## Contents

1. Contents .................................................. 3
2. Indices and tables ......................................... 59
   Python Module Index ..................................... 61
Populus is a smart contract development framework for the Ethereum blockchain.
Quickstart

- System Dependencies
  - Debian, Ubuntu, Mint
  - Fedora, CentOS, RedHat
  - OSX
- Installation
- Initializing a new project
- Compiling your contracts
- Testing your contract
- Setup for development and contribution
  - Virtual environment
  - Install dependencies
  - Install Solidity
  - Confirm

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
```

**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
```

**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`. 
Compiling your contracts

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

```solidity
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

```bash
$ 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
```

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.

```python
def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')
    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

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

```bash
$ 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.

### Setup for development and contribution

In order to configure the project locally and get the whole test suite passing, you’ll need to make sure you’re using the proper version of the solc compiler. Follow these steps to install all the dependencies:

#### Virtual environment

If you don’t already have it, go ahead and install `virtualenv` with `pip install virtualenv`. You can then create and activate your Populus environment with the following commands:

```bash
$ cd populus
$ virtualenv populus
$ source populus/bin/activate
```

This allows you to install the specific versions of the Populus dependencies without conflicting with global installations you may already have on your machine.

#### Install dependencies

Now, run the following commands to install all the dependencies specified in the project except for solc:

```bash
$ pip install -r requirements-dev.txt
$ pip install -r requirements-docs.txt
$ pip install -r requirements-gevent.txt
$ pip install -e .
```

#### Install Solidity

Here’s where the fun begins: you’ll have to build Solidity from source, and it specifically needs to be the `release_0.4.8` branch. Here’s how to do that:

First, clone the repository and switch to the proper branch:

```bash
$ git clone --recursive https://github.com/ethereum/solidity.git
$ cd solidity
$ git checkout release_0.4.8
```
If you’re on a Mac, you may need to accept the Xcode license as well. Make sure you have the latest version installed, and if you run into errors, try the following:

```bash
$ sudo xcodebuild -license accept
```

If you’re on Windows, make sure you have Git, CMake, and Visual Studio 2015. Now, install all the external dependencies. For Mac:

```bash
$ ./scripts/install_deps.sh
```

Or, for Windows:

```bash
$ scripts\install_deps.bat
```

Finally, go ahead and build Solidity. For Mac:

```bash
$ mkdir build
$ cd build
$ cmake .. && make
```

Or, for Windows:

```bash
$ mkdir build
$ cd build
$ cmake -G "Visual Studio 14 2015 Win64" ..
```

The following command will also work for Windows:

```bash
$ cmake --build . --config RelWithDebInfo
```

**Confirm**

This should have installed everything you need, but let’s be sure. First, try running:

```bash
$ which solc
```

If you didn’t see any output, you’ll need to move the solc executable file into the directory specified in your PATH, or add an accurate PATH in your bash profile. If you can’t find the file, you may need to run:

```bash
$ npm install -g solc
```

This should install the executable wherever your Node packages live.

Once you see output from the `which solc` command (and you’re in the Populus directory with the virtualenv activated), you’re ready to run the tests:

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

At this point, all your tests should pass. If they don’t, you’re probably missing a dependency somewhere. Just retrace your steps and you’ll figure it out.

**Overview**

1.2. Overview

7
Introduction

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

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

Project Layout

By default Populus expects a project to be laid 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
Running `$ populus init` will initialize the current directory with the default project layout that populus uses.

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

Tutorial

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

Contents

Part 1: Basic Testing

- Introduction
- Modify our Greeter
- Testing our changes

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. Let’s add another test to ./tests/test_greeter.py so that the file looks as follows.

def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')
    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter, _ = chain.provider.get_or_deploy_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.provider.get_or_deploy_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**
- **Setting up a local chain**
- **Deploying the contract**

**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:

```
{
  "version": "3",
  "chains": {
    "horton": {
      "chain": {
        "class": "populus.chain.geth.LocalGethChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider"
        }
      },
      "contracts": {
        "backends": {
          "JSONFile": {"$ref": "contracts.backends.JSONFile"}
        }
      }
    }
  }
}
```
We have just setup the minimal configuration necessary to run a local chain named horton. You can run this chain now in your terminal with the following command.

```
$ populus chain run horton
```

You should see a lot 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: 0

Deploying Greeter
Deploy Transaction Sent:

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 allotted a lot of ether.

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

```json
{
    "version": "3",
    "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.

Part 3: Installing a Package

- Introduction
- Setting up the project folder

**Introduction**

In this tutorial we will be creating our own mintable ERC20 token. However, instead of writing our own ERC20 implementation we'll be taking advantage of an existing implementation through the use of populus's package management features.

**Setting up the project folder**

Create a new directory for your project and run $ populus project init to populate the initial project structure.

```
$ populus init
Wrote default populus configuration to `./populus.json`. <Paste>
Created Directory: ./contracts
Created Example Contract: ./contracts/Greeter.sol
Created Directory: ./tests
Created Example Tests: ./tests/test_greeter.py
```
Now, delete the ./contracts/Greeter.sol and ./tests/test_greeter.py files as we won’t be using the Greeter contracts in this tutorial.

Once you’ve removed those files create a new solidity source file ./contracts/MintableToken.sol and paste in the following solidity source code.

```solidity
pragma solidity ^0.4.0;

import {owned} from "example-package-owned/contracts/owned.sol";
import {StandardToken} from "example-package-standard-token/contracts/StandardToken.sol";

contract MintableToken is StandardToken(0), owned {
    function mint(address who, uint value) public onlyowner returns (bool) {
        balances[who] += value;
        totalSupply += value;
        Transfer(0x0, who, value);
        return true;
    }
}
```

If you are familiar with solidity, the two import statements should stand out to you. These two statements will currently cause an error during compilation. Let’s see.

```bash
$ populus compile
============ Compiling ==============
> Loading source files from: ./contracts
Traceback (most recent call last):
...
    > command: `solc --optimize --combined-json bin,bin-runtime,abi,devdoc,userdoc contracts/MintableToken.sol'
    > stderr: ^--------------------------------------------------------------^ contracts/MintableToken.sol:3:1: Error: Source "example-package-owned/contracts/owned.sol" not found: File not found.
```

The solidity compiler clearly gets angry that we’re trying to import files that don’t exist. In order to install these files and make solidity happy we’ll first need to generate a package manifest using the $ populus package init command.

You will be presented with an interactive prompt to populus various pieces of project information. There will now be a new file in the root of your project named ethpm.json that should look something like this.

```json
{
    "authors": [
        "Piper Merriam <pipermerriam@gmail.com>"
    ],
    "description": "Mintable ERC20 token contract",
}
```
Now we are ready to install some dependencies using the \$ populus package install command. We want to install both the example-package-owned and example-package-standard-token packages.

\$ populus package install example-package-owned example-package-standard-token

Installed Packages: owned, standard-token

If you look in your project directory you should also see a new folder ./installed_packages.

\$ tree .
.
- contracts
  | - MintableToken.sol
- ethpm.json
- installed_packages
  | - example-package-owned
  |   | - build_identifier.lock
  |   | - contracts
  |   |   | - owned.sol
  |   | - install_identifier.lock
  |   | - installed_packages
  |   | - lock.json
  - example-package-standard-token
    | - build_identifier.lock
    | - contracts
    |   | - AbstractToken.sol
    |   | - StandardToken.sol
    | - install_identifier.lock
    | - installed_packages
    | - lock.json
- populus.json
- tests
9 directories, 12 files

And if you look in your ethpm.json file you should see two dependencies.

```json
{
  "authors": ["Piper Merriam <pipermerriam@gmail.com>"],
  "dependencies": {
    "example-package-owned": "1.0.0",
    "example-package-standard-token": "1.0.0"
  },
  "description": "Mintable ERC20 token contract",
  "keywords": ["ERC20"]
}
```
Now, we can try to compile our project again and everything should work.

```
$ populus compile
============ Compiling ==============  
> Loading source files from: ./contracts  
> Found 1 contract source files  
- contracts/MintableToken.sol  
> Compiled 4 contracts  
- MintableToken  
- StandardToken  
- Token  
- owned  
> Wrote compiled assets to: ./build/contracts.json/contracts.json
```

Let's go ahead and write a quick test for our new minting functionality. Add the following test code to a new file ./tests/test_token_minting.py

```python
import pytest

def test_minting_tokens(chain, accounts):
    provider = chain.provider
    mintable_token, deploy_txn_hash = provider.get_or_deploy_contract('MintableToken',
        deploy_kwargs={"_totalSupply": 0},
    )

    assert mintable_token.call().balanceOf(accounts[0]) == 0
    assert mintable_token.call().balanceOf(accounts[1]) == 0
    assert mintable_token.call().totalSupply() == 0

    chain.wait.for_receipt(mintable_token.transact().mint(
        who=accounts[0],
        value=12345,
    ))
    chain.wait.for_receipt(mintable_token.transact().mint(
        who=accounts[1],
        value=54321,
    ))

    assert mintable_token.call().balanceOf(accounts[0]) == 12345
    assert mintable_token.call().balanceOf(accounts[1]) == 54321
    assert mintable_token.call().totalSupply() == 66666

def test_only_owner_can_mint(chain, accounts):
    provider = chain.provider
    mintable_token, deploy_txn_hash = provider.get_or_deploy_contract('MintableToken',
        deploy_kwargs={"_totalSupply": 0},
    )

    assert mintable_token.call().balanceOf(accounts[0]) == 0
    assert mintable_token.call().balanceOf(accounts[1]) == 0
    assert mintable_token.call().totalSupply() == 0

    chain.wait.for_receipt(mintable_token.transact().mint(
        who=accounts[0],
        value=12345,
    ))
    chain.wait.for_receipt(mintable_token.transact().mint(
        who=accounts[1],
        value=54321,
    ))

    assert mintable_token.call().balanceOf(accounts[0]) == 12345
    assert mintable_token.call().balanceOf(accounts[1]) == 54321
    assert mintable_token.call().totalSupply() == 66666
```
'MintableToken',
    deploy_kwargs={"_totalSupply": 0},
)

with pytest.raises(Exception):
    mintable_token.transact({'from': accounts[1]}).mint(
        who=accounts[0],
        value=12345,
    )

And you can the tests with the `py.test` command.

```
$ py.test tests/
======================== test session starts ========================
platform darwin -- Python 3.5.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0
rootdir: /Users/piper/sites/scratch/populus-tutorial-3, inifile: plugins: populus-1.5.0
collected 2 items
tests/test_token_minting.py ..
======================== 2 passed in 0.74 seconds ========================
```

Fin.

**Part 4: Publishing a Package**

- **Introduction**
- **Configuring Populus for Publishing**
- **Configuring your package for publishing**
- **Building the release lockfile**
- **Publishing the release lockfile**

**Introduction**

In the previous tutorial we explored installing packages and using the contracts from those packages in our project. This tutorial will pick up where that one left off. We will be publishing our `mintable-standard-token` package to The Package Registry.

**Configuring Populus for Publishing**

In order to publish our package you will need to add some configuration to the the `RopstenPackageIndexBackend` which can be found in the `populus.json` file in the root of the project. It should currently look like this.

```
"RopstenPackageIndexBackend": {
    "class": "populus.packages.backends.index.PackageIndexBackend",
    "priority": 40,
```

1.3. Tutorial
We’re going to add the key `web3-for-publish` to the `settings` portion of this config. Populus will need to be able to send transactions through the configured web3 instances. For the purposes of this tutorial you will need to run a `geth` node that is connected to the Ropsten testnetwork with an unlocked account. Modify the config to look like the following, but with your address substituted in place the address `0xaaffa9e11a8deac514b93169c764aa042b4fe316f` and the path to your `geth.ipc` file for the running ropsten instance.

```
"RopstenPackageIndexBackend": {
  "class": "populus.packages.backends.index.PackageIndexBackend",
  "priority": 40,
  "settings": {
    "package_index_address": "0x8011df4830b4f696cd81393997e5371b93338878",
    "web3-for-install": {
      "$ref": "web3.InfuraRopsten"
    }
  },
  "web3-for-publish": {
    "provider": {
      "class": "web3.providers.ipc.IPCProvider",
      "settings": {
        "ipc_path": "/Users/piper/Library/Ethereum/ropsten/geth.ipc"
      }
    },
    "eth": {
      "default_account": "0xaaffa9e11a8deac514b93169c764aa042b4fe316f"
    }
  }
}
```

### Configuring your package for publishing

The next thing you’ll need to do is rename your package to something other than `mintable-standard-token` as that package name is already registered on the package index. The package name is set in the `ethpm.json` file located in the root of the project.

### Building the release lockfile

To build the package we will use the `$ populus package build` command. We want to include our `MintableToken` contract in the release. Use the following command to build the release lockfile.

```
$ populus package build --contract-type MintableToken
Wrote release lock file: build/1.0.0.json
```

If you open up the built release lockfile `./build/1.0.0.json` you should see something similar to the following (which was truncated for readability sake).
Publishing the release lockfile

The last step is to publish the release lockfile. This is done with the \$ populus package publish command.

```
$ populus package publish build/1.0.0.json
Publishing to RopstenPackageIndexBackend
```

If you wait for the transaction to be confirmed and head over to The Package Registry you should see your newly
Published package in the package index.

**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.

**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
```

**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
```
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.

```json
{
    "Greeter": {
        "abi": [
            {
                "constant": false,
                "inputs": [
                    {
                        "name": "_greeting",
                        "type": "string"
                    }
                ],
                "name": "setGreeting",
                "outputs": [],
                "payable": false,
                "type": "function"
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greet",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "payable": false,
                "type": "function"
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greeting",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "payable": false,
                "type": "function"
            },
            {
                "inputs": [],
                "payable": false,
                "type": "constructor"
            }
        ],
        "bytecode": "0x6060604052....",
        "bytecode_runtime": "0x6060604052...."
    }
}
```
"metadata": {
  "compiler": {
    "version": "0.4.8+commit.60cc1668.Darwin.appleclang"
  },
  "language": "Solidity",
  "output": {
    "abi": [
      {
        "constant": false,
        "inputs": [
          {
            "name": "_greeting",
            "type": "string"
          }
        ],
        "name": "setGreeting",
        "outputs": [],
        "payable": false,
        "type": "function"
      },
      {
        "constant": true,
        "inputs": [],
        "name": "greet",
        "outputs": [
          {
            "name": "",
            "type": "string"
          }
        ],
        "payable": false,
        "type": "function"
      },
      {
        "constant": true,
        "inputs": [],
        "name": "greeting",
        "outputs": [
          {
            "name": "",
            "type": "string"
          }
        ],
        "payable": false,
        "type": "function"
      },
      {
        "inputs": [],
        "payable": false,
        "type": "constructor"
      }
    ],
    "devdoc": {
      "methods": {}
    },
    "userdoc": {
      "methods": {}
    }
  }
}
Configuration

The following configuration options can be set to control aspects of how Populus compiles your project contracts.

- `compilation.contracts_source_dir`

  Defaults to `./contracts`. This sets the root path where populus will search for contract source files.

- `compilation.settings.optimize`

  Defaults to `True`. Determines if the optimizer will be enabled during compilation.

Testing

Introduction

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

By default tests are run against an in-memory ethereum blockchain.

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_`.  

1.5. Testing 23
Test Contracts

Populus supports writing contracts that are specifically for testing. These contract filenames should match the glob pattern `Test*.sol` and be located anywhere under your project tests directory `./tests/`.

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
```

Pytest 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.

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

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

Chain

- `chain`

A running 'tester' test chain.

```python
def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

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

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

- registrar

Convenience fixture for the chain.registrar property.

Provider

- provider

Convenience fixture for the chain.provider property.

Web3

- web3

Convenience fixture for the chain.provider property. A Web3.py instance configured to connect to chain fixture.

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

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

    assert after_balance - initial_balance == 1234
```

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.

```python
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

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

```python
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.

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

Deploy

- Introduction
- Deploying A Contract with the Command Line
- Programmatically deploy a contract

Introduction

Populus provides a command line interface for contract deployments which is suitable for simple contract deployments which do not involve constructor arguments as well as APIs for performing more complex deployments using python.

Deploying A Contract with the Command Line

Deployment is handled through the `populus deploy` command. All of the following are handled automatically.

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.

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:
0x29e90f07314db495989f03ca931088e1feb7fb0f0c13286c1724f11b2d6b239e7
Waiting for confirmation...

Transaction Mined
Tx Hash : 0x29e90f07314db495989f03ca931088e1feb7fb0f0c13286c1724f11b2d6b239e7
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.

**Programmatically deploy a contract**

You can also deploy contracts using a Python script. This is a suitable method if your contracts take constructor arguments or need more complex initialization calls.

Example (deploy_testnet.py):

```python

###Deploy Edgeless token and smart contract in testnet.
A simple Python script to deploy contracts and then do a smoke test for them.

```from populus import Project
from populus.utils.wait import wait_for_transaction_receipt
from web3 import Web3

1.6. Deploy
def check_successful_tx(web3: Web3, txid: str, timeout=180) -> dict:
    """See if transaction went through (Solidity code did not throw).
    """
    receipt = wait_for_transaction_receipt(web3, txid, timeout=timeout)
    txinfo = web3.eth.getTransaction(txid)
    # EVM has only one error mode and it's consume all gas
    assert txinfo["gas"] != receipt["gasUsed"]
    return receipt

def main():
    project = Project()

    # This is configured in populus.json
    # We are working on a testnet
    chain_name = "ropsten"
    print("Make sure {} chain is running, you can connect to it, or you'll get timeout →".format(chain_name))

    with project.get_chain(chain_name) as chain:

        # Load Populus contract proxy classes
        Crowdsale = chain.get_contract_factory('Crowdsale')
        Token = chain.get_contract_factory('EdgelessToken')

        web3 = chain.web3
        print("Web3 provider is", web3.currentProvider)

        # The address who will be the owner of the contracts
        beneficiary = web3.eth.coinbase
        assert beneficiary, "Make sure your node has coinbase account created"

        # Random address on Ropsten testnet
        multisig_address = "0x83917f644df1319a6ae792bb24433332e65fff8"

        # Deploy crowdsale, open since 1970
        txhash = Crowdsale.deploy(transaction={"from": beneficiary},
        args=[beneficiary, multisig_address, 1])
        print("Deploying crowdsale, tx hash is", txhash)
        receipt = check_successful_tx(web3, txhash)
        crowdsale_address = receipt["contractAddress"]
        print("Crowdsale contract address is", crowdsale_address)

        # Deploy token
        txhash = Token.deploy(transaction={"from": beneficiary}, args=[beneficiary])
        print("Deploying token, tx hash is", txhash)
        receipt = check_successful_tx(web3, txhash)
        token_address = receipt["contractAddress"]
        print("Token contract address is", token_address)

        # Make contracts aware of each other
print("Initializing contracts")
crowdsale = Crowdsale(address=crowdsale_address)
token = Token(address=token_address)
txhash = crowdsale.transact({"from": beneficiary}).setToken(token_address)
check_suceesful_tx(web3, txhash)

# Do some contract reads to see everything looks ok
print("Token total supply is", token.call().totalSupply())
print("Crowdsale max goal is", crowdsale.call().maxGoal())
print("All done! Enjoy your decentralized future.")

if __name__ == "__main__":
    main()

See full source code repository example.

Project

- Introduction
- Basic Usage
- Configuration
- Chains

Introduction

class populus.project.BaseChain

The Project class is the primary entry point to all aspects of your populus project.

Basic Usage

- Project(config_file_path=None)

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

```python
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.

```ini
>>> project.project_dir
'/path/to/your-project'
```
>>> project.contracts_dir
'./contracts'

>>> project.config
{}
# Your project configuration.

>>> project.compiled_contract_data
{
    'Greeter': {
        'code': '0x...',
        'code_runtime': '0x...',
        'abi': [...],
        ...
    },
    ...
}

>>> with p.get_chain('temp') as chain:
...
    print(chain.web3.eth.coinbase)
...
0x4949dce962e182bc148448efa93e73c6ba163f03

**Configuration**

**Project.config**

Returns the current project configuration.

**Project.load_config()**

Loads the project configuration from disk, populating `Project.config`.

**Project.write_config()**

Writes the current project configuration from `Project.config` to disk.

**Chains**

**Project.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.

**Configuration**

- **Introduction**
  - What you can Configure
  - Compiler Configuration
    - Contract Source Directory
    - Compiler Backend
    - Import Remappings
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.
### Contract Source Directory

The directory that project source files can be found in.

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

### Compiler Backend

Set which compiler backend should be used

- **key**: `compilation.backend.class`
- **value**: Dot separated python path
- **default**: `populus.compilation.backends.SolcStandardJSONBackend`

**Settings for the compiler backend**

- **key**: `compilation.backend.settings`
- **value**: Object of configuration parameters for the compiler backend.
- **default**: `{ "optimize": true, "output_values": ["abi", "bin", "bin-runtime", "devdoc", "metadata", "userdoc"] }`

### Import Remappings

Set `solc import path remappings`. This is especially useful if you want to use libraries like [OpenZeppelin](https://github.com/OpenZeppelin) with your project. Then you can directly import Zeppelin contracts like `import "zeppelin/contracts/token/TransferableToken.sol"`.

- **key**: `compilation.import_remappings`
- **value**: Array of strings
- **default**: `[]`
- **example**: `["zeppelin=zeppelin"]` assuming that the root directory for the Zeppelin contracts is `./zeppelin` in the root of your project.
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 whose data directory is removed when the chain is shutdown. Runs via `geth`.

```json
{
    "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.

```json
{
    "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 its data directory to `<project-dir>/chains/my-chain/`.

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

```python
>>> 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

#### Web3 Configuration

Configuration for setting up a Web3 instance.

```python
{
    "provider": {
        "class": "web3.providers.ipc.IPCProvider",
        "settings": {
            "ipc_path": "/path/to/geth.ipc"
        }
    },
    "eth": {
        "default_account": "0xd3cda913deb6f67967b99d67acdfaf1712c293601",
    }
}
```

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

```python
>>> from populus.config import Web3Config
>>> web3_config = Web3Config()
>>> web3_config.set_provider('ipc')
>>> web3_config.provider_kwargs['ipc_path'] = '/path/to/geth.ipc'  # 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

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.

```json
{
'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.

```python
>>> '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.

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

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

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

**Defaults**

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

Populus ships with the following default configuration.
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:

```json
{
    "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
Command Line Interface

You can manage your configuration using the command line with the $ populus config command.

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

Manage and run ethereum blockchains.

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

Commands:
  delete Deletes the provided key/value pairs from the...
  get Gets the provided key/value pairs from the...
  list Prints the project configuration out to the...
  set Sets the provided key/value pairs in the...
```

To interact with nested keys simply separate them with a ..

```
$ populus config list
some.nested.key_a: the_value_a
some.nested.key_b: the_value_b
$ populus config set some.nested.key_c:the_value_c
$ populus config list
some.nested.key_a: the_value_a
some.nested.key_b: the_value_b
some.nested.key_c: the_value_c
$ populus config get some.nested.key_a
some.nested.key_a: the_value_a
$ populus config delete some.nested.key_a
some.nested.key_a: the_value_a (deleted)
```

Chains

Chains are how populus interacts with the Ethereum blockchain.

Introduction to Chains

- **Introduction**
  - Transient Chains
  - Local Chains
  - Public Chains
- **Running from the command line**
- **Running programatically from code**
Introduction

Populus has the ability to run and/or connect to 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.

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.
```
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.

Let's look at a basic example of using the `temp` chain.

```python
>>> from populus import Project
>>> project = Project()
>>> with project.get_chain('temp') 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: 0x16e11a86ca5cc6e3e819efee610aa77d78d6e075
blockNumber: 1
blockNumber: 2
Snapshot: 0
blockNumber: 4
blockNumber: 2
```

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.

```python
>>> 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)
...    chain.mine()
...    print('blockNumber:', chain.web3.eth.blockNumber)
...    chain.mine()
...    print('blockNumber:', chain.web3.eth.blockNumber)
...    chain.revert(snapshot_id)
...    print('blockNumber:', chain.web3.eth.blockNumber)

coinbase: 0x82a978b3f5962a5b0957d9ee9eef472ee55b42f1
blockNumber: 1
blockNumber: 2
blockNumber: 4
blockNumber: 2
```

Note: The `testrpc` chain can be run in the same manner.
class populus.chain.base.BaseChain

**Accessing Contracts**

The `BaseChain` object is the entry point for the Provider and Registrar APIs which collectively give access to your project contracts and related information.

- The Provider API gives access to both the raw compiler output, the contract factories and the deployed instances of your contracts. This api can be accessed from the `BaseChain.provider` property.
- The Registrar API records the addresses of deployed contract instances for later retrieval. This api can be accessed from the `BaseChain.registrar` property.

**Getting the raw compiled data**

To retrieve the contract data for a specific contract you will use the `BaseChain.provider.get_base_contract_factory()` method. Supposing that your project contained a contract named “Math” you could retrieve the contract data using the following code.

```python
>>> chain.provider.get_base_contract_factory('Math')
{
    'abi': [...],
    'bytecode': '0x...',
    'bytecode_runtime': '0x...',
    'metadata': {...},
}
```

You may also want to retrieve all of the contract data for your project. This can be done with the `BaseChain.provider.get_all_contract_data()` method.

```python
>>> chain.provider.get_all_contract_data()
{
    'Math': {'abi': [...], ...},
    'MyOtherContract': {'abi': [...], ...},
}
```

**Getting contract factories**

The `BaseChain.provider.get_contract_factory()` method gives you access to the contract factory classes for your contracts.

```python
>>> Math = chain.provider.get_contract_factory('Math')
>>> Math.abi
[...]
>>> Math.bytecode
"0x..."
>>> Math.bytecode_runtime
"0x..."
```

Contract factories returned by this method will be returned with their underlying bytecode linked against the appropriate library addresses. In the event that one of the underlying dependencies is not available a `NoKnownAddress` exception will be raised.
In some cases you may want the contract factory class without worrying about whether the underlying bytecode linking. Such contract factories are referred to as “base” contract factories and can be retrieved using the `BaseChain.provider.get_base_contract_factory()` method.

```python
>>> Math = chain.provider.get_base_contract_factory('Math')
>>> Math.abi
[...]
>>> Math.bytecode
"0x..." # <- may contain unlinked bytecode.
>>> Math.bytecode_runtime
"0x..." # <- may contain unlinked bytecode.
```

### Registering contract addresses

When you deploy an instance of a contract populus stores the contract address using the registry API. This is an API that you should rarely need to interact with directly as populus does the registration of new addresses automatically. To set the address for a contract manually you would use the `BaseChain.registrar.set_contract_address()` method.

```python
>>> chain.registrar.set_contract_address('Math', '0x...')
```

### Retrieving contract addresses

You can use the `BaseChain.registrar.get_contract_addresses()` method to retrieve all known addresses for a given contract. This method will return an iterable of addresses or throw a `populus.contracts.exceptions.NoKnownAddress` exception.

```python
>>> chain.registrar.get_contract_address('Math')
['0x123abc....']
```

### Retrieving contracts

Populus provides the following APIs for retrieving instances of your deployed contracts.

- `BaseChain.provider.get_contract()`
- `BaseChain.provider.deploy_contract()`
- `BaseChain.provider.get_or_deploy_contract()`

The `BaseChain.provider.get_contract()` function returns an instance of the requested contract.

```python
>>> math = chain.provider.get_contract('Math')
>>> math.address
'0x123abc....'
```

The `BaseChain.provider.deploy_contract()` function will deploy a new instance of the requested contract and return a two-tuple of the contract instance and the transaction hash that it was deployed with.

```python
>>> math, deploy_txn_hash = chain.provider.deploy_contract('Math')
>>> math.address
'0x123abc....' # 20 byte hex encoded address
>>> deploy_txn_hash
'0xabcdef...' # 32 byte hex encoded transaction hash
```
The `BaseChain.provider.get_or_deploy_contract()` function is primarily for testing purposes. If the contract is already available this method will return a two tuple of the contract instance and `None`. If the contract is not available it will be deployed using the provided deploy transaction and arguments, returning a two tuple of the contract instance and the deploy transaction hash.

```python
>>> math, deploy_txn_hash = chain.provider.get_or_deploy_contract('Math')
>>> math.address
'0x123abc....' # 20 byte hex encoded address
>>> deploy_txn_hash
'0xabcd6f...' # 32 byte hex encoded transaction hash
>>> chain.provider.get_or_deploy_contract('Math')
(<Math at 0x123abc>, None)
```

### Checking availability of contracts

Sometimes it may be useful to query whether a certain contract or its dependencies are available. This can be done with the following APIs.

- `BaseChain.provider.are_contract_dependencies_available()`
- `BaseChain.provider.is_contract_available()`

The `BaseChain.provider.are_contract_dependencies_available()` method returns `True` if all of the necessary dependencies for the provided contract are available. This check includes checks that the bytecode for all dependencies matched the expected compiled bytecode.

The `BaseChain.provider.is_contract_available()` method returns `True` if all dependencies for the requested contract are available and there is a known address for the contract and the bytecode at the address matches the expected bytecode for the contract.

### Wait API

```python
class populus.wait.Wait(web3, timeout=empty, poll_interval=empty)
```

Each chain object exposes the following API through a property `Chain.wait`.

- The `timeout` parameter sets the default number of seconds that each method will block before raising a `Timeout` exception.
- The `poll_interval` determines how long it should wait between polling. If `poll_interval == None` then a random value between 0 and 1 second will be used for the polling interval.

- **Wait for contract address**

  ```python
  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**

  ```python
  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**

  ```python
  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**

  ```python
  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**

  ```python
  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.\texttt{for\_syncing}(\texttt{timeout}=120, \texttt{poll\_interval}=None)
   
   Blocks for up to \texttt{timeout} seconds waiting the chain to begin syncing.

### Chain API

```python
class populus.chain.base.BaseChain
    All chain objects inherit from the \texttt{populus.chain.base.BaseChain} base class and expose the following API.

BaseChain.\texttt{web3}
    Accessor for the \texttt{Web3} instance that this chain is configured to use.

BaseChain.\texttt{wait}
    Accessor for the \texttt{Wait API}.

BaseChain.\texttt{registrar}
    Accessor for the Registrar API.

BaseChain.\texttt{provider}
    Accessor for the Provider API.
```

### Packaging

#### Contents

- Introduction
- Project Manifest
- Installing Packages
- Using Contracts from Installed Packages
- Library Linking
- Building and Publishing Releases

**Warning:** The packaging functionality is highly experimental. All APIs are subject to change without notice.

### Introduction

Populus can be used as a package manager to interact with any ERC190 smart contract packages.
Project Manifest

In order to take advantage of the packaging features you will first need to create a package manifest for your project. This can either be done manually or using the command line helper `$ populus package init` which will present an interactive prompt for creating the `ethpm.json` file.

```bash
$ populus package init
Writing new ethpm.json file.
Package Name: fancy-greeter
Author(s) []: Piper Merriam <pipermerriam@gmail.com>
Version [1.0.0]:
License [MIT]:
Description []: A fancy greeter contract
Keywords []: greeter, greetings
Links {}:
Wrote package manifest: ethpm.json
```

Installing Packages

Packages can be installed using the `$populus package install` command. Packages may be specified in the following formats.

- `populus package install .`
  
  To install all of the declared dependencies found within the project's package manifest.

- `populus package install some-package-name`
  
  To install a named package `some-package-name` sourced from a package index.

- `populus package install ipfs://QmUwVUMVtkVctrLDeL12SoeCPUacELBU8nAxRtHUzvtjND`
  
  To install directly from a release lockfile via IPFS

- `populus package install /path/to/release-lockfile.json`
  
  To install directly from a release lockfile on the local filesystem.

Populus also supports installing packages under aliased names. This can be used to allow multiple versions of the same package to be installed in tandem.

- `populus package install some-alias:some-package-name`
  
  To install a named package `some-package-name` under the name `some-alias` sourced from a package index.

- `populus package install some-alias@ipfs://QmUwVUMVtkVctrLDeL12SoeCPUacELBU8nAxRtHUzvtjND`
  
  To install directly from a release lockfile via IPFS using the name `some-alias`.

- `populus package install some-alias@/path/to/release-lockfile.json`
  
  To install directly from a release lockfile on the local filesystem using the name `some-alias`.

Packages are installed in the `./installed_packages` directory in the root project directory under their aliased name, or their package name if no alias is used.

When a package is installed it is automatically saved to the project dependencies within the package manifest. This can be disabled by passing in the `--no-save` flag during installation.
Using Contracts from Installed Packages

Importing a contract from an installed package is done by prefixing the source path with the name of the installed package, or the alias name if an alias was used.

Let's use the common `owned` pattern for an example. Suppose we have the `owned` package installed in our project. We know that this package has a single solidity source file that contains the `owned` contract located at `./contracts/owned.sol`.

To import a contract from this file into local solidity source files you would simply prefix the import path with the package name.

```solidity
pragma solidity ^0.4.0;

import "owned/contracts/owned.sol";

contract MyContract is owned {
    ...
}
```

**Note:** If you install a package which either has source files which do not compile with the solidity compiler version you are using, or which have a `pragma solidity` statement which is incompatible with your version of solidity then compilation will fail.

Library Linking

If you have a package installed which contains a library contract with a deployed instance of that library, populus will automatically find and link against that existing deployed library. One of the default contract backends that populus uses will check all installed packages.

Building and Publishing Releases

Populus can be used to build and publish packages to The Ethereum Package Registry or any registry which implements a compatible API.

To build a release use the `$ populus package build` command.

Release Notes

1.8.1

- Add `--logging` option to main cli command to set logging level.

1.8.0

- Change default compiler backend to `populus.compilation.backends.SolcAutoBackend` which will automatically select the appropriate solc compiler backend based on the current installed version of solc.
- Add support for standard JSON based solc compilation.
• Bugfix: Compilation now correctly respects import remappings.

1.7.0

• Remove deprecated `chain.contract_factories` API.
• Remove deprecated `chain.get_contract_factory` API.
• Remove deprecated `chain.is_contract_available` API.
• Remove deprecated `chain.get_contract` API.
• Remove deprecated `chain.deployed_contracts` API.
• Remove deprecated `contracts pytest fixture`.
• Remove deprecated `project.compiled_contracts_file_path` API
• Remove deprecated `project.contracts_dir` API
• Remove deprecated `project.build_dir` API
• Remove deprecated `project.compiled_contracts` API
• Remove deprecated `project.blockchains_dir` API
• Remove deprecated `project.get_blockchain_data_dir` API
• Remove deprecated `project.get_blockchain_chaindata_dir` API
• Remove deprecated `project.get_blockchain_dapp_dir` API
• Remove deprecated `project.get_blockchain_ipc_path` API
• Remove deprecated `project.get_blockchain_nodekey_path` API

1.6.9

• Bump py-geth version to account for removed `--ipcapi CLI flag`.

1.6.8

• Allow for empty passwords when unlocking accounts.

1.6.7

• Bugfix for registrar address sorting to handle nodes which were fast synced and do not have access to the full chain history.

1.6.6

• Add support to contract provider to handle case where registrar has more than one address for a given contract.

1.6.5

• Bugfix for compilation of abstract contracts.
1.6.4

- Bugfix for `project.config` setter function not setting correct value.

1.6.3

- Add `TestContractsBackend` for loading test contracts.

1.6.2

- Fix incorrect example test file from `$ populus init` command.

1.6.1

- Fix warning message for outdated config file so that it actually shows up in terminal.

1.6.0

- Introduce new Registrar API.
- Introduce new Provider API.
- Deprecate `Chain.get_contract_factory`, `Chain.get_contract` and `Chain.is_contract_available` APIs.
- Deprecate `Chain.contract_factories` API.
- Deprecate `Chain.deployed_contracts` API.
- Remove deprecated migrations API.

1.5.3

- Bump `web3.py` version to pull in upstream fixes for `ethereum-abi-utils`

1.5.2

- Bugfix for remaining `web3.utils` imports

1.5.1

- Update upstream `web3.py` dependency.
- Switch to use `ethereum-utils` library.
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.

1.4.2

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

1.4.1

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

1.4.0

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

1.3.0

- Bugfix for geth data_dir directory on linux systems.

1.2.2

- Support solc 0.4.x

1.2.1

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

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.
1.1.0

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`.

1.0.0

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-optimize`. - 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 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.

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.

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.
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.

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.

0.7.2

- Make the ethtester client work with asynchronous code.

0.7.1

- Adds `ipc_client` fixture.

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.

0.6.6

- Actually fix the address bug.

0.6.5

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

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.

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.

0.6.1

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

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.
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

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.

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.

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.

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.
0.4.3

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

0.4.2

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

0.4.1

- Missing `index.html` file.

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.

0.3.7

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

0.3.6

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

0.3.5

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

0.3.4

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

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.
0.3.2

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

0.3.1

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

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.

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.

0.1.4

- Fix broken import in cli module.

0.1.3

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

0.1.2

- Add missing pytest dependency.

0.1.1

- Fix bug when deploying contracts onto a real blockchain.

0.1.0

- Project Creation
Indices and tables

- genindex
- modindex
- search
Python Module Index

p

populus.chain.base, 42
populus.project, 29
populus.wait, 45
Index

B
BaseChain (class in populus.chain.base), 42, 46
BaseChain (class in populus.project), 29

C
config (populus.project.Project attribute), 30

F
for_block() (populus.wait.Wait method), 45
for_contract_address() (populus.wait.Wait method), 45
for_peers() (populus.wait.Wait method), 45
for_receipt() (populus.wait.Wait method), 45
for_syncing() (populus.wait.Wait method), 46
for_unlock() (populus.wait.Wait method), 45

L
load_config() (populus.project.Project method), 30

P
populus.chain.base (module), 42, 46
populus.project (module), 29
populus.wait (module), 45
populus.wait.Wait (class in populus.wait), 45
provider (populus.chain.base.BaseChain attribute), 46

R
registrar (populus.chain.base.BaseChain attribute), 46

W
wait (populus.chain.base.BaseChain attribute), 46
web3 (populus.chain.base.BaseChain attribute), 46
write_config() (populus.project.Project method), 30