LASS Metadata Package

Release 1.0

URY Computing Team and contributors

April 06, 2013

CONTENTS

django-lass-metadata provides the `metadata` package, which introduces a generic, external database user friendly means of attaching "strands" of typed data to other models in an extensible and generic manner.

CHAPTER

ONE

CONTENTS

1.1 Licence

The original code in `lass_utils` is dual-licenced under the terms of the GNU General Public Licence, version 2 (included as `COPYING.GPL`) and the terms of the licence(s) included in `LICENSE`.

This code is based on a reusable app template from DabApps, licenced under the 2-clause BSD licence; this licence and copyright notice is also included in `LICENSE`.

1.1.1 GPL Declaration

Copyright (C) 2012 University Radio York and contributors

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Why GPL2?

The various *django-lass-XYZ* modules were split from a project that was licenced under the GPL2 due to including `xapian-haystack`. The terms of this licence are preserved, but the copyright holders of the code decided to licence the original code (as separate from the work including xapian) under the terms of 2-clause BSD as well. Hence the kerfuffle.

1.1.2 This documentation

As this documentation is mostly extracted from the source, it is also distributed under the terms of 2BSD/GPL2 (for better or worse).

1.2 Contributors

This section lists contributors to the `metadata` package.

Where possible authors of contributed modules/apps used as dependencies to the project will be recognised; however this list may be outdated.

- Matt Windsor <matt.windsor@ury.org.uk> - Initial version code

1.3 Overview

In which the purpose of this app is explained, alongside pointers as to how to use it to add metadata to arbitrary models.

1.3.1 In short

This package provides metadata services that can be attached to existing models to add key-value metadata. It is used in the URY LASS project to store show names, descriptions, thumbnails and other such data.

1.3.2 What is metadata?

The `django-lass-metadata` system defines *metadata* as a key-value map whereby the same keys can be mapped to different values in different models. The keys are typically referred to by a string, but are themselves contained in a global model alongside some additional properties each key has.

Each collection of metadata attached to a model is known as a *metadata strand*, and is homogeneous (that is, every value inside it is of the same type).

Each strand is its own model, but the package provides convenience functions for spawning new models for This package provides both a generic metadata abstract model that can be used to define these homogeneous strand models, as well as predefined versions for textual and image-based metadata.

Keys are global

As mentioned before, there is only one key repository shared by all models; this was a design decision primarily for simplicity, but has the result of promoting a common language or taxonomy for all objects inside a domain.

Keys can be singular or multiple

Each key can decide whether or not multiple instances of itself in a strand are allowed, via the `metadata.models.MetadataKey.allow_multiple` field.

When retrieving a multiple-use metadatum, all values active during the reference date are returned in a list (and the empty list is used to represent no active data); otherwise the latest value is returned (and a `KeyError` is raised if none are active).

This distinction allows the metadata system to store items such as titles and descriptions unambiguously whilst also being usable for storing, for example, arbitrary tags or notes.

Metadata has history

Metadata has an *effective range*, which means that it can be set to come into and go out of force at given date-times. This allows the metadata system to track history of metadata, instead of just holding the current state.

Metadata has approvers and creators

Metadata also tracks who created it, and (optionally) who approved it for usage.

1.4 Details

This section of the documentation provides detailed information about the metadata system and how it works.

1.4.1 Usage

More documentation will hopefully be added to this later. Always check the API auto-docs for the latest insights into how to use this module.

How can I add metadata to a model?

Adding metadata requires the following steps:

1. Inherit your subject model from `metadata.mixins.MetadataSubjectMixin`. This pulls in the nice frontend code to allow metadata to be accessed in a mostly transparent way.
2. *(Optional)*. Override the `metadata.mixins.MetadataSubjectMixin.metadata_parent()` function in your subject to return a model instance whose metadata should be inherited by this model when an attempt to access nonexistent metadata (or any multiple-value metadata) happens. For more fine-grained control over metadata inheritance, try experimenting with the inheritance function system.
3. For each strand you want to attach, derive a model from the appropriate subclass of `metadata.models.GenericMetadata` using the `metadata.models.GenericMetadata.make_model()` function.
4. Override the `metadata.mixins.MetadataSubjectMixin.metadata_strands()` function in your subject to provide a dict mapping strand names to the related sets corresponding to the models you just created. (See the Django documentation for more information about related sets.)

How can I access metadata?

By default, the metadata of a subject model can be accessed through the `MetadataSubjectMixin.metadata` pseudo-field, which retrieves a dict-like object of all active metadata strands on that field. A method `MetadataSubjectMixin.metadata_at()` allows finer-grained control over this object, including overriding the current reference date (to look at historical or future metadata).

Each strand can then also be accessed like a dict, with metadata accessed via its key's name, ID or the key object itself as the dict key.

The first strand defined on a model is special: it by default serves as the fallback for any attribute requests on the subject itself. This allows you to specify, for example, `foo.title` instead of `foo.metadata['text']['title']`, and should help with migration to the metadata system from discrete fields.

1.4.2 Metadata packages

Sometimes you need to be able to apply entire collections of metadata to an item at a single time, or be able to link some metadata into one unit that you can update across a large amount of items of different types.

One example of where this might be useful is branding. If you have a certain brand that you want to apply to, say, a range of podcasts, you could manually set those podcasts up to have a consistent set of images, stock titles and descriptions, and so on. This is quite tedious and any changes made later won't automatically propagate to existing podcasts.

This is where *metadata packages* come in. Packages are metadata subject items that live in the `metadata.models.package.Package` model, which can be attached in a many-to-many relationship to your models using the `metadata.models.package.PackageEntry` attachable. (See the django-lass-utils documentation for information about how to use attachables.)

Using as a fallback for metadata subjects

The default *hooks* set can automatically try getting metadata from a metadata package if no item-specific metadata exists. Creating a method `packages` on the subject class that returns the reverse set of a package entry attachable will enable this. See the unit tests for an example.

Querying a package for metadata

`metadata.models.package.Package` implements `metadata.mixins.MetadataSubjectMixin`, so the *standard interface* for retrieving metadata works on them.

1.4.3 Hooks

Section author: Matt Windsor <matt.windsor@ury.org.uk>

The engine for running *metadata queries* is quite flexible and overridable for individual *metadata subject models*. It is based on the idea of running the query against a list of functions, known as *hooks*.

The default hooks set and support scaffolding for running hooks is found in the module `metadata.hooks`.

What a hook does

A hook is a function that takes a metadata query, processes it, and either returns the result of that query (if the hook can find metadata to fulfil it) or raises `metadata.hooks.HookFailureError` if it cannot fulfil it.

A hook can use *any* means necessary to find the metadata, including searching external files, caches, the database, or even recursively running metadata queries for other models. This makes the metadata system very general and very powerful.

How hooks lists are used

The hooks system runs lists of hooks in order from first to last with the query being passed to each.

If `metadata.hooks.HookFailureError` is raised, the hook runner skips to the next hook.

If the hook succeeds, the hook runner checks with the query to see if it can terminate with the result it has. If it can, it does so; otherwise, it runs either until it can or the runner hits the end of the hook list, at which point `metadata.hooks.QueryFailureError` is raised and the query fails.

The default hooks

When running a metadata query through the high-level *subject API*, unless the subject has overridden the default behaviour, the hooks in `metadata.hooks.DEFAULT_HOOKS` are called.

The current set of default hooks and their semantics can be found in the API documentation for **:py:module:‘metadata.hooks‘**.

1.4.4 The lower level

1.5 API Documentation

This section of the django-lass-metadata documentation is automatically generated from the code.

1.5.1 *metadata* - the LASS metadata system

The *metadata* app contains the generic metadata system used in LASS, which allows objects in the LASS model set to contain key-value stores of textual, image-based and other formats of metadata by inheriting a mixin and providing a subclass of the standard metadata models.

Metadata system

The *metadata* app is dedicated to the LASS *metadata system*, which allows various different *strands* of data (generally text, but also image-based and other formats) to be attached to items.

LASS uses the metadata system, for example, to provide shows with names and descriptions that have full recorded history and hooks for an approval system. It is also used to associate images (thumbnails and player insets) with podcasts.

Strands

Each model can have zero or more *strands* of metadata attached to it. Each strand is its own model (see below for more information on how to create a metadata provider model), and represents a specific collection of metadata on objects in the subject model.

Strands are indexed by name; an entire strand (as a dictionary-like object) can be retrieved from an implementor of *MetadataSubjectMixin* with `object.metadata()['strand-name']`. Generally, there will be a `text` strand containing all textual metadata, and an `images` strand containing thumbnail images and other related pictorial metadata.

Implementation

All metadata strands are implemented as key-value stores, the key store being implemented as one unified model for simplicity reasons and the value stores being separate for each strand for each model.

Two classes (*metadata.mixins.MetadataSubjectMixin* and *metadata.models.GenericMetadata*) provide the core framework for defining a metadata subject and a metadata strand. There are descendents of *GenericMetadata* available for specific commonly used strand types.

Examples

In the LASS project, examples of how to use the metadata system can be found in *schedule.models.show*, *ury-player.models.podcast* and *people.models.role*.

Models

The *metadata* module contains several model definitions under *metadata.models*.

Metadata system models

MetadataKey The MetadataKey model, which forms the key in the metadata key-value storage system.

class `metadata.models.key.`**`MetadataKey`**(**args*, ***kwargs*)

Bases: `lass_utils.models.type.Type`

A metadata key, which defines the semantics of a piece of metadata.

exception `DoesNotExist`

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MetadataKey.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

MetadataKey.**objects = <django.db.models.manager.Manager object at 0x254c350>**

MetadataKey.**packageimagemetadata_set**

MetadataKey.**packagetextmetadata_set**

GenericMetadata Generic metadata base class.

class metadata.models.generic.**GenericMetadata**(**args*, ***kwargs*)
Bases: people.mixins.approvable.ApprovableMixin, lass_utils.mixins.attachable.Attachabl
people.mixins.creatable.CreatableMixin, lass_utils.mixins.effective_range.EffectiveRa

An item of generic metadata.

NOTE TO IMPLEMENTORS: The final concrete instances of this class must have a field named 'element' that foreign key references the element of data to which the metadata is to be attached.

Any further implementors must also include a field named 'value' that stores the metadatum value.

class Meta
Bases: lass_utils.mixins.effective_range.Meta

abstract = False

GenericMetadata.**approver**

GenericMetadata.**creator**

GenericMetadata.**key**

GenericMetadata.**kwargs = {}**

GenericMetadata.**objects = <model_utils.managers._PassThroughManager object at 0x2dc1490>**

TextMetadata Models for the URY text metadata system.

To add metadata to a model, create a subclass of 'Metadata' for that model, descend the model from 'MetadataSubjectMixin', and fill out the methods identified in those two classes.

class metadata.models.text.**TextMetadata**(**args*, ***kwargs*)
Bases: metadata.models.generic.GenericMetadata

Abstract base for items of textual metadata.

class Meta
Bases: metadata.models.generic.Meta

abstract = False

TextMetadata.**approver**

TextMetadata.**creator**

TextMetadata.**key**

TextMetadata.**objects = <model_utils.managers._PassThroughManager object at 0x2dc8950>**

ImageMetadata Models for the URY image metadata system.

To add metadata to a model, create a subclass of 'ImageMetadata' for that model, descend the model from 'ImageMetadataSubjectMixin', and fill out the methods identified in those two classes.

class metadata.models.image.**ImageMetadata**(**args*, ***kwargs*)
Bases: metadata.models.generic.GenericMetadata

Abstract base class for usages of images as metadata.

class Meta
Bases: metadata.models.generic.Meta

abstract = False

ImageMetadata.**approver**

ImageMetadata.**creator**

ImageMetadata.**key**

ImageMetadata.**objects = <model_utils.managers._PassThroughManager object at 0x2dc8fd0>**

Other models

Type Common base class for type-of models.

In *LASS*, the pattern of dynamic lookup tables representing types and categories of other objects is very prevalent; in order to make using these type models more convenient, we have a common abstract base class for them.

class metadata.models.type.**Type**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

An abstract base class for models representing types, categories or other keyed collections of other models.

class Meta

abstract = False

classmethod Type.**get**(*identifier*)
User-friendly type get function.

This function uses the caching system to cache the type for a short amount of time.

If the input is an integer, it will be treated as the target type's primary key.

If the input is a string, it will be treated as the target type's name (case-insensitively).

If the input is an instance of cls itself, it will simply be returned.

Else, TypeError will be raised.

Parameters **identifier** (*string, integer or an element of the called class*) – an item of data representing the type to retrieve, or the type itself

Return type an element of the called class

classmethod Type.**get_or_404**(**args*, ***kwargs*)
Attempts to use *get* to retrieve an instance of this type, but returns *Http404* instead of *cls.DoesNotExist* if no object of the given type exists.

Arguments are passed verbatim to *get*; see the docstring for *get* for parameter details.

Mixins

The *metadata* module contains mixins for adding metadata to existing models and suchlike.

Misc

Administration hooks

Administration system hooks for the *metadata* application.

class metadata.admin.**MetadataKeyAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

An adminstration snap-in for maintaining the list of metadata keys.

list_display = ('name', 'description', 'allow_multiple')

media

class metadata.admin.**PackageAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

An administration snap-in for maintaining the list of packages.

inlines = [<class 'metadata.admin.PackageTextMetadataInline'>, <class 'metadata.admin.PackageImageMetadata

list_display = ('name', 'description', 'weight')

media

class metadata.admin.**PackageImageMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.ImageMetadataInline

media

model
alias of PackageImageMetadata

class metadata.admin.**PackageTextMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.TextMetadataInline

media

model
alias of PackageTextMetadata

metadata.admin.**register**(*site*)
Registers the metadata admin hooks with an admin site.

Unit tests

Test suite for the metadata package.

TODO: Cache tests

class metadata.tests.**MetadataSubjectTest**(**args*, ***kwargs*)
Bases: django.db.models.base.Model, metadata.mixins.metadata_subject.MetadataSubjectMix

Test metadata subject model.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MetadataSubjectTest.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

classmethod MetadataSubjectTest.**make_foreign_key**(*nullable=False*)

MetadataSubjectTest.**metadata_strands**()

MetadataSubjectTest.**metadatasubjecttestpackageentry_set**

MetadataSubjectTest.**metadatasubjecttesttextmetadata_set**

MetadataSubjectTest.**objects = <django.db.models.manager.Manager object at 0x32b2310>**

MetadataSubjectTest.**packages**()

MetadataSubjectTest.**range_start**()

MetadataSubjectTest.**testimagemetadata_set**

class metadata.tests.**MultipleMetadataDictTest**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests to see if the dictionary access method for metadata is functional on multiple-entry metadata.

fixtures = ['test_people', 'metadata_test']

test_image_get()
Tests whether getting image metadata works.

test_text_get()
Tests whether getting textual metadata works.

test_text_in()
Tests whether the metadata view supports 'in'.

class metadata.tests.**PackageTest**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests to see if the metadata packages system is hooked in correctly and used as a fallback metadata source.

fixtures = ['test_people', 'metadata_test', 'package_test']

test_text()
Tests whether the package is used to provide default textual metadata.

class metadata.tests.**SingleMetadataDictTest**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests to see if the dictionary access method for metadata is functional on single-entry metadata.

fixtures = ['test_people', 'metadata_test']

test_default()
Tests whether default metadata works as expected.

test_effective_range()
Tests whether the single metadata getting system correctly respects the *effective_from* and *effective_to* bounds.

test_image_get()
Tests whether getting image metadata works.

test_text_get()
Tests whether getting textual metadata works.

test_text_in()
Tests whether the metadata view supports 'in'.

admin_base

Base admin classes for metadata administration.

class metadata.admin_base.**GeneralMetadataInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.TabularInline

Base inline class for anything that's like metadata.

extra = 0

formfield_for_foreignkey(*db_field*, *request*, ***kwargs*)
Provides a form field for foreign keys.

Overrides the normal inline so that submitter and approver are set, by default, to the currently logged in user.

media

class metadata.admin_base.**ImageMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.MetadataInline

Specialisation of MetadataInline for image metadata.

media

verbose_name = 'associated image'

verbose_name_plural = 'Associated images'

class metadata.admin_base.**MetadataAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

Base class for metadata admin-lets.

date_hierarchy = 'effective_from'

list_display = ('element', 'key', 'value', 'creator', 'approver', 'effective_from', 'effective_to')

media

class metadata.admin_base.**MetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.GeneralMetadataInline

Base inline class for metadata inline admin-lets.

fields = ('key', 'value', 'effective_from', 'effective_to', 'approver', 'creator')

media

class metadata.admin_base.**PackageEntryInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.GeneralMetadataInline

Snap-in for editing package entries inline.

fields = ('package', 'effective_from', 'effective_to', 'approver', 'creator')

media

verbose_name = 'branding package'

verbose_name_plural = 'branding packages'

class metadata.admin_base.**TextMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.MetadataInline

Specialisation of MetadataInline for text metadata.

media

verbose_name = 'associated text item'

verbose_name_plural = 'associated text items'

Hooks

Default hooks for the metadata system, as well as a function for running metadata hooks on queries.

exception metadata.hooks.**HookFailureError**(*value*)
Bases: exceptions.Exception

Exception raised when a metadata hook fails to fulfil a query, but not in a way that should halt execution.

exception metadata.hooks.**QueryFailureError**(*value*)
Bases: exceptions.Exception

Exception raised when a metadata query returns no results, but was expected to do so.

metadata.hooks.**get_active_metadata**(*strand_set*, *key*, *date*)
From the given queryset, extracts metadata matching the given key that was active at the given date.

metadata.hooks.**get_strand_set**(*query*)
Attempts to get a metadata strand related-set from the element of the given query, given the query's requested strand.

metadata.hooks.**handle_set**(*metadata*, *allow_multiple*, *query_type*)
Handles a metadata set as required by the metadata's multiplicity and the metadata query type.

Parameters

- **metadata** (*QuerySet*) – A set of metadata.
- **allow_multiple** (bool) – Whether or not multiple values should be returned, in the case of the query type being VALUE.
- **query_type** – The query type, which determines the behaviour expected of this function.

metadata.hooks.**metadata_from_cache**(*query*)
Given a metadata query, attempts to fulfil the request using the Django cache.

Parameters **query** (metadata.query.MetadataQuery or similar object.) – The MetadataQuery this hook is trying to run.

metadata.hooks.**metadata_from_default**(*query*)
Given a metadata query, attempts to return the default value for the metadata key in the given strand.

Will raise metadata.hooks.HookFailureError on failure.

Parameters **query** (metadata.query.MetadataQuery or similar object.) – The MetadataQuery this hook is trying to run.

metadata.hooks.**metadata_from_package**(*query*)
Given a metadata query, attempts to use the query element's designated metadata packages to fulfil the request.

Will raise metadata.hooks.HookFailureError on failure.

Parameters **query** (metadata.query.MetadataQuery or similar object.) – The MetadataQuery this hook is trying to run.

metadata.hooks.**metadata_from_parent**(*query*)
Given a metadata query, attempts to use the query element's designated parent to fulfil the request.

Will raise metadata.hooks.HookFailureError on failure.

Parameters **query** (metadata.query.MetadataQuery or similar object.) – The MetadataQuery this hook is trying to run.

metadata.hooks.**metadata_from_strand_sets**(*query*)
Given a metadata query, attempts to use the query element's own metadata sets to fulfil the request.

Will raise metadata.hooks.HookFailureError on failure.

Parameters **query** (metadata.query.MetadataQuery or similar object.) – The MetadataQuery this hook is trying to run.

metadata.hooks.**run_query**(*query*, *hooks=None*)
Runs a metadata query, optionally with the given set of hooks.

If hooks is None, the default set will be used. (See metadata.hooks.DEFAULT_HOOKS)

Parameters

- **query** (metadata.query.MetadataQuery or similar object.) – A metadata query.
- **hooks** (*iterable, for example list*) – A set of hooks to run the query with.

Queries

In which a *metadata query* is defined.

class `metadata.query.`**`MetadataQuery`**(*subject*, *date*, *key*, *strand='text'*, *query_type=0*)
Bases: `object`

An object that holds together all state required for a query for metadata on a metadata subject to be run.

For running a metadata query, see the `metadata.hooks` module.

Generally, however, most code shouldn't need to use metadata queries directly. A sugary frontend is provided by `metadata.mixins.MetadataSubjectMixin`.

`cache_key`()
Returns a representation of the query that can be used as a cache key.

Return type basestring

`date`
Returns the target metadata date of this query.

This will be the time of query creation if no date was specified.

`initial_state`()
Returns the value that should be the initial state of any running of this query.

`join`(*old*, *new*)
Join two successful query runnings in a way that satisfies the query's requirements.

The old state is given precedence so, for example, trying to join two results for a value query on a single-value key will return the old state only.

Parameters

- **old** – the old answer to this query
- **new** – the new answer to this query

`replace`(***kwargs*)
Creates a new query representing the query with the initialising parameters in kwargs replacing their counterparts in this query.

Parameters **kwargs** – A keyword argument dict of substitutions to make.

`satisfied_by`(*result*)
Given an intermediate result of a metadata query run, returns True if there need not be any more hook processing in order to get a valid answer to the query.

Parameters **result** – the current result of a metadata query run

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

m