.

1.11.39 0.3.2

- Enable decoding of `bytes` types returned by contract function calls.

1.11.40 0.3.1

- Enable decoding of `boolean` values returned by contract function calls.

1.11.41 0.3.0

- Removed `watch` command in favor of passing `--watch` into the `compile` command.
- Add granular control to the `compile` command so that you can specify specific files, contract names, or a combination of the two.

1.11.42 0.2.0

- Update to `pypi` version of `eth-testrpc`
- Add new watch command which observes the project contracts and recompiles them when they change.
- Improved shell output for compile command.
- Re-organized portions of the `utils` module into a new `compilation` module.

1.11.43 0.1.4

- Fix broken import in `cli` module.

1.11.44 0.1.3

- Remove the local RPC client in favor of using https://github.com/pipermerriam/ethereum-rpc-client

1.11.45 0.1.2

- Add missing pytest dependency.

1.11.46 0.1.1

- Fix bug when deploying contracts onto a real blockchain.

1.11.47 0.1.0

- Project Creation

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P