Py-EthPM Documentation

Piper Merriam, et al.

Sep 19, 2018

Contents:

CHAPTER 1

Overview

This is a Python implementation of the Ethereum Smart Contract Packaging Specification, driven by discussions in ERC 190 and ERC 1123.

WARNING

`Py-EthPM` is currently in public alpha. *Keep in mind*:

- It is expected to have bugs and is not meant to be used in production
- Things may be ridiculously slow or not work at all

`Py-EthPM` is being built out to:

- Parse and validate packages.
- Provide access to contract factory classes (given a `web3` instance).
- Provide access to all of the deployed contract instances on a chain (given a connected `web3` instance).
- Validate package bytecode matches compilation output.
- Validate deployed bytecode matches compilation output.
- Access to package's dependencies.
- Construct and publish new packages.

CHAPTER 2

Manifest Builder

The Manifest Builder is a tool designed to help construct custom manifests. The builder is still under active development, and can only handle simple use-cases for now.

2.1 Cookbook

2.1.1 To create a simple manifest

For all manifests, the following ingredients are *required*.

```
build(
    {},
    package_name(str),
    version(str),
    manifest_version(str),
    ...,
)
# Or
build(
    init_manifest(package_name: str, version: str, manifest_version: str="2")
    ...,
)
```

The builder (i.e. `build()`) expects a dict as the first argument. This dict can be empty, or populated if you want to extend an existing manifest.

```
>>> from ethpm.tools.builder import *

>>> expected_manifest = {
...   "package_name": "owned",
...   "version": "1.0.0",
...   "manifest_version": "2"
```

(continues on next page)

(continued from previous page)

```
... }
>>> base_manifest = {"package_name": "owned"}
>>> built_manifest = build(
...     {},
...     package_name("owned"),
...     manifest_version("2"),
...     version("1.0.0"),
... )
>>> extended_manifest = build(
...     base_manifest,
...     manifest_version("2"),
...     version("1.0.0"),
... )
>>> assert built_manifest == expected_manifest
>>> assert extended_manifest == expected_manifest
```

With `init_manifest()`, which populates "version" with "2" (the only supported EthPM specification version), unless provided with an alternative "version".

```
>>> build(
...     init_manifest("owned", "1.0.0"),
... )
{'package_name': 'owned', 'version': '1.0.0', 'manifest_version': '2'}
```

2.1.2 To return a `Package`

```
build(
    ...,
    as_package(w3: Web3),
)
```

By default, the manifest builder returns a dict representing the manifest. To return a `Package` instance (instantiated with the generated manifest) from the builder, add the `as_package()` builder function with a valid `web3` instance to the end of the builder.

```
>>> from ethpm import Package
>>> from web3 import Web3

>>> w3 = Web3(Web3.EthereumTesterProvider())
>>> built_package = build(
...     {},
...     package_name("owned"),
...     manifest_version("2"),
...     version("1.0.0"),
...     as_package(w3),
... )
>>> assert isinstance(built_package, Package)
```

2.1.3 To validate a manifest

```
build(
    ...,
```

(continues on next page)

(continued from previous page)

```
    validate(),
)
```

By default, the manifest builder does *not* perform any validation that the generated fields are correctly formatted. There are two

- Return a Package, which automatically runs validation.
- Add the `validate()` function to the end of the manifest builder.

```
>>> valid_manifest = build(
...     {},
...     package_name("owned"),
...     manifest_version("2"),
...     version("1.0.0"),
...     validate(),
... )
>>> assert valid_manifest == {"package_name": "owned", "manifest_version": "2",
↪"version": "1.0.0"}
>>> invalid_manifest = build(
...     {},
...     package_name("_InvalidPkgName"),
...     manifest_version("2"),
...     version("1.0.0"),
...     validate(),
... )
Traceback (most recent call last):
ethpm.exceptions.ValidationError: Manifest invalid for schema version 2. Reason: '_
↪InvalidPkgName' does not match '^[a-z][-a-z0-9]{0,255}$'
```

2.1.4 To write a manifest to disk

```
build(
    ...,
    to_disk(
        manifest_root_dir: Optional[Path],
        manifest_name: Optional[str],
        prettify: Optional[bool],
    ),
)
```

Writes the active manifest to disk. Will not overwrite an existing manifest with the same name and root directory.

Defaults - Writes manifest to current working directory (as returned by *os.getcwd()*) unless a `Path` is provided as manifest_root_dir. - Writes manifest with a filename of "<version>.json" unless desired manifest name (which must end in ".json") is provided as manifest_name. - Writes the minified manifest version to disk unless prettify is set to True

```
>>> from pathlib import Path
>>> import tempfile
>>> p = Path(tempfile.mkdtemp("temp"))
>>> build(
...     {},
...     package_name("owned"),
...     manifest_version("2"),
```

(continues on next page)

(continued from previous page)

```
...     version("1.0.0"),
...     to_disk(manifest_root_dir=p, manifest_name="manifest.json", prettify=True),
... )
{'package_name': 'owned', 'manifest_version': '2', 'version': '1.0.0'}
>>> with open(str(p / "manifest.json")) as f:
...     actual_manifest = f.read()
>>> print(actual_manifest)
{
    "manifest_version": "2",
    "package_name": "owned",
    "version": "1.0.0"
}
```

2.1.5 To add meta fields

```
build(
    ...,
    description(str),
    license(str),
    authors(*args: str),
    keywords(*args: str),
    links(*kwargs: str),
    ...,
)
```

```
>>> BASE_MANIFEST = {"package_name": "owned", "manifest_version": "2", "version": "1.
↪0.0"}
>>> expected_manifest = {
...   "package_name": "owned",
...   "manifest_version": "2",
...   "version": "1.0.0",
...   "meta": {
...     "authors": ["Satoshi", "Nakamoto"],
...     "description": "An awesome package.",
...     "keywords": ["auth"],
...     "license": "MIT",
...     "links": {
...       "documentation": "www.readthedocs.com/...",
...       "repo": "www.github/...",
...       "website": "www.website.com",
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     authors("Satoshi", "Nakamoto"),
...     description("An awesome package."),
...     keywords("auth"),
...     license("MIT"),
...     links(documentation="www.readthedocs.com/...", repo="www.github/...", website=
↪"www.website.com"),
... )
>>> assert expected_manifest == built_manifest
```

2.1.6 Compiler Output

To build a more complex manifest, it is required that you provide standard-json output from the solidity compiler.

Here is an example of how to compile the contracts and generate the standard-json output. More information can be found in the Solidity Compiler docs.

```
solc --allow-paths <path-to-contract-directory> --standard-json < standard-json-input.
↪json > owned_compiler_output.json
```

Sample standard-json-input.json

```
{
    "language": "Solidity",
    "sources": {
        "Contract.sol": {
            "urls": [<path-to-contract>]
        }
    },
    "settings": {
        "outputSelection": {
            "*": {
                "*": ["abi", "evm.bytecode.object"]
            }
        }
    }
}
```

The `compiler_output` as used in the following examples is the entire value of the `contracts` key of the solc output, which contains compilation data for all compiled contracts.

2.1.7 To add a source

```
# To inline a source
build(
    ...,
    inline_source(
        contract_name: str,
        compiler_output: Dict[str, Any],
        package_root_dir: Optional[Path]
    ),
    ...,
)
# To pin a source
build(
    ...,
    pin_source(
        contract_name: str,
        compiler_output: Dict[str, Any],
        ipfs_backend: BaseIPFSBackend,
        package_root_dir: Optional[Path]
    ),
    ...,
)
```

There are two ways to include a contract source in your manifest.

Both strategies require that either . . .

- The current working directory is set to the package root directory or
- The package root directory is provided as an argument (`package_root_dir`)

To inline the source code directly in the manifest, use `inline_source()` or `source_inliner()` (to inline multiple sources from the same compiler_output), which requires the contract name and compiler output as args.

Note: *owned_compiler_output.json* below is expected to be the standard-json output generated by the solidity compiler as described *here <https://solidity.readthedocs.io/en/v0.4.24/using-the-compiler.html>*. The output must contain the *abi* and *bytecode* objects from compilation.

```
>>> import json
>>> from ethpm import ASSETS_DIR, V2_PACKAGES_DIR
>>> owned_dir = V2_PACKAGES_DIR / "owned" / "contracts"
>>> owned_contract_source = owned_dir / "Owned.sol"
>>> compiler_output = json.loads((ASSETS_DIR / "owned_compiler_output.json").read_
↪text())['contracts']
>>> expected_manifest = {
...   "package_name": "owned",
...   "version": "1.0.0",
...   "manifest_version": "2",
...   "sources": {
...     "./Owned.sol": """pragma solidity ^0.4.24;\n\ncontract Owned {\n    address"""
...     """ owner;\n    \n    modifier onlyOwner { require(msg.sender == owner); _; }
↪\n\n    """
...     """constructor() public {\n        owner = msg.sender;\n    }\n}\n"""
...   }
... }
>>> # With `inline_source()`
>>> built_manifest = build(
...     BASE_MANIFEST,
...     inline_source("Owned", compiler_output, package_root_dir=owned_dir),
... )
>>> assert expected_manifest == built_manifest
>>> # With `source_inliner()` for multiple sources from the same compiler output
>>> inliner = source_inliner(compiler_output, package_root_dir=owned_dir)
>>> built_manifest = build(
...     BASE_MANIFEST,
...     inliner("Owned"),
...     # inliner("other_source"), etc...
... )
>>> assert expected_manifest == built_manifest
```

To include the source as a content-addressed URI, `Py-EthPM` can pin your source via the Infura IPFS API. As well as the contract name and compiler output, this function requires that you provide the desired IPFS backend to pin the contract sources.

```
>>> from ethpm.backends.ipfs import get_ipfs_backend
>>> ipfs_backend = get_ipfs_backend()
>>> expected_manifest = {
...   "package_name": "owned",
...   "version": "1.0.0",
...   "manifest_version": "2",
...   "sources": {
...     "./Owned.sol": "ipfs://Qme4otpS88NV8yQi8TfTP89EsQC5bko3F5N1yhRoi6cwGV"
```

(continues on next page)

(continued from previous page)

```
...     }
... }
>>> # With `pin_source()`
>>> built_manifest = build(
...     BASE_MANIFEST,
...     pin_source("Owned", compiler_output, ipfs_backend, package_root_dir=owned_
↪dir),
... )
>>> assert expected_manifest == built_manifest
>>> # With `source_pinner()` for multiple sources from the same compiler output
>>> pinner = source_pinner(compiler_output, ipfs_backend, package_root_dir=owned_dir)
>>> built_manifest = build(
...     BASE_MANIFEST,
...     pinner("Owned"),
...     # pinner("other_source"), etc
... )
>>> assert expected_manifest == built_manifest
```

2.1.8 To add a contract type

```
build(
    ...,
    contract_type(
        contract_name: str,
        compiler_output: Dict[str, Any],
        alias: Optional[str],
        abi: Optional[bool],
        compiler: Optional[bool],
        contract_type: Optional[bool],
        deployment_bytecode: Optional[bool],
        natspec: Optional[bool],
        runtime_bytecode: Optional[bool]
    ),
    ...,
)
```

The default behavior of the manifest builder's `contract_type()` function is to populate the manifest with all of the contract type data found in the `compiler_output`.

```
>>> expected_manifest = {
...   'package_name': 'owned',
...   'manifest_version': '2',
...   'version': '1.0.0',
...   'contract_types': {
...     'Owned': {
...       'abi': [{'inputs': [], 'payable': False, 'stateMutability': 'nonpayable',
↪'type': 'constructor'}],
...       'deployment_bytecode': {
...         'bytecode':
↪'0x6080604052348015600f57600080fd5b50336000806101000a81548173ffffffffffffffffffffffffffffffffffffffff
↪'
...       },
...       'natspec': {}
...     }
```

(continues on next page)

(continued from previous page)

```
...     }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output)
... )
>>> assert expected_manifest == built_manifest
```

To select only certain contract type data to be included in your manifest, provide the desired fields as `True` keyword arguments

- `abi`
- `compiler`
- `deployment_bytecode`
- `natspec`
- `runtime_bytecode`

```
>>> expected_manifest = {
...     'package_name': 'owned',
...     'manifest_version': '2',
...     'version': '1.0.0',
...     'contract_types': {
...       'Owned': {
...         'abi': [{'inputs': [], 'payable': False, 'stateMutability': 'nonpayable',
↪'type': 'constructor'}],
...         'natspec': {}
...       }
...     }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output, abi=True, natspec=True)
... )
>>> assert expected_manifest == built_manifest
```

If you would like to alias your contract type, provide the desired alias as a kwarg. This will automatically include the original contract type in a `contract_type` field. Unless specific contract type fields are provided as kwargs, `contract_type` will stil default to including all availabe contract type data found in the compiler output.

```
>>> expected_manifest = {
...     'package_name': 'owned',
...     'manifest_version': '2',
...     'version': '1.0.0',
...     'contract_types': {
...       'OwnedAlias': {
...         'abi': [{'inputs': [], 'payable': False, 'stateMutability': 'nonpayable',
↪'type': 'constructor'}],
...         'natspec': {},
...         'contract_type': 'Owned'
...       }
...     }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
```

(continues on next page)

(continued from previous page)

```
...     contract_type("Owned", compiler_output, alias="OwnedAlias", abi=True,␣
↪natspec=True)
... )
>>> assert expected_manifest == built_manifest
```

CHAPTER 3

Packages

The `Package` object will function much like the `Contract` class provided by `web3`. Rather than instantiating the base class provided by `ethpm`, you will instead use a `classmethod` which generates a new `Package` class for a given package.

`Package` objects *must* be instantiated with a valid `web3` object.

- Creating a `Package` object from a local manifest file.

```
>>> from ethpm import Package, V2_PACKAGES_DIR
>>> from web3 import Web3

>>> owned_manifest_path = str(V2_PACKAGES_DIR / 'owned' / '1.0.0.json')
>>> w3 = Web3(Web3.EthereumTesterProvider())

>>> OwnedPackage = Package.from_file(owned_manifest_path, w3)
>>> assert isinstance(OwnedPackage, Package)
```

- Creating a `Package` object from a content-addressed URI pointing towards a valid manifest.

```
OwnedPackage = Package.from_uri('ipfs://QmbeVyFLSuEUxiXKwSsEjef6icpdTdA4kGG9BcrJXKNKUW
↪', w3)
```

- To change the `web3` instance of a `Package`.

```
>>> new_w3 = Web3(Web3.EthereumTesterProvider())
>>> OwnedPackage.set_default_w3(new_w3)
>>> assert OwnedPackage.w3 == new_w3
```

The following properties are available on a `Package` object.

```
>>> OwnedPackage.name
'owned'
>>> OwnedPackage.version
'1.0.0'
>>> OwnedPackage.manifest_version
```

(continues on next page)

(continued from previous page)

```
'2'
>>> OwnedPackage.__repr__()
'<Package owned==1.0.0>'
```

3.1 Contract Factories

Contract factories should be accessible from the package class.

```
Owned = OwnedPackage.get_contract_factory('owned')
```

In cases where a contract uses a library, the contract factory will have unlinked bytecode. The `ethpm` package ships with its own subclass of `web3.contract.Contract` with a few extra methods and properties related to bytecode linking.

```
>>> math = owned_package.contract_factories.math
>>> math.has_linkable_bytecode
True
>>> math.is_bytecode_linked
False
>>> linked_math = math.link_bytecode({'MathLib': '0x1234...'})
>>> linked_math.is_bytecode_linked
True
```

> Note: the actual format of the link data is not clear since library names aren't a one-size-fits all solution. We need the ability to specify specific link references in the code.

3.2 Contract Instances

To return a contract instance of a contract type belonging to a `Package`.

```
owned = OwnedPackage.get_contract_instance('owned', '0x123...')
```

3.3 Deployments

Deployed contracts are only available from package instances. The package instance will filter the `deployments` based on the chain that `web3` is connected to.

Accessing deployments is done with property access

```
package.deployed_contracts.Greeter
```

3.4 Dependencies

The `Package` class should provide access to the full dependency tree.

```
>>> owned_package.build_dependencies['zeppelin']
<ZeppelinPackage>
```

3.5 Validation

The `Package` class currently verifies the following things.

- Manifests used to instantiate a `Package` object conform to the EthPM V2 Manifest Specification

And in the future should verify.

- Included bytecode matches compilation output
- Deployed bytecode matches compilation output

CHAPTER 4

URI Schemes and Backends

4.1 Registry URI

The URI to lookup a package from a registry should follow the following format. (subject to change as the Registry Contract Standard makes it's way through the EIP process)

```
scheme://authority/package-name?version=x.x.x
```

- URI must be a string type
- `scheme`: `ercxxx`
- `authority`: Must be a valid ENS domain or a valid checksum address pointing towards a registry contract.
- `package-name`: Must conform to the package-name as specified in the EthPM-Spec.
- `version`: The URI escaped version string, *should* conform to the semver version numbering specification.

i.e. `ercxxx://packages.zeppelinos.eth/owned?version=1.0.0`

CHAPTER 5

Indices and tables

- genindex
- modindex
- search