Py-EthPM Documentation

Piper Merriam, et al.

Jul 09, 2019

Contents:

1 Overview **1**

2 Tools **3**
- 2.1 Builder . . . 3
- 2.2 Checker . . . 14

3 Package **15**
- 3.1 Properties . . . 15
- 3.2 Methods . . . 15
- 3.3 Validation . . . 16

4 LinkableContract **17**
- 4.1 Properties . . . 18
- 4.2 Methods . . . 18

5 URI Schemes and Backends **19**
- 5.1 BaseURIBackend . . . 19
- 5.2 IPFS . . . 19
- 5.3 HTTPS . . . 20
- 5.4 Registry URIs . . . 20

6 Release Notes **21**
- 6.1 v0.1.4-alpha.19 . . . 21
- 6.2 v0.1.4-alpha.18 . . . 21
- 6.3 v0.1.4-alpha.17 . . . 21
- 6.4 v0.1.4-alpha.16 . . . 21
- 6.5 v0.1.4-alpha.15 . . . 22
- 6.6 v0.1.4-alpha.14 . . . 22
- 6.7 v0.1.4-alpha.13 . . . 22
- 6.8 v0.1.4-alpha.12 . . . 22
- 6.9 v0.1.4-alpha.11 . . . 22
- 6.10 v0.1.4-alpha.10 . . . 22

7 Indices and tables **25**

Index **27**

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

Tools

2.1 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.

2.1.1 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"
... }
>>> base_manifest = {"package_name": "owned"}
```

(continues on next page)

(continued from previous page)

```
>>> 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(
    ...,
    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(
    ...,
    write_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"),
...     version("1.0.0"),
...     write_to_disk(manifest_root_dir=p, manifest_name="manifest.json",␣
↪prettify=True),
... )
{'package_name': 'owned', 'manifest_version': '2', 'version': '1.0.0'}
```

(continues on next page)

(continued from previous page)

```
>>> 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 pin a manifest to IPFS

```
build(
    ...,
    pin_to_ipfs(
        backend: BaseIPFSBackend,
        prettify: Optional[bool],
    ),
)
```

Pins the active manfiest to disk. Must be the concluding function in a builder set since it returns the IPFS pin data rather than returning the manifest for further processing.

2.1.6 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",
...     }
...   }
... }
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

2.1.7 Compiler Output

To build a more complex manifest for solidity contracts, 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.

2.1.8 To add a source

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

(continues on next page)

(continued from previous page)

```
    ...,
)
# 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" / "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}"""
...   }
... }
>>> # 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,
```

(continues on next page)

(continued from previous page)

```
...     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"
...   }
... }
>>> # 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
```

2.1.9 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
↪'
...       }
...     }
...   }
... }
>>> 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'}],
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output, abi=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',
```

(continues on next page)

(continued from previous page)

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

2.1.10 To add a deployment

```
build(
    ...,
    deployment(
        block_uri,
        contract_instance,
        contract_type,
        address,
        transaction=None,
        block=None,
        deployment_bytecode=None,
        runtime_bytecode=None,
        compiler=None,
    ),
    ...,
)
```

There are two strategies for adding a deployment to your manifest.

deployment(*block_uri*, *contract_instance*, *contract_type*, *address*, *transaction=None*, *block=None*, *deployment_bytecode=None*, *runtime_bytecode=None*, *compiler=None*)

This is the simplest builder function for adding a deployment to a manifest. All arguments require keywords. This builder function requires a valid `block_uri`, it's up to the user to be sure that multiple chain URIs representing the same blockchain are not included in the "deployments" object keys.

`runtime_bytecode`, `deployment_bytecode` and `compiler` must all be validly formatted dicts according to the EthPM Spec. If your contract has link dependencies, be sure to include them in the bytecode objects.

```
>>> from eth_utils import to_canonical_address
>>> expected_manifest = {
...     'package_name': 'owned',
...     'manifest_version': '2',
...     'version': '1.0.0',
...     'deployments': {
...       'blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef': {
...         'Owned': {
```

(continues on next page)

(continued from previous page)

```
...         'contract_type': 'Owned',
...         'address': '0x4f5b11c860b37b68de6d14fb7e7b5f18a9a1bd00',
...       }
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     deployment(
...         block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
...         contract_instance='Owned',
...         contract_type='Owned',
...         address=to_canonical_address('0x4f5b11c860b37b68de6d14fb7e7b5f18a9a1bd00
↪'),
...     ),
... )
>>> assert expected_manifest == built_manifest
```

deployment_type(*contract_instance*, *contract_type*, *deployment_bytecode=None*, *runtime_bytecode=None*, *compiler=None*)

This builder function simplifies adding the same contract type deployment across multiple chains. It requires both a `contract_instance` and `contract_type` argument (in many cases these are the same, though `contract_type` *must* always match its correspondent in the manifest's "contract_types") and all arguments require keywords.

`runtime_bytecode`, `deployment_bytecode` and `compiler` must all be validly formatted dicts according to the EthPM Spec. If your contract has link dependencies, be sure to include them in the bytecode objects.

```
owned_type = deployment_type(contract_instance="Owned", contract_type="Owned")
escrow_type = deployment_type(
    contract_instance = "Escrow",
    contract_type = "Escrow",
    deployment_bytecode = {
        "bytecode":
↪"0x6080604052348015610010576000080fd5b506040516020806104538339810160408181529151600281905533600081
↪"
    },
    runtime_bytecode = {
        "bytecode":
↪"0x6080604052600436106100775763ffffffff7c010000000000000000000000000000000000000000000000000000000
↪"
    },
    compiler = {
        "name": "solc",
        "version": "0.4.24+commit.e67f0147.Emscripten.clang",
        "settings": {
            "optimize": True
        }
    }
)
manifest = build(
    package_name("escrow"),
    version("1.0.0"),
    manifest_version("2"),
```

(continues on next page)

(continued from previous page)

```
    owned_type(
        block_uri='blockchain://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=owned_testnet_address,
    ),
    owned_type(
        block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=owned_mainnet_address,
        transaction=owned_mainnet_transaction,
        block=owned_mainnet_block,
    ),
    escrow_type(
        block_uri='blockchain://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=escrow_testnet_address,
    ),
    escrow_type(
        block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=escrow_mainnet_address,
        transaction=escrow_mainnet_transaction,
    ),
)
```

2.1.11 To add a build dependency

```
build(
    ...,
    build_dependency(
        package_name,
        uri,
    ),
    ...,
)
```

build_dependency (*package_name*, *uri*)

To add a build dependency to your manifest, just provide the package's name and a supported, content-addressed URI.

```
>>> expected_manifest = {
...   'package_name': 'owned',
...   'manifest_version': '2',
...   'version': '1.0.0',
...   'build_dependencies': {
...     'owned': 'ipfs://QmbeVyFLSuEUxiXKwSsEjef6icpdTdA4kGG9BcrJXKNKUW',
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     build_dependency('owned', 'ipfs://
↪QmbeVyFLSuEUxiXKwSsEjef6icpdTdA4kGG9BcrJXKNKUW'),
```

(continues on next page)

(continued from previous page)

```
... )
>>> assert expected_manifest == built_manifest
```

2.2 Checker

The manifest Checker is a tool designed to help validate manifests according to the natural language spec (link).

2.2.1 To validate a manifest

```
>>> from ethpm.tools.checker import check_manifest

>>> basic_manifest = {"package_name": "example", "version": "1.0.0", "manifest_version
↪": "2"}
>>> check_manifest(basic_manifest)
{'meta': "Manifest missing a suggested 'meta' field.", 'sources': 'Manifest is␣
↪missing a sources field, which defines a source tree that should comprise the full␣
↪source tree necessary to recompile the contracts contained in this release.',
↪'contract_types': "Manifest does not contain any 'contract_types'. Packages should␣
↪only include contract types that can be found in the source files for this package.␣
↪Packages should not include contract types from dependencies. Packages should not␣
↪include abstract contracts in the contract types section of a release."}
```

CHAPTER 3

Package

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.

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

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

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

3.1 Properties

Each `Package` exposes the following properties.

`Package.`**`w3`**
: The `Web3` instance currently set on this `Package`. The deployments available on a package are automatically filtered to only contain those belonging to the currently set `w3` instance.

`Package.`**`manifest`**
: The manifest dict used to instantiate a `Package`.

3.2 Methods

Each `Package` exposes the following methods.

3.3 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

LinkableContract

Py-EthPM uses a custom subclass of `Web3.contract.Contract` to manage contract factories and instances which might require bytecode linking. To create a deployable contract factory, both the contract type's *abi* and *deployment_bytecode* must be available in the Package's manifest.

```
>>> from eth_utils import is_address, to_canonical_address
>>> from web3 import Web3
>>> from ethpm import Package, ASSETS_DIR

>>> w3 = Web3(Web3.EthereumTesterProvider())
>>> escrow_manifest_path = ASSETS_DIR / 'escrow' / '1.0.3.json'

>>> # Try to deploy from unlinked factory
>>> EscrowPackage = Package.from_file(escrow_manifest_path, w3)
>>> EscrowFactory = EscrowPackage.get_contract_factory("Escrow")
>>> assert EscrowFactory.needs_bytecode_linking
>>> escrow_instance = EscrowFactory.constructor(w3.eth.accounts[0]).transact()
Traceback (most recent call last):
     ...
ethpm.exceptions.BytecodeLinkingError: Contract cannot be deployed until its bytecode␣
↪is linked.

>>> # Deploy SafeSendLib
>>> SafeSendFactory = EscrowPackage.get_contract_factory("SafeSendLib")
>>> safe_send_tx_hash = SafeSendFactory.constructor().transact()
>>> safe_send_tx_receipt = w3.eth.waitForTransactionReceipt(safe_send_tx_hash)

>>> # Link Escrow factory to deployed SafeSendLib instance
>>> LinkedEscrowFactory = EscrowFactory.link_bytecode({"SafeSendLib": to_canonical_
↪address(safe_send_tx_receipt.contractAddress)})
>>> assert LinkedEscrowFactory.needs_bytecode_linking is False
>>> escrow_tx_hash = LinkedEscrowFactory.constructor(w3.eth.accounts[0]).transact()
>>> escrow_tx_receipt = w3.eth.waitForTransactionReceipt(escrow_tx_hash)
>>> assert is_address(escrow_tx_receipt.contractAddress)
```

4.1 Properties

`LinkableContract.`**`unlinked_references`**
: A list of link reference data for the deployment bytecode, if present in the manifest data used to generate a `LinkableContract` factory. Deployment bytecode link reference data must be present in a manifest in order to generate a factory for a contract which requires bytecode linking.

`LinkableContract.`**`linked_references`**
: A list of link reference data for the runtime bytecode, if present in the manifest data used to generate a `LinkableContract` factory. Runtime bytecode link reference data must be present in a manifest in order to use `pytest-ethereum`'s `Deployer` for a contract which requires bytecode linking.

`LinkableContract.`**`needs_bytecode_linking`**
: A boolean attribute used to indicate whether a contract factory has unresolved link references, which must be resolved before a new contract instance can be deployed or instantiated at a given address.

4.2 Methods

`classmethod` `LinkableContract.`**`link_bytecode`**(*attr_dict*)
: This method returns a newly created contract factory with the applied link references defined in the *attr_dict*. This method expects *attr_dict* to be of the type `Dict[`contract_name`: `address`]` for all link references that are unlinked.

CHAPTER 5

URI Schemes and Backends

5.1 BaseURIBackend

`Py-EthPM` uses the `BaseURIBackend` as the parent class for all of its URI backends. To write your own backend, it must implement the following methods.

`BaseURIBackend.`**`can_resolve_uri`**(*uri*)
Return a bool indicating whether or not this backend is capable of resolving the given URI to a manifest. A content-addressed URI pointing to valid manifest is said to be capable of "resolving".

`BaseURIBackend.`**`can_translate_uri`**(*uri*)
Return a bool indicating whether this backend class can translate the given URI to a corresponding content-addressed URI. A registry URI is said to be capable of "transalating" if it points to another content-addressed URI in its respective on-chain registry.

`BaseURIBackend.`**`fetch_uri_contents`**(*uri*)
Fetch the contents stored at the provided uri, if an available backend is capable of resolving the URI. Validates that contents stored at uri match the content hash suffixing the uri.

5.2 IPFS

`Py-EthPM` has multiple backends available to fetch/pin files to IPFS. The desired backend can be set via the environment variable: `ETHPM_IPFS_BACKEND_CLASS`.

- **`InfuraIPFSBackend` (default)**
  - *https://ipfs.infura.io*
- **`IPFSGatewayBackend` (temporarily deprecated)**
  - *https://ipfs.io/ipfs/*
- **`LocalIPFSBacked`**
  - Connect to a local IPFS API gateway running on port 5001.

- **DummyIPFSBackend**
  - Won't pin/fetch files to an actual IPFS node, but mocks out this behavior.

`BaseIPFSBackend.`**`pin_assets`**(*file_or_directory_path*)
Pin asset(s) found at the given path and returns the pinned asset data.

5.3 HTTPS

`Py-EthPM` offers a backend to fetch files from Github, `GithubOverHTTPSBackend`.

A valid content-addressed Github URI *must* conform to the following scheme, as described in ERC1319, to be used with this backend.

```
https://api.github.com/repos/:owner/:repo/git/blobs/:file_sha
```

`create_content_addressed_github_uri`(*uri*)
This util function will return a content-addressed URI, as defined by Github's blob scheme. To generate a content-addressed URI for any manifest stored on github, this function requires accepts a Github API uri that follows the following scheme.

```
https://api.github.com/repos/:owner/:repo/contents/:path/:to/manifest.json
```

```
>>> from ethpm.uri import create_content_addressed_github_uri

>>> owned_github_api_uri = "https://api.github.com/repos/ethpm/py-ethpm/contents/
↪ethpm/assets/owned/1.0.1.json"
>>> content_addressed_uri = "https://api.github.com/repos/ethpm/py-ethpm/git/blobs/
↪a7232a93f1e9e75d606f6c1da18aa16037e03480"

>>> actual_blob_uri = create_content_addressed_github_uri(owned_github_api_uri)
>>> assert actual_blob_uri == content_addressed_uri
```

5.4 Registry URIs

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 6

Release Notes

6.1 v0.1.4-alpha.19

Released May 24nd, 2019

- Relax web3 dependency to 5.0.0b1

6.2 v0.1.4-alpha.18

Released May 23nd, 2019

- Bugfix in LinkableContract class - #159

6.3 v0.1.4-alpha.17

Released May 22nd, 2019

- Update ipfshttplib dependency & add tests - #157

6.4 v0.1.4-alpha.16

Released May 17th, 2019

- Update to use new IPFS library - #158
- Update mypy dependency - #153

6.5 v0.1.4-alpha.15

Released April 25th, 2019

- **Write is_supported_content_addressed_uri util.**
  - #152

6.6 v0.1.4-alpha.14

Released April 10th, 2019

- Bugfix
  - Update registry backend to work with `web3.pm` - #151

6.7 v0.1.4-alpha.13

Released March 22nd, 2019

- Bugfix
  - Remove auto infura endpoint - #149

6.8 v0.1.4-alpha.12

Released February 12th, 2019

- Breaking Changes
  - Change `Package.switch_w3` to `Package.update_w3` - #146

6.9 v0.1.4-alpha.11

Released February 12th, 2019

- Breaking Changes
  - Remove `py-solc` dependency and solidity compilation - #143
  - Update vyper reference registry assets - #145
- Features
  - Support contract aliasing for deployments - #144

6.10 v0.1.4-alpha.10

Released January 17th, 2019

- Breaking Changes

  - `Package.set_default_w3()` returns new `Package` instance. - #139
  - `Web3` dependency updated to `v5.0.0a3`. - #137
- Bugfixes
  - `Builder` bugfix to account for contract factories. - #138
  - Add `global_doctest_setup` for autodoc to use.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

B

build_dependency() (*built-in function*), 13

C

can_resolve_uri() (*BaseURIBackend method*), 19
can_translate_uri() (*BaseURIBackend method*), 19
create_content_addressed_github_uri(), 20

D

deployment() (*built-in function*), 11
deployment_type() (*built-in function*), 12

F

fetch_uri_contents() (*BaseURIBackend method*), 19

L

link_bytecode() (*LinkableContract class method*), 18
linked_references (*LinkableContract attribute*), 18

M

manifest (*Package attribute*), 15

N

needs_bytecode_linking (*LinkableContract attribute*), 18

P

pin_assets() (*BaseIPFSBackend method*), 20

U

unlinked_references (*LinkableContract attribute*), 18

W

w3 (*Package attribute*), 15