owmeta-core Documentation

Release dev

Mark Watts

Jan 18, 2022

Contents

Our main README is available online on Github.[1] This documentation contains additional materials beyond what is covered there.

Contents:

[1] http://github.com/openworm/owmeta-core

CHAPTER 1

owmeta_core

1.1 owmeta_core package

1.1.1 Subpackages

owmeta_core.bundle package

Subpackages

owmeta_core.bundle.loaders package

Submodules

owmeta_core.bundle.loaders.http module

owmeta_core.bundle.loaders.sftp module

Submodules

owmeta_core.bundle.archive module

owmeta_core.bundle.common module

owmeta_core.bundle.exceptions module

owmeta_core.commands package

Submodules

1.1.2 Submodules

CHAPTER 2

For Users

2.1 Making data objects

To make a new object type like you just need to make a subclass of `DataObject` with the appropriate.

Say, for example, that I want to record some information about drug reactions in dogs. I make `Drug`, `Experiment`, and `Dog` classes to describe drug reactions:

```
>>> from owmeta_core.dataobject import (DataObject,
...                                     DatatypeProperty,
...                                     ObjectProperty,
...                                     Alias)
>>> from owmeta_core.context import Context
>>> from owmeta_core.mapper import Mapper

>>> module_context = 'http://example.com/animals'

>>> class Dog(DataObject):
...     breed = DatatypeProperty()

>>> class Drug(DataObject):
...     name = DatatypeProperty()
...     drug_name = Alias(name)
...     key_property = dict(property=name, type='direct')

>>> class Experiment(DataObject):
...     drug = ObjectProperty(value_type=Drug)
...     subject = ObjectProperty(value_type=Dog)
...     route_of_entry = DatatypeProperty()
...     reaction = DatatypeProperty()

# Do some accounting stuff to register the classes. Usually happens behind
# the scenes.
>>> m = Mapper()
>>> m.process_classes(Drug, Experiment, Dog)
```

So, we have created I can then make a Drug object for moon rocks and describe an experiment by Aperture Labs:

```
>>> ctx = Context('http://example.org/experiments', mapper=m)
>>> d = ctx(Drug)(name='moon rocks')
>>> e = ctx(Experiment)(key='experiment001')
>>> w = ctx(Dog)(breed='Affenpinscher')
>>> e.subject(w)
owmeta_core.statement.Statement(...Context(.../experiments"))

>>> e.drug(d)
owmeta_core.statement.Statement(...)

>>> e.route_of_entry('ingestion')
owmeta_core.statement.Statement(...)

>>> e.reaction('no reaction')
owmeta_core.statement.Statement(...)
```

and save those statements:

```
>>> ctx.save()
```

For simple objects, this is all we have to do.

You can also add properties to an object after it has been created by calling either ObjectProperty or DatatypeProperty on the class:

```
>>> d = ctx(Drug)(name='moon rocks')
>>> Drug.DatatypeProperty('granularity', owner=d)
__main__.Drug_granularity(owner=Drug(ident=rdflib.term.URIRef('http://data.openworm.
↪org/Drug#moon%20rocks')))

>>> d.granularity('ground up')
owmeta_core.statement.Statement(...Context(.../experiments"))

>>> do = Drug()
```

Properties added in this fashion will not propagate to any other objects:

```
>>> do.granularity
Traceback (most recent call last):
    ...
AttributeError: 'Drug' object has no attribute 'granularity'
```

They will, however, be saved along with the object they are attached to.

2.2 Working with contexts

2.2.1 Background

Contexts were introduced to owmeta-core as a generic tool for grouping statements. We need to group statements to make statements about statements like “Who made these statements?” or “When were these statements made?”. That’s the main usage. Beyond that, we need a way to share statements. Contexts have identifiers by which we can naturally refer to contexts from other contexts.

owmeta-core needs a way to represent contexts with the existing statement form. Other alternatives were considered, such as using Python's context managers, but I (Mark) also wanted a way to put statements in a context that could also be carried with the subject of the statement. Using the wrapt package's proxies allows to achieve this while keeping the interface of the wrapped object the same, which is useful since it doesn't require a user of the object to know anything about contexts unless they need to change the context of a statement.

The remainder of this page will go into doing some useful things with contexts.

2.2.2 Classes and contexts

owmeta-core can load classes as well as instances from an RDF graph. The packages which define the classes must already be installed in the Python library path, and a few statements need to be in the graph you are loading from or in a graph imported (transitively) by that graph. The statements you need are these

```
:a_class_desc <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://openworm.org/
↪entities/PythonClassDescription> .
:a_class_desc <http://openworm.org/entities/ClassDescription/module> :a_module .
:a_class_desc <http://openworm.org/entities/PythonClassDescription/name> "AClassName"␣
↪.

:a_module <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://openworm.org/
↪entities/PythonModule> .
:a_module <http://openworm.org/entities/PythonModule/name> "APackage.and.module.name"␣
↪.
```

where `:a_class_desc` and `:a_module` are placeholders for objects which will typically be created by owmeta-core on the user's behalf, and `AClassName` is the name of the class available at the top-level of the module `APackage.and.module.name`. These statements will be created in memory by owmeta-core when a module defining a `DataObject`-derived class is first processed by a `Mapper` which will happen after the module is imported.

2.3 `owm` Command Line

The `owm` command line provides a high-level interface for working with owmeta-core-managed data. The central object which `owm` works on is the owmeta-core project, which contains the triple store – a set of files in a binary format. The sub-commands act on important files inside the project or with entities in the database.

To get usage information:

```
owm --help
```

To clone a project:

```
owm clone $database_url
```

This will clone a project into `.owm` in your current working directory. After a successful clone, a binary database usable as a owmeta store will have been created from the serialized graphs (i.e., sets of RDF (Resource Description Framework) triples) in the project.

To save changes made to the database, run the `commit` sub-command like this:

```
owm commit -m "Adding records from January-March"
```

To recreate the database from serialized graphs, run the `regendb` sub-command:

```
owm regendb
```

Be careful with `regendb` as it will delete anything you have added to binary database beyond what's in the serialized graphs.

To make a new project:

```
owm init
```

This will create a project in `.owm` in your current working directory.

2.4 Software Versioning

The owmeta-core library follows the semanitc versioning scheme. For the sake of versioning, the software interface consists of:

1. The `owm` command line interface
2. The underlying `owmeta_core.command.OWM` class underlying that CLI
3. All "public" definitions (i.e., those whose names do not begin with '_') in the `owmeta_core` package, sub-packages, and sub-modules
4. The format of RDF data generated by `owmeta_core.dataobject.DataObject` and the subclasses thereof defined in the `owmeta_core` package, sub-packages, and sub-modules
5. The API documentation for the `owmeta_core` package, sub-packages, and sub-modules

In addition, any changes to the packages released on PyPI mandates at least a patch version increment.

For Git, our software version control system, software releases will be represented as tags in the form `v$semantic_version` with all components of the semantic version represented.

2.4.1 Documentation versioning

The documentation will have a distinct version number from the software. The version numbers for the documentation must change at least as often as the software versioning since the relationship of the documentation to the software necessarily changes. However, changes *only* to the non-API documentation will not be a cause for a change to any of the components of the software version number. For documentation releases which coincide with software releases, the documentation version number will simply be the software version number. Any subsequent change to documentation between software releases will compel an increase in the documentation version number by one. The documentation version number for such documentation releases will be represented as `${software_version}+docs${documentation_increment}`.

2.4.2 Mapped Class Versioning

Versioning for mapped classes has special considerations related to how Python classes map to RDF types. RDF types map to a specific class, module, and package version through the "class registry". This class registry is contained within a *bundle* and each bundle is free to have its own registry. The registry, moreover, can be replaced by another registry by the user of the bundle.

We want data created with later versions of a mapped class to be compatible with earlier versions of that class so that we can use pre-existing analysis and transformations (e.g., with a `DataTranslator`). This flexibility allows for pre-existing processes to keep working without change even as the upstream moves on. On the other hand, newer

versions of a software package should still have access to the data created with older versions of the corresponding Python classes so that users of that package are not forced to translate or abandon the old data.

The two-way compatibility described above is appropriate in the context of the "open world assumption": the relationships an RDF type participates in are described by the Python class, but that description may be incomplete. We may make the description of an RDF type more complete by adding properties to the Python class or constraining existing properties. When we add properties, however, we should create a new Python class rather than modifying the existing one: this allows for querying for data created with the earlier version of the Python class while also being able to create instances of the new class. The new class should not, however, have the same RDF type as the old one since the code for resolving types from the class registry only supports mapping to one Python type from any given RDF type[1]. The recommended way to handle this is to include a version number in the URI for the RDF type and, when making the new type, to increment the version number for the new URI. The new type should be declared as a sub-class of the old type, and owmeta-core will add the appropriate sub-class relationships so that querying for the old type will return instances of the new type as well. This split also means that while I use the new software package, I can utilize the data generated with the old Python class without needing to have the old Python package because the new package retains the old class.

2.4.3 Release Notes

Release notes are organized, generally into three sections. 'Features and Enhancements' are changes to the external interface of owmeta-core where there wasn't anything that fulfilled the use case previously (features) or where the previous behavior was sub-optimal or just different (enhancements), but not wrong per se. The second section, 'Fixes', contains corrections to previous behavior. The third section 'Internal/Misc. Changes' contains changes that either don't really change owmeta-core itself, like changes in to project metadata, documentation changes, or changes to build automation. Other sections may be added, like 'Known Issues', which should be self-explanatory when used.

Notes:

2.5 Python Release Compatibility

All Python releases will be supported until they reach their official end-of-life, typically reported as "Release Schedule" PEPs (search "release schedule" on the PEP index) Thereafter, any regressions due to dependencies of owmeta-core dropping support for an EOL Python version, or due to a change in owmeta-core making use of a feature in a still-supported Python release will only be fixed for the sake of OpenWorm projects when requested by an issue on our tracker or for other projects when a compelling case can be made.

This policy is intended to provide support to most well-maintained projects which depend on owmeta-core while not overburdening developers.

2.6 BitTorrent client for P2P filesharing

1. **Download** desired contents:

- A `LocalFileDataSource` created and stored within the local graph store contains a torrent_file_name `Informational`. This refers to the torrent containing the location of the desired contents on the BitTorrent. A `torrent` is used to locate files on the File System [BEP 3]. A `DataSource` defines attributes about the contents that it represents.

[1] One alternative to this is returning, for each RDF instance of a type, `N` Python instances for `N` Python classes in the registry mapped to the RDF type.

Module `t` describes the `DataSource` attributes:

```
def owm_data(ns):
    ns.context.add_import(ConnectomeCSVDataSource.definition_context)
    ns.context(ConnectomeCSVDataSource)(
    key = '2000_connections',
    csv_file_name = 'connectome.csv',
    torrent_file_name = 'd9da5ce947c6f1c127dfcdc2ede63320.torrent'
 )
```

The `DataSource` can be created and stored on the local graph with:

```
$ owm save t
```

The `DataSource` identifier can be used to see contents stored in the local graph with:

```
$ owm source show ConnectomeCSVDataSource:2000_connections
```

ConnectomeCSVDataSource CSV file name: 'connectome.csv'

File name: 'connectome.csv'

Torrent file name: 'd9da5ce947c6f1c127dfcdc2ede63320.torrent'

- The BitTorrentDataSourceDirLoader class inherits from the `DataSourceDirLoader` and overrides its `load()` method. Google Drive stores the `torrents` uploaded by other researchers. `load()` fetches the `torrent` refered to in `torrent_file_name` of the `DataSource`, performs `DataTranslator` from one form to another and then adds the `torrent` to the BitTorrent Client for downloading its contents.

This `BitTorrent Client` is available on PyPI and is included in the owmeta_core setup.

To install separately:

```
$ pip install torrent-client
```

For reference, use the torrent-client repository and its usage information with:

```
$ torrent_cli.py -h
```

The `DataSourceDirLoader` attribute - `base_directory`, which is set in the `BitTorrentDataSourceDirLoader` constructor is where both the `torrent` and its contents are downloaded:

```
content = BitTorrentDataSourceDirLoader("./")
```

- Within the .owm directory we have the credentials.json and token.pickle these are for authentication of the Google Drive. For the purpose of access control the `client_secret` required by `credentials.json` will only be shared by owmeta maintainers.

- The `torrent` file name is the MD5 message digest of its contents. If the hash of the downloaded contents is the same as its `torrent` name the data is unaltered.

Data-Integrity is to be checked after 100% download completion:

```
$ python3 integrity.py 'd9da5ce947c6f1c127dfcdc2ede63320.torrent' 'Merged_
↪Nuclei_Stained_Worm.zip'
```

2. **Upload** your contents:

- On an AWS EC2 instance is running a Nginx WSGI and a Flask Server to accept .zip content file uploads. Visit this Elastic IP address [13.235.204.78] to upload your files by browsing through your filesystem and then clicking the `Submit Query button`.
- This will create a `torrent` and `seed` your contents in parts, to other peers on the BitTorrent network. Content can then be downloaded as described above.

2.7 Querying for data objects

2.7.1 DataObject query form

Sub-classes of `DataObject` have a `query` attribute that provides a modified form of the class which is fit for creating instances used in queries. The query form may do other things later, but, principally, it overrides identifier generation based on attributes (see `IdMixin`).

For example, to query for a `Neuron` object with the name "AVAL" you would instantiate the `Neuron` like this:

```
>>> Neuron.query(name='AVAL')
```

Although it is possible to include instances without the query form, it is generally preferred to the basic form since later versions of a class may change how they generate identifiers while keeping property URIs and RDF types the same (or declaring new ones as sub-properties or sub-classes). Use of the query form is also recommended when a class generates identifiers based on some number of properties, but a subclass doesn't use the superclass identifier scheme (`Cell` and `Neuron` are an example). The query form allows to query for instances of the superclass for subclass instances.

CHAPTER 3

For Developers

3.1 Testing in owmeta-core

3.1.1 Preparing for tests

owmeta_core should be installed like:

```
pip install -e .
```

3.1.2 Running tests

Tests should be run via setup.py like:

```
python setup.py test
```

you can pass options to `pytest` like so:

```
python setup.py test --addopts '-k CommandTest'
```

3.1.3 Writing tests

Tests are written using Python's unittest. In general, a collection of closely related tests should be in one file. For selecting different classes of tests, tests can also be tagged using pytest marks like:

```
@pytest.mark.tag
class TestClass(unittest.TestCase):
    ...
```

Currently, marks are used to distinguish between unit-level tests and others which have the `inttest` mark

3.1.4 Deselecting tests

Tests can be deselected by adding a pytest "marker" to the test function, class, or module and then adding `-m 'not <your_marker>'` to the pytest command line. Marking tests to be explicitly deselected is preferred to skipping tests since skipped tests tend to break silently, especially with conditional skips such as with with `pytest.mark.skipif`. A set of markers is, however, deselected by default in the `addopts` line in our `pytest.ini` file. Deselected marks are added on a case-by-case basis and will always run on CI.

3.2 Writing documentation

Documentation for owmeta-core is housed in two locations:

1. In the top-level project directory as `INSTALL.md` and `README.md`.
2. As a Sphinx project under the `docs` directory

By way of example, to add a page about useful facts concerning *C. elegans* to the documentation, include an entry in the list under `toctree` in `docs/index.rst` like:

```
worm-facts
```

and create the file `worm-facts.rst` under the `docs` directory and add a line:

```
.. _worm-facts:
```

to the top of your file, remembering to leave an empty line before adding all of your wonderful worm facts.

You can get a preview of what your documentation will look like when it is published by running `sphinx-build` on the docs directory. To get the `sphinx-build` command, install the documentation requirements with:

```
pip install -r doc-requirements.txt
```

Then, you can run `sphinx-build` like this:

```
sphinx-build -w sphinx-errors docs <build_destination>
```

You can also invoke the command with default arguments (i.e., with output to `build/sphinx` using `setup.py`:

```
python setup.py build_sphinx
```

The docs will be compiled to html which you can view by pointing your web browser at `<build_destination>/index.html`. The documentation will be rendered using the same theme as is used on the readthedocs.org site.

3.2.1 API Documentation

API documentation is generated by the Sphinx autodoc and apidoc extensions. The numpydoc format should be easy to pick up on, but a reference is available here. Just add a docstring to your function/class/method and your class should appear among the other documented classes. Note, however, that "special" methods like `__call__` will not show up by default – if they need to be documented for a given class, add a declaration like this to the class documentation:

```
class SpecialMethodDocExample:
    '''
    Example class doc

    .. automethod:: __call__
```

(continues on next page)

(continued from previous page)

```
    '''

    def __call__(self):
        '''
        Hey, I'm in the API documentation!
        '''
```

3.2.2 Substitutions

Project-wide substitutions can be (conservatively!) added to allow for easily changing a value over all of the documentation. Currently defined substitutions can be found in `conf.py` in the `rst_epilog` setting. More about substitutions

3.2.3 Conventions

If you'd like to add a convention, list it here and start using it. It can be reviewed as part of a pull request.

1. Narrative text should be wrapped at 80 characters.
2. Long links should be extracted from narrative text. Use your judgement on what 'long' is, but if it causes the line width to stray beyond 80 characters that's a good indication.

3.3 owmeta-core coding standards

Pull requests are *required* to follow the PEP-8 Guidelines for contributions of Python code to owmeta-core, with some exceptions noted below. Compliance can be checked with the `pep8` tool and these command line arguments:

```
--max-line-length=120 --ignore=E261,E266,E265,E402,E121,E123,E126,E226,E24,E704,E128
```

Refer to the pep8 documentation for the meanings of these error codes.

Lines of code should only be wrapped before 120 chars for readability. Comments and string literals, including docstrings, can be wrapped to a shorter length.

Some violations can be corrected with `autopep8`.

3.4 Design documents

These comprise the core design artifacts for owmeta.

3.4.1 Project Bundles

A project bundle is composed of:

- a universally unique identifier,
- a version number,
- a collection of contexts,

- a distinguished "imports" context describing relationships between contexts, both those in the bundle, and between contexts in the bundle and in dependencies,

plus several optional components:

- a human-friendly name,
- a description of the bundle's contents,
- a collection of files,
- a listing of dependencies on other bundles,
- a set of mappings between project-scoped identifiers and universal context identifiers.

They solve the problem of contexts containing different statements having the same identifier for different purposes.

There are several ways we can get different contexts with the same identifier:

- through revisions of a context over time,
- by distinct groups using the same context identifier,
- or by contexts being distributed with different variants (e.g., a full and an abridged version).

In solving this problem of context ID aliasing, bundles also helps solve the problem of having contexts with inconsistent statements in the same project by providing a division within a project, between groups of contexts that aren't necessarily related.

Dependencies

A bundle can declare other bundles upon which it depends, by listing those other bundles identifiers and version numbers. In addition, a bundle can declare contexts and files within the dependency that should be included or excluded. More interestingly, a dependency specification may declare that contexts declared within the dependency be renamed according to a number of rewrite rules. This is to allow for using bundles with conflicting Context Identifiers.

Certain problems come up when dealing with contexts across different bundles. This rewriting allows to keep separate the contexts in one bundle from another and to prevent contexts with the same ID from conflicting with one another just because they're brought in by a transitive dependency.

An example

This example describes a likely naming conflict that can arise in context naming between bundles.

Bundles α, β, and γ. With dependencies like so:

```
α -> β -> γ
```

where both α and γ contain a context with ID $\subset$. The dependency resolution system will find the $\subset$ context in γ and if there is no remapping that removes the conflict, either in β or in α, then the system will deliver a message indicating that the context needs to be deconflicted and in which bundle each of the conflicting declarations is. At this point, the maintainer of the α package can make the change to omit $\subset$ from γ, omit it from α, rename $\subset$ in γ, or rename it in α. One special case, where α's $\subset$ and γ's $\subset$ are identical, permits an automatic resolution; nonetheless, the system emits a warning in this case, with the option to fail similarly to the case where the contexts are distinct.

Core bundles

The "core" bundle contains (or depends on) metadata of all of the core classes in owmeta which are needed to make owmeta features work. The core bundle is generated automatically for whichever version of owmeta is in use and a

reference to it is added automatically when a bundle is installed. A given bundle may, however, explicitly use a specific version of the core bundle.

Relationships

Where not specified, the subject of a relationship can participate in the relationship exactly once. For example, "A Dog has a Human", means "A Dog has one and only one Human".

- A Project can have zero or more Bundles
- A Bundle can belong to only one Project
- A Context Identifier is associated with one or more Content-Based Identifiers
- A Content-Based Identifier has a Hash
- A Content-Based Identifier has an RDF Serialization Format
- A Hash can appear in zero or more Content-Based Identifiers
- A Hash has an Algorithm ID and a Message Digest

Types

Below is a description in terms of lower-level types of some higher-level types referenced above.

- A Message Digest is a Base-64 encoding of a string of bytes
- An Algorithm ID is a string that identifies an algorithm. Valid strings will be determined by any applications reading or writing the hashes, but in general will come from the set of constructors of Python's `hashlib` module.
- An RDF Serialization Format is a string naming the format of a canonical RDF graph serialization. Supported format strings:

  "nt" N-Triples

3.4.2 Project Distribution

Projects are distributed as *bundle* archives, also referred to as `dists` (short for distributions) in the documentation and commands. The layout of files in a dist is largely the same as the format of a `.owm` directory on initial clone. In other words the bundle contains a set of serialized graphs, an index of those graphs, an optional set of non-RDF data that accompanies data sources stored amongst the graphs, and a configuration file which serves as a working owmeta configuration file and a place for metadata about the bundle. The archive file format can be allowed to vary, between producers and consumers of dists, but at least the `tar.gz` format should be supported by general-purpose clients.

3.4.3 Data Packaging Lifecycle

The package lifecycle encompasses the creation of data, packaging of said data, and uploading to shared resources. The data packaging lifecycle draws from the Maven build lifecycle in the separation of local actions (e.g., `compile`, `stage`, `install` phases) from remote interactions (the `deploy` phase). To explain why we have these distinct phases, we should step back and look at what needs to happen when we share data.

In owmeta-core, we may be changing remote resources outside of the owmeta-core system. We also want to support local use and staging of data because it is expected that there is a lengthy period of data collection/generation, analysis, curation, and editing which precedes the publication of any data set. Having separate phases allows us to support a wider range of use-cases with owmeta-core in this local "staging" period.

To make the above more concrete, the prototypical example for us is around `LocalFileDataSource`, which wants to make the files described in the data source available for download. Typically, the local path to the file isn't useful outside of the machine. Also, except for files only a few tens of bytes in size, it isn't feasible to store the file contents in the same database as the metadata. We, still want to support metadata about these files and to avoid the necessity of making *n* different `DataSource` sub-classes for *n* different ways of getting a file. What we do is define a "deploy" phase that takes every `LocalFileDataSource` and "deploys" the files by uploading them to one or more remote stores or, in the case of a peer-to-peer solution, by publishing information about the file to a tracker or distributed hash table.

Packaging proceeds in phases to serve as an organizational structure for data producers, software developers, management, and information technology personnel. Compared with a more free-form build strategy like using an amalgam of shell scripts and disconnected commands, or even rule-based execution (e.g., GNU make), phases organize the otherwise implicit process by which the local database gets made available to other people. This explicitness is very useful since, when different people can take different roles in creating the configuration for each phase, having named phases where things happen aids in discussion, process development, and review. For instance, junior lab technicians may be responsible for creating or maintaining packaging with guidance from senior technicians or principal investigators. IT personnel may be interested in all phases since they all deal with the computing resources they manage, but they may focus on the phases that affect "remote" resources since those resources may, in fact, be managed within the same organization and require additional effort on the back-end to prepare those remote resources (e.g., generating access credentials).

The remainder of this document will describe the default lifecycle and what takes place within each phase.

Default Lifecycle

The default lifecycle takes a *bundle*, including the contents of a owmeta-core triple store, creates one or more packages from that, stages the packages for ongoing development, and, finally, deploys packages to shared resources so that colleagues and other interested parties can access them. Each phase is associated with a sub-command in *owm*.

Install

Preparation for distribution.

When we're generating data, our workspace is not necessarily in the right state for distribution. We may have created temporary files and notes to ourselves, or we may have generated data in trial runs, intentionally or mistakenly, which do not reflect our formal experimental conditions. In the install phase, we bring together just the data which we wish to distribute for a given bundle and place it in the local bundle cache. This includes gathering hashes for files that belong to the bundle and serializing named graphs. Once these data are installed, they should be immutable – in other words, they should not change any more. Consequently, the install phase is the appropriate time for creating summary statistics, signatures, and content-based identifiers.

Much of the data which is created in a research lab is append-only: observations are logged and timestamped either by a human or by a machine in the moment they happen, and, if recorded properly, such logs are rarely edited, or, if there is an amendment, it also is logged as such, with the original record preserved. As long as this append-only property is preserved, we only need to designate the range of such time-stamped records which belong in a bundle to have the desired immutability for a locally installed bundle without requiring a file copy operation. Of course, if the source data is expected to be changed, then we would want either a copy-on-write mechanism (at the file system level) or to copy the files. Regardless, file hashes and/or signatures created during the install phase would be available for guarding against accidental changes.

owmeta-core will create a local repository to house installed packages. The repository stores the relationship between the human-friendly name for the package (serving a purpose similar to Maven's group-artifact-version coordinates) and the set of serialized RDF graphs in the package. Given that the repository is meant to serve a user across projects,

the repository will be stored in the "user directory", if one can be found on the system.[1]

Deploy

Creation of configuration for upload/download. Sharing packages.

In the "deploy" phase, we publish our data to "remotes". A "remote" may be a repository or, in the case of a peer-to-peer file sharing system, a file index or DHT. Above, we referred to non-RDF data files on the local file system – during the deploy phase, these files are actually published and accession information (e.g., a database record identifier) for those files is generated and returned to the system where the deployment was initiated. This assumes a fully automated process for publication of files: If, instead, the publication platform requires some manual interaction, that must be done outside of owmeta-core and then the accession information would be provided with the deploy command.

3.4.4 Publishing DataSources

`DataSource` is a subclass of `DataObject` with a few features to make describing data files (CSV, HDF5, Excel) a bit more consistent and to make recovering those files, and information about them, more reliable. In order to have that reliability we have to take some extra measures when publishing a `DataSource`. In particular, we must publish local files referred to by the `DataSource` and relativize those references. This file publication happens in the *"deploy" phase* of the data packaging lifecycle. Before that, however, a description of what files need to be published is generated in the *"stage" phase*. In the "stage" phase, the `DataSources` with files needing publication are queried for in the configured triple store, and the "staging manager", the component responsible for coordinating the "stage" phase identifies file references that refer to the same files and directories.

[1] This will be the user directory as determined by `os.path.expanduser()`

CHAPTER 4

Issues

CHAPTER 5

Indices and tables

- genindex
- modindex
- search