Aristotle Metadata Registry Documentation

Release 0.0.1

Samuel Spencer

Aug 08, 2018

Contents

Aristotle Metadata Registry is an open-source metadata registry framework as laid out by the requirements of the ISO/IEC 11179:2013 specification.

Aristotle Metadata Registry represents a new way to manage and federate content built on and extending the principles of leading metadata registry. The code of Aristotle-MDR is completely open-source, building on the Django web framework and the mature model of the ISO 11179 standard, agencies can easily run their own metadata registries while also having the ability to extend the information model and tap into the permissions and roles defined in ISO 11179.

By allowing organisations to run their own independent registries they are able to expose authoritative metadata and the governance processes behind its creation, and building upon known and open systems agencies, can build upon a stable platform or the sharing of 11179 metadata items.

CHAPTER 1

Table of Contents

1.1 Aristotle Metadata Registry Mission Statement

The core principle behind the design of the Aristotle Metadata Registry is to build a framework for building ISO/IEC 11179 compliant Metadata Registries, using 100% Free Open Source Software, and released to the public as Free Open Source Software.

By designing Aristotle-MDR in an extensible way, the core data model of Aristotle aims to be as close to the model of ISO/IEC 11179-3, without burdening the framework with unnecessary code.

Aristotle-MDR is designed to provide the framework for a metadata registry, and is explicitly *not* designed to be a standard web content management system, and a core assumption in the design of Aristotle is that the management of 'non-metadata' content is a matter for each party installing Aristotle-MDR to handle independent of the registry.

There are some simple url hooks available in Aristotle for including extra pages using the django template system, alternatively Django Packages has a list of a number of excellent CMS packages for Django. Many of these should be able slot in besides the Aristotle-MDR app in a custom site, without having to alter the code or compromise the core principles of Aristotle.

1.2 Installing Aristotle Metadata Registry

1.2.1 Easy installer documentation

This is a quick guide to setting up a new metadata registry based on the Aristotle Metadata Registry framework using the easy installer.

Such a server should be considered for *demonstration* purposes, and deployment should be done in accordance with the best practices and specific requirements of the installing agency.

For more information on configuring a more complete installation review the help article *Integrating Aristotle-MDR with a Django project*.

1. Make sure you have a server setup for hosting the project with an appropriate WSGI web server configured. If the server is only used for development, the inbuilt django server can be accessed by running the `./manage.py runserver` command.

   PythonAnywhere also provides a free python server suitable for development and low traffic sites.

2. (Optional but recommended) Configure a `virtualenv` for your server, so that the dependancies for Aristotle-MDR do conflict any other software you may be running. If you are running Aristotle on an isolated server with root privileges you may skip this step.

   For PythonAnywhere, information is available on installing virtualenv and configuring a new virtualenv.

3. Next install the Aristotle Metadata Registry package. This can be done using pip with the following command `pip install aristotle-metadata-registry`. If you already have a version installed your can update with `pip install -U aristotle-metadata-registry`

4. To run the easy installer simply run `aristotle-installer` from the command line. There are a number of command line arguments that are explained in the help documentation which can be accessed from the command line:

   ```
   ``aristotle-installer --help``
   ```

   To install your registry in a different directory use the –dir option `python install.py --dir ./myregistry`

   This installer will setup an example registry, and will prompt you for a new name, ask for a few additional settings, install requirements, setup a database and collect the static files.

5. If required, browse to the directory of your project that was named in the above directory, and edit the `settings.py` files to meet your requirements. Although the installer generates a pseudo-random hash for the `SECRET_KEY`, **It is strongly recommmended you generate a fresh** `SECRET_KEY`. Also consider which customisations to implement using the options in the `ARISTOTLE_SETTINGS` dictionary - details of which can be found under *Configuring the behavior of Aristotle-MDR*.

   The example registry includes commented out lines for some useful Aristotle-MDR extensions. If you wish to use these, remove the comments as directed by the documentation in `settings.py`.

6. If you are using a WSGI server (such as PythonAnywhere) you'll need to either point your server to the projects `wsgi.py` file or update your WSGI configuration.

   For more information on configuring the PythonAnywhere WSGI server review their documentation.

7. Start (or restart) the development server and visit its address. In the case of a local development server this will likely be `127.0.0.1`.

Using a different database

The easy installer using a simple SQLite database for storing content, however for large scale production servers with multiple concurrent users this may not be appropriate. Django supports a wide range of database server which can be used instead of SQLite. However to the very specific nature of the options required to connect to a database, to use an alternate database with the easy installer a few additional steps are required.

1. Let the installer run to completion, without the `--dry` option, and selecting yes when asked `Ready to install requirements? (y/n):`.

2. Edit your `settings.py` file and add a variable `DATABASES` set to connect to your database as described in the Django documentation.

3. Remove the `pos.db3` file that will have been created during the installation. This file is the name of the default SQLite database and can be safely deleted without any issues.

4. Call the Django `migrate` command again using the updated settings:

```
./manage.py migrate
```

5. Start (or restart) the development server and visit its address. In the case of a local development server this will likely be `127.0.0.1`.

Disabling the DEBUG options

Because of the The easy installer using a simple SQLite database for storing content, however for large scale production servers with multiple concurrent users this may not be appropriate. Django supports a wide range of database servers which can be used instead of SQLite. However to the very specific nature of the options required to connect to a a database, to use an alternate database with the easy installer a few additional steps are required.

1. Let the installer run to completion, without the `--dry` option, and selecting yes when asked `Ready to install requirements? (y/n):`.
2. **Edit your `settings.py` file and set the `DEBUG` to False::** DEBUG=False
3. Remove the `pos.db3` file that will have been created during the installation. This file is the name of the default SQLite database and will have a number of example objects and users created within it as the migrate step when `DEBUG` is set to `True`.
4. Call the Django `migrate` command again using the updated settings:

```
./manage.py migrate
```

5. Start (or restart) the development server and visit its address. In the case of a local development server this will likely be `127.0.0.1`. To access the administators sections of the site you will need to create a super user.

Creating a superuser for the registry

Creating a superuser is covered in more depth in the Django documentation, however a quick guide is given here. These steps assume a valid database exists and has been appropriately set up with the Django `migrate` command.

To create a super user, browse to the project folder and run the command:

```
$ django-admin createsuperuser
```

This will prompt you for a username, email and password.

A username and email can be applied with the `--username` and `--email` switches respectively. For example:

```
$ django-admin createsuperuser  --username=my_registry_admin --email=admin@registry.
↪example.gov
```

1.2.2 Integrating Aristotle-MDR with a Django project

Note: this guide relies on some experience with Python and Django. For new users looking at getting a site up and running look at the *Easy installer documentation*.

The first step is starting a project as described in the Django tutorial. Once this is done, follow the steps below to setup Aristotle-MDR.

1. Add "aristotle_mdr" to your INSTALLED_APPS setting like this:

```
INSTALLED_APPS = (
    ...
    'haystack',
    'aristotle_mdr',
    ...
)
```

To ensure that search indexing works properly *haystack* **must** be installed before *aristotle_mdr*. If you want to take advantage of Aristotle's WCAG-2.0 access-key shortcut improvements for the admin interface, make sure it is installed *before* the django admin app.

2. Include the Aristotle-MDR URLconf in your project urls.py. Because Aristotle will form the majority of the interactions with the site, as well as including a number of URLconfs for supporting apps its recommended to included it at the server root, like this:

```
url(r'^/', include('aristotle_mdr.urls')),
```

3. Create the database for the metadata registry using the Django migrate command:

```
python manage.py migrate
```

4. Start the development server with `python manage.py runserver` and visit http://127.0.0.1:8000/ to see the home page.

For a complete example of how to successfully include Aristotle, see the *example_mdr* directory.

1.2.3 Configuring the behavior of Aristotle-MDR

Environment variables

The default django settings file for Aristotle-MDR looks for a number of enviromnet variables for storing files or configuring your webapp. These are all prefixed with `aristotlemdr:`.

`BASE_DIR` Defaults to the path of where Aristotle is installed. Its highly adviced this is changed.

`SECRET_KEY` Defaults to a very insecure value - you MUST change this before going into production. From Django settings documentation:

> A secret key for a particular Django installation. This is used to provide cryptographic signing, and should be set to a unique, unpredictable value.

`STATIC_ROOT` Defaults to the value of `BASE_DIR + "/static"` From Django settings documentation:

> A secret key for a particular Django installation. This is used to provide cryptographic signing, and should be set to a unique, unpredictable value.

`MEDIA_ROOT` Defaults to the value of `BASE_DIR + "/media"` From Django settings documentation:

> Absolute filesystem path to the directory that will hold user-uploaded files.

`ARISTOTLE_SETTINGS`

The following are required within a dictionary in the settings for the configured Django project.

`CONTENT_EXTENSIONS` A list of the *namespaces* used to add additional content types, these are used when discovering the available extensions for about pages - required format a `list` of `strings`.

BULK_ACTIONS A list of fully-qualified python paths to the bulk action form classes that provide the action. *More information on configuring bulk actions is available here*.

PDF_PAGE_SIZE The default page size to deliver PDF downloads if a page size is not specified in the URL

SEPARATORS A key:value set that describes the separators to be used for name suggestions in the admin interface. These are set by specifying the key as the django model name for a given model, and the value as the separator. When a value for a model isn't stated in this field it defaults to a hyphen `-`. The default settings in `required_settings.py` set additional defaults and specify the separator for "DataElements" as a comma with a single space `, `` and the separator for "DataElementConcepts" as an em-dash ``–`.

SITE_NAME The main title for the site - required format `string` or `unicode`.

SITE_BRAND A URL to the logo to use for the site, this can be relative or absolute.

SITE_INTRO The introductory text use on the home page as a prompt for users - required format `string` or `unicode`.

WORKGROUP_CHANGES An array that specified which classes of user can move items between workgroups. Possible options include `'admin'`, `'manager'` or `'submitter'`.

DOWNLOADERS A list of download options - explained below:

ARISTOTLE_SETTINGS.DOWNLOADERS

This is a **list of tuples** that define the different download options that will be made available to users. This tuple defines in order:

- filetype used in the URL to catch this download type - must match the regex `[a-zA-Z0-9\-\.]+`.
- the name presented on the download menu
- The Font-Awesome icon used in the download menu
- the python module that includes the `downloader.py` file for handling this filetype

For example:

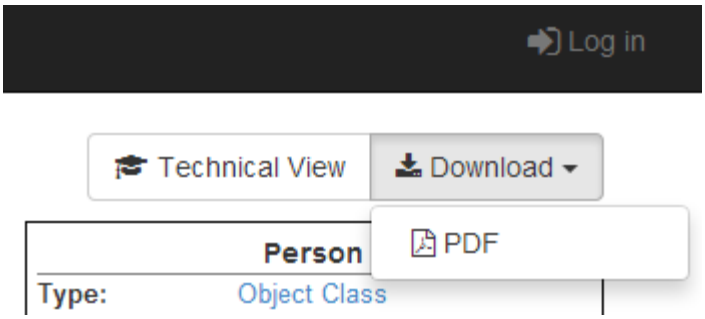
```
('pdf',"PDF","fa-file-pdf-o","aristotle_mdr")
```

Menu options are only given if a template for that download file type exists for a given object. The first (filetype) setting is used when catching URLs for downloads, so that when resolving URLs the filetype is used in the URL in the following way:

```
/download/<download-file-type>/<item-id>
```

This file type is also passed to the download manager for this filetype, so that multiple file types can be handled by the same extension.

For example, if an object class had a PDF template, based on the above configuration the menu below would be accessible:



And clicking this would access the following relative URL:

```
/download/pdf/<object_class_id>
```

For more information on creating additional download extensions consult the guide on *Adding new download formats*.

Sample settings

Below is the `ARISTOTLE_SETTINGS` used on the hosted Aristotle example:

```
ARISTOTLE_SETTINGS = {
   # 'The main title for the site.'
    'SITE_NAME': 'Aristotle Metadata Registry',
   # URL for the Site-wide logo
    'SITE_BRAND': '/static/aristotle_mdr/images/aristotle_small.png',
   # 'Intro text use on the home page as a prompt for users.'
    'SITE_INTRO': 'Use Aristotle Metadata to search for metadata...',
   # Extensions that add additional object types for search/display.
    'CONTENT_EXTENSIONS' : [ 'comet' ],
   # Separators for auto-generating the names of constructed items.
    'SEPARATORS': { 'DataElement':',',
                    'DataElementConcept':'-'},
    'DOWNLOADERS': [
        ('pdf','PDF','fa-file-pdf-o','aristotle_pdf'),
    ]
  }
```

1.2.4 Adding new static pages into Aristotle

While Aristotle provides a strong framework for setting up a metadata registry, there some static pages which are important for a site, but unlikely to be changed, such as the home page, CSS and about pages.

These exist in aristotle as template pages, and like all Django tempaltes are easy to override with more custom, site-specific content. The first step is to ensure the settings for the site include a Django `TEMPLATE_DIR` directive, like that below:

```
TEMPLATE_DIRS = [os.path.join(BASE_DIR, 'templates')]
```

Setting a separate template directory when using Aristotle ensure that templates can be easily overriden, without requiring a separate django app or editing of the main Aristole codebase.

When attempting to resolve templates, one of the first locations checked will be the directory stated in `TEMPLATE_DIRS`. Examining the code in the Aristotle-MDR code should give an understanding of how the templates are laid out if changes are necessary.

1.2.5 Changing the look and feel of the site

Changing site CSS using Django `staticfiles`

Changing the CSS of the site can be done by overriding the static files that serve the Bootstrap and Aristotle CSS files, these are available at:

```
aristotle_mdr/static/aristotle_mdr/css/aristotle.css
aristotle_mdr/static/aristotle_mdr/bootstrap/bootstrap.min.css
```

Overriding these will require setting the `STATICFILES_DIR` setting in `settings.py` , like so:

```
STATICFILES_DIR = [os.path.join(BASE_DIR, "site_static")]
```

Its important, to make sure if setting a `STATICFILES_DIR` that `'django.contrib.staticfiles.finders.FileSystemFinder'` is added to the `STATICFILES_FINDERS` setting. If importing all of the settings from Aristotles *required_settings.py* file this is already included, so this doesn't need to be redefined. But if `settings.py` doesn't import `required_settings.py`, `STATICFILES_FINDERS` can be declared like this:

```
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
)
```

Once this is set, to override the Aristotle bootstrap css, a file at the following location in the project site will be used instead:

```
custom_site_static/aristotle_mdr/bootstrap/bootstrap.min.css
```

More information about these is available in the Django documentation on static files.

Changing the Bootstrap file by overriding the settings

Aristotle uses Django-bootstrap3 to import bootstrap. By default Aristotle stores the boostrap file at:

```
/static/aristotle_mdr/bootstrap/
```

but, an alternative solution is to override this value be redefining the *BOOTSTRAP3* setting in your projects `settings.py`, like so:

```
BOOTSTRAP3 = {
    # The Bootstrap base URL
    'base_url': '/static/your_path_to/bootstrap/',
}
```

Completely overhauling the site

It is also possbile to override the home page and base templates to completely overhaul the look and feel of the site, and these are available under the `templates` directory at:

- `aristotle_mdr/templates/aristotle_mdr/base.html`
- `aristotle_mdr/templates/aristotle_mdr/static/home.html`

However, doing so may break the rendering of pages and prevent the registry from working. It is strongly recommended that overrides of these files are done by someone with a strong working knowledge of HTML, CSS and Django templates.

1.2.6 Configuring third-party apps

Aristotle takes care of most of the work of getting a registry setup with the settings import:

```
from aristotle_mdr.required_settings import *
```

but there are few areas for customisation or tweaking.

Django

Every django setting can be overridden, but the ones that will be most important when configuring Aristotle-MDR are:

- `DATABASE` - By default Aristotle will configure a SQLite file-based database. While this is fine for very small low-traffic registries, configuring Django to use a fully-fledged relational database management system like PostgreSQL or MySQL will be better for larger, high-traffic sites.
- `ROOT_URLCONF` - This is the python library that will be used to define the settings for django to resolve URLs. If you aren't using any extensions, you can just leave this as the default which points to the Aristotle URLs file - `aristotle_mdr.urls`. If you are using extensions, you'll need to point this at the URLs file that you have created to handle all of the different URL configuration files for each extension.
- `WSGI_APPLICATION` - This points to the file and WSGI application that you have created to if you are intending to deploy via a WSGI server.

Haystack

For search to work, Haystack is required to be installed. There are no options to disable this, as without search a registry is quite useless. However you can change some settings.

- `HAYSTACK_SEARCH_RESULTS_PER_PAGE` - Self explanatory, this defaults to 10 items per page.
- `HAYSTACK_CONNECTIONS` - This define which search indexers are being used and how they are connected. By default this uses the Whoosh Engine, which is quite fast and because its a Pure-Python implementation reduces the complexity in getting it setup. For more advanced usage, read the Haystack documentation.
- `HAYSTACK_SIGNAL_PROCESSOR` - Included for completion, this defaults to `aristotle_mdr.contrib.help.signals.AristotleHelpSignalProcessor`. This is a custom signal processor that performs real-time, status-aware changes to the index and monitors for changes to Help Pages. The alternative recommended option is `aristotle_mdr.signals.AristotleSignalProcessor`, which only monitors changes to metadata items. **Read the warnings below for why you probably only want to use these options.**

Warnings about Haystack

- Always make sure `haystack` is included **once and only once** in `INSTALLED_APPS`, otherwise your installation will throw errors.
- Make sure `haystack` is included in `INSTALLED_APPS` *before* `aristotle_mdr`.

- Be aware that Haystack will only update search indexes when told, Aristotle includes a `SignalProcessor` that performs registation status-aware real-time updates to the index. **Switching this for another processor may expose private information** through search results, *but will not allow unauthorised users to access the complete item*.

LESS Compilation

Aristotle-MDR includes a number of uncompiled LESS files that need to be compiled by django-static-precompiler. By default Aristotle-MDR uses the Python-based `lesscpy` compiler for this which is approximately compatible, but slower than, to the Node `lessc` compiler. If you have complex requirements in your custom LESS files, want a faster compile time or wish to use another CSS precompile type, override the following setting in your `settings.py`:

```
STATIC_PRECOMPILER_COMPILERS = (
  ('static_precompiler.compilers.LESS', {"executable": "lesscpy"}),
)
```

In production, its advisable to compile the LESS files *once* and cache these withother static files. This makes the choice of precompiler less of an issue for production environments.

1.2.7 Technical requirements

The Aristotle Metadata Registry is built on the Django framework which supports a wide range of operating systems and databases. While Aristotle-MDR should support most of these only a small set of configurations have been thoroughly tested on the Travis-CI continuous integration systems as "supported infrastucture".

Operating system support

- Ubuntu Linux (Precise Pangolin) 12.04 LTS (verification courtesy of Travis-CI)

Travis-CI does not yet have containerised support for the Ubuntu 14.04 or 16.04 long-term support releases.

Python

Only the latest releases of Python are supported. New users are recommended to use Python 3.5 or above.

- Python 3.5+

Django

- Django version 1.11 LTS

Database support

- SQLite
- Postgres
- MariaDB

Notes:

Aristotle has been tested against Microsoft SQL Server 2016 on Windows, but we no longer provide official testing against this database.

MySQL has issues incompatible with Aristotle that prevent it from being used. Consider using an alternative like MariaDB if you need MySQL-like support.

Search index support

- Elasticsearch 5.0+ (Only tested on Linux)
- Whoosh (Linux and Windows)

1.3 Features of Aristotle-MDR

1.3.1 100% Free open-source software

The entire suite of software that Aristotle-MDR is built upon is free open-source software. The majority of these requirements are managed through the Python Package Index, the rest are online resources, such as jQuery, Twitter Bootstrap CSS Framework and Font Awesome are hosted through online content delivery networks for improved speed.

Because of the open nature, low-requirements and no-cost of this framework a new registry can be setup on a Python shared hosting service like Python Anywhere in a matter of minutes - and includes everything you need to get a professional scalable metadata registry up and running.

The only restriction with running Aristotle-MDR is that if you are running a public facing site, you keep a link to the Aristotle GitHub page in the footer, but even this can can be waived with permission.

1.3.2 Easily extensible

One of the core features of the ISO/IEC 11179-3 information model is the ability to extend the models by subclassing from the included items. Aristotle-MDR captures the core of the ISO/IEC 11179 as faithfully as possible, but provides a rich API to quickly and easily add new items for management using the Object-Oriented approach of Python and Django.

You can read more about the *content type API* and *template overrides* in the *extensions documentation*.

1.3.3 Mobile-friendly interface

Every page of Aristotle-MDR has been built upon the Twitter Bootstrap CSS Framework. This means that every page has a responsive, mobile first design and flawlessly scales from the largest desktop to the smallest phone. Along with this, the use of the Font Awesome icon toolkit means menus and pages have a consistent look and feel.

1.3.4 Real-time enterprise search

Integration of the Django-Haystack search API provides a rich search engine capability, so content can always be found. Content can be search on not just by text fields, but also by Registration Status, the owner workgroup and content type, with more advanced search options to come!

Although the default settings for Aristotle use the Whoosh search engine, Haystack provides backend hooks for a number enterprise ready search engines.

The default settings for Aristotle-MDR include a real-time search index manager that tracks changes as they are made, and updates visibilty and indexes immediately.

With appropriate tweaking the Haystack engine can scale from the smallest research facility to the largest government agency.

1.3.5 Secure, thoroughly tested permissions

Using a set of thoroughly tested custom permissions, content created within the Aristotle registry can be show or hidden from the public and registered users based on the well documented status workflow in part 6 of the ISO/IEC 11179 standard.

Strict version control of the code on GitHub, continuous testing of the code using Travis-CI and code coverage analysis using Coveralls.io ensures that access permissions are clearly defined, and as changes are made if issues with permissions they can be idenitified and rectified immediately.

1.3.6 Easy content creation

Aristotle-MDR includes an easy to use editing system, that uses the robust CKEditor WYSIWYG (What-You-See-Is-What-You-Get) editor, that gives users instant feedback on changes to content. The in-built editor gives access to a rich-text editor, easy insertion of links to content and an image upload and linking facility.

1.4 Extending Aristotle-MDR

One of the core features of the ISO/IEC 11179-3 information model is the ability to extend the models by subclassing from the included items. The core item that most 11179 objects are based on is the "Concept".

Due to this encouragement of inheritance and enhancement, Aristotle-MDR follows similar principles and uses the Object-Oriented approach of Python and Django, to expose a rich API that makes adding new content, and altering templates quick and easy.

Before starting it is strongly encouraged that you have a clear understanding of the Python programming language as well as how to build Django apps and sites.

1.4.1 Making new metadata types

Most of the overhead for creating new item types in Aristotle-MDR is taken care of by inheritance within the Python language and the Django web framework.

Making new item types

Most of the overhead for creating new item types in Aristotle-MDR is taken care of by inheritance within the Python language and the Django web framework.

For example, creating a new item within the registry requires as little code as:

```
import aristotle_mdr
class Question(aristotle_mdr.models.concept):
    questionText = models.TextField()
    responseLength = models.PositiveIntegerField()
```

This code creates a new "Question" object in the registry that can be progressed like any standard item in Aristotle-MDR. Once the the appropriate admin pages are set up, from a usability and publication standpoint this would be indistinguishable from an Aristotle-MDR item, and would instantly get a number of *features that are available to all Aristotle 'concepts' without having to write any additional code*

Once synced with the database, this immediately creates a new item type that not only has a `name` and `description`, but also can immediately be associated with a workgroup, can be registered and progressed within the registry and has all of the correct permissions associated with all of these actions.

Likewise, creating relationships to pre-existing items only requires the correct application of Django relationships such as a `ForeignKey` or `ManyToManyField`, like so:

Listing 1.1: mymodule.models.Question

```
import aristotle_mdr
from django.db import models


class Question(aristotle_mdr.models.concept):
    template = "extension_test/concepts/question.html"
    questionText = models.TextField(blank=True, null=True)
    responseLength = models.PositiveIntegerField(blank=True, null=True)
    collectedDataElement = models.ForeignKey(
        aristotle_mdr.models.DataElement,
        related_name="questions",
        null=True,
        blank=True
    )
```

This code, extends our Question model from the previous example and adds an optional link to the ISO 11179 Data Element model managed by Aristotle-MDR and even adds a new property on to Data Elements, so that `myDataElement.questions` would return of all Questions that are used to collect information for that Data Element. Its also possible to *include content from objects across relations on other pages* without having to alter the templates of other content types. For example, this would allow pertinant information about questions to appear on data elements, and vice versa.

Customising the edit page for a new type

To maintain consistancy edit pages have a similar look and feel across all concept types, but some customisation is possible. If one or more fields should be hidden on an edit page, they can be specified in the `edit_page_excludes` property of the new concept class.

An example of this is when an item specifies a ManyToManyField that has special attributes. This can be hidden on the default edit page like so:

```
class Questionnaire(aristotle_mdr.models.concept):
    edit_page_excludes = ['questions']
    questions = models.ManyToManyField(
            Question,
            related_name="questionnaires",
            null=True,blank=True)
```

Including additional items when downloading a custom concept type

`concept.get_download_items()`

When downloading a concept, extra items can be included for download by overriding the `get_download_items` method on your item. By default this returns an empty list, but can be modified to include any number of items that inherit from `_concept`.

When overriding, each entry in the list must be a two item tuple, with the first entry being the python class of the item or items being included, and the second being the queryset of items to include.

For example:

Listing 1.2: mymodule.models.Questionnaire.get_download_items

```
    def get_download_items(self):
        return [
            (
                Question,
                self.questions.all().order_by('name')
            ),
            (
                aristotle_mdr.models.DataElement,
                aristotle_mdr.models.DataElement.objects.filter(questions__
↪questionnaires=self).order_by('name')
            ),
        ]
```

Caveats: `concept` versus `_concept`

There is a need for some objects to link to any arbitrary concept, for example the favourites field of *aristotle.models.AristotleProfile*. Because of this there is a distinction between the Aristotle-MDR model objects `concept` and `_concept`.

Abstract base classes in Django allow for the easy creation of items that share similar properties, without introducing additional fields into the database. They also allow for self-referential ForeignKeys that are restricted to the inherited type, rather than to the base type.

class aristotle_mdr.models.**_concept**(**args*, ***kwargs*)

9.1.2.1 - Concept class Concept is a class each instance of which models a concept (3.2.18), a unit of knowledge created by a unique combination of characteristics (3.2.14). A concept is independent of representation.

This is the base concrete class that `Status` items attach to, and to which collection objects refer to. It is not marked abstract in the Django Meta class, and **must not be inherited from**. It has relatively few fields and is a convenience class to link with in relationships.

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **uuid** (*UUIDField*) – Universally-unique Identifier. Uses UUID1 as this improves uniqueness and tracking between registries
- **name** (*TextField*) – The primary name used for human identification purposes.
- **definition** (*RichTextUploadingField*) – Representation of a concept by a descriptive statement which serves to differentiate it from related concepts. (3.2.39)
- **workgroup_id** (*ForeignKey*) – Workgroup
- **submitter_id** (*ForeignKey*) – This is the person who first created an item. Users can always see items they made.
- **_is_public** (*BooleanField*) – is public

- **_is_locked** (*BooleanField*) – is locked
- **version** (*CharField*) – Version
- **references** (*RichTextUploadingField*) – References
- **origin_URI** (*URLField*) – If imported, the original location of the item
- **comments** (*RichTextUploadingField*) – Descriptive comments about the metadata item (8.1.2.2.3.4)
- **submitting_organisation** (*CharField*) – Submitting organisation
- **responsible_organisation** (*CharField*) – Responsible organisation
- **superseded_by_id** (ConceptForeignKey) – Superseded by

class aristotle_mdr.models.**concept**(**args*, ***kwargs*)

This is an abstract class that all items that should behave like a 11179 Concept **must inherit from**. This model includes the definitions for many long and optional text fields and the self-referential `superseded_by` field. It is not possible to include this model in a `ForeignKey` or `ManyToManyField`.

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **uuid** (*UUIDField*) – Universally-unique Identifier. Uses UUID1 as this improves uniqueness and tracking between registries
- **name** (*TextField*) – The primary name used for human identification purposes.
- **definition** (*RichTextUploadingField*) – Representation of a concept by a descriptive statement which serves to differentiate it from related concepts. (3.2.39)
- **workgroup_id** (*ForeignKey*) – Workgroup
- **submitter_id** (*ForeignKey*) – This is the person who first created an item. Users can always see items they made.
- **_is_public** (*BooleanField*) – is public
- **_is_locked** (*BooleanField*) – is locked
- **version** (*CharField*) – Version
- **references** (*RichTextUploadingField*) – References
- **origin_URI** (*URLField*) – If imported, the original location of the item
- **comments** (*RichTextUploadingField*) – Descriptive comments about the metadata item (8.1.2.2.3.4)
- **submitting_organisation** (*CharField*) – Submitting organisation
- **responsible_organisation** (*CharField*) – Responsible organisation
- **superseded_by_id** (ConceptForeignKey) – Superseded by
- **_concept_ptr_id** (*OneToOneField*) – concept ptr

The correct way to use both of these models would be as shown below:

```
import aristotle_mdr.models import concept, _concept
class ReallyComplexExampleItem(concept):
    relatedTo = models.ManyToManyField(_concept)
```

In this example, the model `ReallyComplexExampleItem` inherits from `concept`, but also includes a many-to-many relationship that links it to any number of registerable concepts, such as Data Element or Objects Classes, additionally because of the inheritance, this would allow links to extended models such as Questions or even self-referential links to other instances of the `ReallyComplexExampleItem` model type.

Retrieving the "true item" when you are returned a `_concept`.

Because `_concept` is not a true abstract class, queries on this table or a Django `QuerySet` that reference a `_concept` won't return the "actual" object but will return an object of type `_concept` instead. There is a `item` property on both the `_concept` and `concept` classes that will return the properly subclassed item using the `get_subclass` method from `django-model-utils`.

`_concept.`**`item`**
: Performs a lookup using `model_utils.managers.InheritanceManager` to find the subclassed item.

`concept.`**`item`**
: Return self, because we already have the correct item.

On the inherited `concept` class this just returns a reference to the original item - `self`. So once the true item is retrieved, this property can be called infinitely without a performance hit.

For example, in code or in a template it is always safe to call an item like so:

```
question.item
question.item.item
question.item.item.item
```

When in doubt about what object you are dealing with, calling `item` will ensure the expected item, and not the `_concept` parent, is used. In the very worst case a single additional query is made and the right item is used, in the best case a very cheap Python property is called and the item is returned straight back.

Setting up search, admin pages and autocompletes for new items types

The easiest way to configure an item for searching and editing within the django-admin app is using the `aristotle_mdr.register.register_concept` method, described in *Using register_concept to connect new concept types*.

Creating admin pages

However, if customisation of Admin pages for an extension is required this can be done through the creation and registration of classes in the `admin.py` file of a Django app.

Because of the intricate permissions around content with the Aristotle Registry, it's recommended that admin pages for new items extend from the `aristotle.admin.ConceptAdmin` class. This helps to ensure that there is a consistent ordering of fields, and information is exposed only to the correct users.

The most important property of the `ConceptAdmin` class is the `fieldsets` property that defines the inclusion and ordering of fields within the admin site. The easiest way to extend this is to add extra options to the end of the `fieldsets` like so:

```
from aristotle_mdr import admin as aristotle_admin

class QuestionAdmin(aristotle_admin.ConceptAdmin):
    fieldsets = aristotle_admin.ConceptAdmin.fieldsets + [
            ('Question Details',
                {'fields': ['questionText','responseLength']}),
            ('Relations',
                {'fields': ['collectedDataElement']}),
    ]
```

It is important to always import `aristotle.admin` **with an alias as shown above**, otherwise there are circular dependancies across various apps when importing which will prevent the app, and thus the whole site, from being used.

Lastly, Aristotle-MDR provides an easy way to give users a suggestion button when entering a name to ensure consistancy within the registry. This can be added to an Admin page by specifying the fields that are used to construct the name - however **these must be fields on the current model**.

For example, if the rules of the registry dictated that a Question name should have the form of its question text along with the name of the collected Data Element, separated by a pipe (`|`), the `QuestionAdmin` class could include the `name_suggest_fields` value of:

```
name_suggest_fields = ['questionText','collectedDataElement']
```

Then to ensure the correct separator is used in `ARISTOTLE_SETTINGS` (which is described in *Configuring the behavior of Aristotle-MDR*) add `"Question"` as a key and `"|"` as its value, like so:

```
ARISTOTLE_SETTINGS = {
    'SEPARATORS': { 'Question':'|',
                    # Other separators not shown
                  },
# Other settings not shown
}
```

For reference, the complete code for the QuestionAdmin class providing extra fieldsets, autcompeletes and suggested names is:

```
from aristotle_mdr import admin as aristotle_admin

class QuestionAdmin(aristotle_admin.ConceptAdmin):
    fieldsets = aristotle_admin.ConceptAdmin.fieldsets + [
            ('Question Details',
                {'fields': ['questionText','responseLength']}),
            ('Relations',
                {'fields': ['collectedDataElement']}),
    ]
    name_suggest_fields = ['questionText','collectedDataElement']
```

For more information on configuring an admin site for Django models, consult the Django documentation.

Making new item types searchable

The creation and registration of haystack search indexes is done in the `search_indexes.py` file of a Django app.

On an Aristotle-MDR powered site, it is possible to restrict search results across a number of criteria including the registration status of an item, its workgroup or Registration Authority or the item type.

In `aristotle.search_indexes` there is the convenience class `conceptIndex` that make indexing a new item within the search engine quite easy, and allows new item types to be searched using these criteria with a minimum of code. Inheriting from this class takes care of nearly all simple cases when searching for new items, like so:

```
from haystack import indexes
from aristotle_mdr.search_indexes import conceptIndex

class QuestionIndex(conceptIndex, indexes.Indexable):
    def get_model(self):
        return models.Question
```

It is important to import the required models from `aristotle.search_indexes` directly, otherwise there are circular dependancies in Haystack when importing. This will prevent the app and the whole site from being used.

The only additional work required is to create a search index template in the `templates` directory of your app with a path similar to this:

```
template/search/indexes/your_app_name/question_text.txt
```

This ensures that when Haystack is indexing the site, some content is available so that items can be queried and weighted accordingly. These templates are passed an `object` variable that is the particualr object being indexed.

Sample content for an index for our question would look like this:

```
{% include "search/indexes/aristotle_mdr/managedobject_text.txt" %}
{{ object.questionText }}
```

Here we include the `managedobject_text.txt` which adds generic content for all concepts into the indexed text, as well as including the `questionText` in the index.

If we wanted to include the content from the related Data Element to add more information for the seach engine to work with we could include this as well, using one of the provided index template in Aristotle, like so:

```
{% include "search/indexes/aristotle_mdr/managedobject_text.txt" %}
{{ object.questionText }}
{% include "search/indexes/aristotle_mdr/dataelement_text.txt" with object=object.
↪collectedDataElement only %}
```

For more information on creating search templates and configuring search options consult the Haystack documentation. For more information on how the search templates are generated read about the Django template engine.

Caveats around extending existing item types

This tutorial has covered how to create new items when inheriting from the base `concept` type. However, Python and Django allow for extension from any object. So if you wished to extend and improve on 11179 item it would be perfectly possible to do so by inheriting from the appropriate class, rather than the abstract `concept`. For example, if you wished to extend a Data Element to create a internationalised DataElement that was only applicable in specific countries, this could be done like so:

```
class Country(model.Models):
    name = models.TextField
    ... # Other attributes could also be applied.

class CountrySpecificDataElement(aristotle.models.DataElement):
    countries = models.ManyToManyField(Country)
```

Aristotle does not prevent you from doing so, however there are a few issues that can arise when extending from non-abstract classes:

- Due to the way that Django handles subclassing, all objects subclassed from a concrete model will also exist in the database as the subclass and an item that belongs to the parent superclass.

  So a `CountrySpecificDataElement` would also be a `DataElement`, so a query like this:

  ```
  aristotle.models.DataElement.objects.all()
  ```

  Would return both `DataElement` s and its subclasses, such as `CountrySpecificDataElement` s, however depending on the domain and objects, this may be desired behaviour.

- Following from the above, restricted searches for only objects of the parent item type will return results from the subclassed item. For example, all searches restricted to a `DataElement` would also return results for `CountrySpecificDataElement`, and they will be displayed in the list as `DataElement` *not* as `CountrySpecificDataElement`.

- Items that inherit from non-abstract classes do not inherit the Django object Managers, this is one of the reasons for the decision to make `concept` an abstract class. As such, it is **strongly adviced** that any new item types that inherit from concrete classes specify the *Aristotle-MDR concept manager*, like so:

  ```
  class CountrySpecificDataElement(aristotle.models.DataElement):
      countries = models.ManyToManyField(Country)
      objects = aristotle_mdr.models.ConceptManager()

  Failure to include this may lead to broken code or pages that expose private␣
  ↪items.
  ```

Creating `unmanagedContent` types

Not all content needs to undergo a standardisation process, and in fact some content should only be accessible to administrators. In Aristotle this is termed an "unmanagedObject". Content types that are unmanaged do not belong to workgroups, and can only be edited by users with the Django "super user" privileges.

It is perfectly safe to extend from the `unmanagedObject` types, however because these are closer to pure Django objects there are much fewer convenience method set up to handle them. By default, `unmanagedContent` is always visible.

Because of their visibility and strict privileges, they are generally suited to relatively static items that may vary between individual sites and add context to other items. Inheriting from this class can be done like so:

```
class Country(aristotle.models.unmanagedObject):
    # Inherits name and description.
    isoCode = models.CharField(maxLength=3)
```

For example, in Aristotle-MDR "Measure" is an `unmanagedObject` type, that is used to give extra context to *UnitOfMeasure* objects.

Including documentation in new content types

To make deploying new content easier, and encourage better documentation, Aristotle reuses help content built into the Django Web framework. When producing dynamic documentation, Aristotle uses the Python docstring of a `concept`-inheriting class and the field level *help_text* to produce documentation.

This can be seen on in the concept editor, administrator pages, item comparator and can be accessed in html pages using the `doc` template tag in the `aristotle_tags` module.

A complete example of an Aristotle Extension

The first content extension for Aristotle that helps clarify a lot of the issues around inheritance is the Comet Indicator Registry. This adds 6 new content types along with admin pages, search indexes and templates and extra content for display on the included Aristotle `DataElement` template - which was all achieved with less than 600 lines of code.

Reusing generic actions to manage relations

class aristotle_mdr.contrib.generic.views.**GenericAlterForeignKey**(***kwargs*)

A view that provides a framework for altering ManyToOne relationships (Include through models from ManyToMany relationships) from one 'base' object to many others.

The URL pattern must pass a kwarg with the name *iid* that is the object from the *model_base* to use as the main link for the many to many relation.

- *model_base* - mandatory - The model with the instance to be altered
- *model_to_add* - mandatory - The model that has instances we will link to the base.
- ***template_name***
  - optional - The template used to display the form.
  - default - "aristotle_mdr/generic/actions/alter_foreign_key.html"
- *model_base_field* - mandatory - the name of the field that goes from the *model_base* to the *model_to_add*.
- *model_to_add_field* - mandatory - the name of the field on the *model_to_add* model that links to the *model_base* model.
- *form_title* - Title for the form

For example: If we have a many to many relationship from *DataElement's to 'Dataset's, to alter the 'DataElement's attached to a 'Dataset*, *Dataset* is the *base_model* and *model_to_add* is *DataElement*.

post(*request*, **args*, ***kwargs*)

Handles POST requests, instantiating a form instance with the passed POST variables and then checked for validity.

class aristotle_mdr.contrib.generic.views.**GenericAlterManyToManyView**(***kwargs*)

A view that provides a framework for altering ManyToMany relationships from one 'base' object to many others.

The URL pattern must pass a kwarg with the name *iid* that is the object from the *model_base* to use as the main link for the many to many relation.

- *model_base* - mandatory - The model with the instance to be altered
- *model_to_add* - mandatory - The model that has instances we will link to the base.
- ***template_name***
  - optional - The template used to display the form.
  - default - "aristotle_mdr/generic/actions/alter_many_to_many.html"
- *model_base_field* - mandatory - the field name that goes from the *model_base* to the *model_to_add*.
- *form_title* - Title for the form

For example: If we have a many to many relationship from *DataElement's to 'Dataset's, to alter the 'DataElement's attached to a 'Dataset*, *Dataset* is the *base_model* and *model_to_add* is *DataElement*.

post(*request*, **args*, ***kwargs*)
Handles POST requests, instantiating a form instance with the passed POST variables and then checked for validity.

class aristotle_mdr.contrib.generic.views.**GenericAlterOneToManyView**(***kwargs*)
A view that provides a framework for altering ManyToOne relationships (Include through models from ManyToMany relationships) from one 'base' object to many others.

The URL pattern must pass a kwarg with the name *iid* that is the object from the *model_base* to use as the main link for the many to many relation.

- *model_base* - mandatory - The model with the instance to be altered
- *model_to_add* - mandatory - The model that has instances we will link to the base.
- ***template_name***
  - optional - The template used to display the form.
  - default - "aristotle_mdr/generic/actions/alter_many_to_many.html"
- *model_base_field* - mandatory - the name of the field that goes from the *model_base* to the *model_to_add*.
- *model_to_add_field* - mandatory - the name of the field on the *model_to_add* model that links to the *model_base* model.
- *ordering_field* - optional - name of the ordering field, if entered this field is hidden and updated using a drag-and-drop library
- *form_add_another_text* - optional - string used for the button to add a new row to the form - defaults to "Add another"
- *form_title* - Title for the form

For example: If we have a many to many relationship from *DataElement's to 'Dataset's, to alter the 'DataElement's attached to a 'Dataset*, *Dataset* is the *base_model* and *model_to_add* is *DataElement*.

post(*request*, **args*, ***kwargs*)
Handles POST requests, instantiating a form instance with the passed POST variables and then checked for validity.

Concept model relations

These are direct reimplementations of Django model relations, at the moment they onyl exist to make permissions-based filtering easier for the GraphQL codebase. However, in future these may add additional functionality such as automatically applying certain permissions to ensure users only retrieve the right objects.

When building models that link to any subclass of `_concept`, use these in place of the Django builtins.

Note: The model these are place on does *not* need to be a subclass of concept. They are for linking *to* a concept subclass.

---

class aristotle_mdr.fields.**ConceptForeignKey**(*to*, *on_delete=None*, *related_name=None*, *related_query_name=None*, *limit_choices_to=None*, *parent_link=False*, *to_field=None*, *db_constraint=True*, ***kwargs*)
Reimplementation of `ForeignKey` for linking a model to a Concept

class aristotle_mdr.fields.**ConceptManyToManyField**(*to*, *related_name=None*, *related_query_name=None*, *limit_choices_to=None*, *symmetrical=None*, *through=None*, *through_fields=None*, *db_constraint=True*, *db_table=None*, *swappable=True*, ***kwargs*)

Reimplementation of `ManyToManyField` for linking a model to a Concept

class aristotle_mdr.fields.**ConceptOneToOneField**(*to*, *on_delete=None*, *to_field=None*, ***kwargs*)

Reimplementation of `OneToOneField` for linking a model to a Concept

Using `register_concept` to connect new concept types

Aristotle-MDR concept register

This module allows developers to easily register new concept models with the core functionality of Aristotle-MDR. The `register_concept` is a wrapper around three methods that registers a new concept with the Django-Admin site, with the Django-Autocomplete and with a class for a Haystack search index. This is all done in a way that conforms to the permissions required for control item visibility.

Other methods in this module can be called, to highly customise how concepts are used within the admin site and search, but should be considered internal methods and future releases of Aristotle-MDR may break code that uses these methods.

aristotle_mdr.register.**register_concept**(*concept_class*, **args*, ***kwargs*)

A handler for third-party apps to make registering extension models based on `aristotle_mdr.models.concept` easier.

Sets up the version controls, search indexes, django administrator page and autocomplete handlers. All `args` and `kwargs` are passed to the called methods. For examples of what can be passed into this method review the other methods in `aristotle_mdr.register`.

Example usage (based on the models in the extensions test suite):

```
register_concept(Question, extra_fieldsets=[('Question','question_text'),]
```

aristotle_mdr.register.**register_concept_admin**(*concept_class*, **args*, ***kwargs*)

Registers the given `concept` with the Django admin backend based on the default `aristotle_mdr.admin.ConceptAdmin`.

Additional parameters are only required if a model has additional fields or references to other models.

Parameters

- **auto_fieldsets** (*boolean*) – If no extra_fieldsets, when set to true this generates a list of fields for the admin page as "Extra fields for [class]"
- **concept_class** (`concept`) – The model that is to be registered
- **extra_fieldsets** (*list*) – Model-specific fieldsets to be displayed. Fields in the tuples given should be those *not* defined by the base `aristotle_mdr.models._concept` class.
- **extra_inlines** (*list*) – Model-specific inline admin forms to be displayed.

aristotle_mdr.register.**register_concept_search_index**(*concept_class*, **args*, ***kwargs*)

Registers the given `concept` with a Haystack search index that conforms to Aristotle permissions. If the concept to be registered does not have a template for serving a search document, a basic document with just the basic fields from `aristotle_mdr.models._concept` will be used when indexing items.

Parameters **concept_class** (`concept`) – The model that is to be registered for searching.

Out-of-the-box features available for new concept types

The modular and object-oriented nature of ISO/IEC 11179 and Python encourage reuse and inheritance when dealing with items. This allows items to have standardised content and behaviours.

Below is a list of features that are available when making new item types based on ISO/IEC 11179 *concepts*.

Content creation wizards that encourage reuse

Every item type is provided with a basic 2-step content creation wizard that shows a user when they may be replicating content that already exists in the registry in an unobtusive way.

This gives freedom to content creators, but gives registry administrators the peace of mind knowing that the system will encourage reuse where possible.

User-friendly modular editor

Descriptive SEO friendly URLs

Basic HTML and downloadable PDF templates

The decoupling of the model management and database back-end and Djangos powerful templating front-end means new item types can be quickly described and prototyped in code, without having to worry about front-end concerns.

Concepts have a generic fallback template that gives a unified look to new items, meaning development can be an iterative process.

Advanced features that require configuration

Admin pages

Search indexing

Using the `ConceptManager` in Django queries

class aristotle_mdr.managers.**ConceptManager**

The `ConceptManager` is the default object manager for `concept` and `_concept` items, and extends from the django-model-utils `InheritanceManager`.

It provides access to the `ConceptQuerySet` to allow for easy permissions-based filtering of ISO 11179 Concept-based items.

class aristotle_mdr.managers.**ConceptQuerySet**(**args*, ***kwargs*)

editable(*user*)
Returns a queryset that returns all items that the given user has permission to edit.

It is **chainable** with other querysets. For example, both of these will work and return the same list:

```
ObjectClass.objects.filter(name__contains="Person").editable()
ObjectClass.objects.editable().filter(name__contains="Person")
```

public()
Returns a list of public items from the queryset.

This is a chainable query set, that filters on items which have the internal *_is_public* flag set to true.

Both of these examples will work and return the same list:

```
ObjectClass.objects.filter(name__contains="Person").public()
ObjectClass.objects.public().filter(name__contains="Person")
```

visible(*user*)
Returns a queryset that returns all items that the given user has permission to view.

It is **chainable** with other querysets. For example, both of these will work and return the same list:

```
ObjectClass.objects.filter(name__contains="Person").visible()
ObjectClass.objects.visible().filter(name__contains="Person")
```

1.4.2 Adding new download formats

While the Aristotle-MDR framework has a PDF download extension, it may be desired to download metadata stored within a registry in a variety of download formats. Rather than include these within the Aristotle-MDR core codebase, additional download formats can be developed included via the download API.

Creating a download module

A download module is a specialised class, that sub-classes `aristotle_mdr.downloader.DownloaderBase` and provides an appropriate `download` or `bulk_download` method.

A download module is just a Django app that includes a specific set of files for generating downloads. The only files required in your app are:

- `__init__.py` - to declare the app as a python module
- `downloader.py` - where your download classes will be stored

Other modules can be written, for example a download module may define models for recording a number of times an item is downloaded.

Writing a `metadata_register`

Your downloader class must contain a register of download types and the metadata concept types which this module provides downloads for. This takes one of the following forms which define which concepts can be downloaded as in the output format:

```
class CSVExample(DownloaderBase):
    download_type = "csv"
    metadata_register = {'aristotle_mdr': ['valuedomain']}
```

```
class XLSExample(DownloaderBase):
    download_type = "xls"
    metadata_register = {'aristotle_mdr': ['__all__']}

class PDFExample(DownloaderBase):
    download_type = "pdf"
    metadata_register = '__template__'

class TXTExample(DownloaderBase):
    download_type = "txt"
    metadata_register = '__all__'
```

Describing these options, these classes specifies the following downloads:

- `csv` provides downloads for Value Domains in the Aristotle-MDR module
- `xls` provides downloads for all metadata types in the Aristotle-MDR module
- `pdf` provides downloads for items in all modules, only if they have a download template
- `txt` provides downloads for all metadata types in all modules

Each download class must also define a class method with the following signature:

```
def download(cls, request, item):
```

This is called from Aristotle-MDR when it catches a download type that has been registered for this module. The arguments are:

- `request` - the request object that was used to call the download view. The current user trying to download the item can be gotten by calling `request.user`.
- `item` - the item to be downloaded, as retrieved from the database.

Note: If a download method is called the user has been verified to have permissions to view the requested item only. Permissions for other items will have to be checked within the download method.

For more information see the `DownloaderBase` class below:

class aristotle_mdr.downloader.**DownloaderBase**

Required class properties:

- description: a description of the downloader type
- download_type: the extension or name of the download to support
- icon_class: the font-awesome class
- metadata_register: can be one of:
  - a dictionary with keys corresponding to django app labels and values as lists of models within that app the downloader supports
  - the string "__all__" indicating the downloader supports all metadata types
  - the string "__template__" indicating the downloader supports any metadata type with a matching download template

classmethod bulk_download(*request*, *item*)

This method must be overriden and return a bulk downloaded set of items as an appropriate django response

classmethod download(*request*, *item*)

This method must be overriden and return the downloadable object as an appropriate django response

How the `download` view works

aristotle_mdr.views.downloads.**download**(*request*, *download_type*, *iid=None*)

By default, `aristotle_mdr.views.download` is called whenever a URL matches the pattern defined in `aristotle_mdr.urls_aristotle`:

```
download/(?P<download_type>[a-zA-Z0-9\-\.]+)/(?P<iid>\d+)/?
```

This is passed into `download` which resolves the item id (`iid`), and determines if a user has permission to view the requested item with that id. If a user is allowed to download this file, `download` iterates through each download type defined in `ARISTOTLE_SETTINGS.DOWNLOADERS`.

A download option tuple takes the following form form:

```
('file_type','display_name','font_awesome_icon_name','module_name'),
```

With `file_type` allowing only ASCII alphanumeric and underscores, `display_name` can be any valid python string, `font_awesome_icon_name` can be any Font Awesome icon and `module_name` is the name of the python module that provides a downloader for this file type.

For example, the Aristotle-PDF with Aristotle-MDR is a PDF downloader which has the download definition tuple:

```
('pdf','PDF','fa-file-pdf-o','aristotle_pdr'),
```

Where a `file_type` multiple is defined multiple times, **the last matching instance in the tuple is used**.

Next, the module that is defined for a `file_type` is dynamically imported using `exec`, and is wrapped in a `try: except` block to catch any exceptions. If the `module_name` does not match the regex `^[a-zA-Z0-9\_]+$` `download` raises an exception.

If the module is able to be imported, `downloader.py` from the given module is imported, this file **MUST** have a `download` function defined which returns a Django `HttpResponse` object of some form.

1.4.3 Adding new bulk actions

Often for user convenience it is useful to perfom the same action across a number of similar metadata items. Aristotle-MDR provides a bulk action API that allows developers to create new discoverable action types that are shown to users in certain item lists, such as search results or workgroup item listings.

Registering a bulk action

The `BULK_ACTIONS` *setting* in the in the `ARISTOTLE_SETTINGS` dictionary stores the register of bulk actions used for generating lists of actions. Adding the qualified path to the form is sufficient to register a new bulk action. For example this set in `ARISTOTLE_SETTINGS` would register an action int Python module `module.forms.MyBulkAction`:

```
'BULK_ACTIONS': [
    'module.forms.MyBulkActionForm',
]
```

Writing a functional bulk action

A bulk action form is just a specialised Django form for acting on multiple Aristotle-MDR concepts, with a few small additions that come from inheriting from `aristotle_mdr.forms.bulk_actions.BulkActionForm`.

After inheriting to make a form function some properties should exist.

- `action_text` - This is the name for an action shown in lists to users. *Default* is based on the class name.
- `classes` - A string of HTML classes that will be applied to each item. *Default* empty. Currently these are used for inserting 'Font Awesome' icons for each action.
- `confirm_page` - An optional template name used to render between a user clicking the action and completing it. By adding extra fields to a form, with this template a bulk action can get additional inforamtion from a user before continuing. *No default*, if this is empty no confirmation is requested.
- `items_label` - An optional override of the label for the list of items the action form acts on. Defaults to "Select some items"

There are two additional methods that complete the class:

- `can_use` - A `classmethod` that provides a boolean response indicating if a certain user has permission to use this action in any context - note this permission does not have knowledge of the items selected. *Default* is true, so if this is not overriden all users will see the action in their list.
- `make_changes` - Performs that actual action of the form, this is called once the user invokes a bulk action (after confirmation is required). *No default*, not including a `make_changes` method will cause your action to fail. Any text returned from this method will be shown to a user via the django messages framework.

An example bulk action form

Below is an example bulk action that is only visible for staff users, and deletes the items requested by a user.:

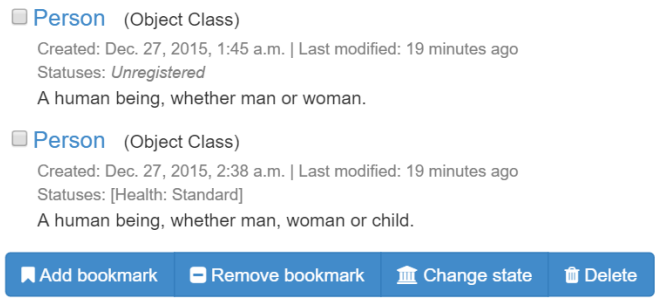
Listing 1.3: mymodule.forms.StaffDeleteActionForm

```
from django import forms
from aristotle_mdr.forms.bulk_actions import BulkActionForm
from django.utils.translation import ugettext_lazy as _


class StaffDeleteActionForm(BulkActionForm):
    action_text = _('Delete')
    classes="fa-trash"
    confirm_page = "confirm_delete.html"
    items_label="Items to delete",

    safe_to_delete = forms.BooleanField(required=True, label="Tick to confirm deletion
↪")

    @classmethod
    def can_use(cls, user):
        return user.is_staff

    def make_changes(self):
        if not self.user.is_staff:
            raise PermissionDenied
        else:
            self.cleaned_data['items'].delete()
        return "Items deleted"
```

Listing 1.4: confirm_delete.html

```
{% extends "aristotle_mdr/base.html" %}

{% block title %}Delete items{% endblock %}
{% block content %}
{{ form.media }}
<form method="post" action="{% url 'aristotle:bulk_action' %}?next={{next}}">{% csrf_
↪token %}
    <p>
        Use this page to confirm you wish to delete the following items.
    </p>
    <input type="hidden" name="bulkaction" value="{{action}}"/>
    <table>
        {{ form.as_table }}
    </table>
    <div>
        <a class="btn btn-default" href="{{ next }}">Cancel</a>
        <button type="submit" name="confirmed" class="btn btn-primary" value="Delete">
↪Delete</button>
    </div>
</form>
{% include 'autocomplete_light/static.html' %}
{% endblock %}
```

This will produce a button wherever other bulk actions are available, similar to the 'Delete' button available on the right in the image below.



1.4.4 Including extra content on pages

Often when adding new content types there will be references to pre-existing items which may need to be shown on an existing page. While it's possible to completely override a template it can be easier to simply declare any "extra content" in a special template that Aristotle will then include when rendering the main template. This can be very advantageous when an extension is designed to be included on a site alongside other extensions.

To do this, all that needs to be done is to include a directory path `extra_content` under the template path for your extension. If an entry for the extension has been included in the Aristotle settings, the main will then look in this directory for templates with the same name as the currently rendered content type.

For example, the comet extension defines an `Indicator` content type that has multiple references to `DataElement` items. Rather than redefine the `aristotle/concept/dataElement.html` template, it defines an `extra_content` template.

Templates for comet content types are included under `templates/comet/`. Under this directory there is a further path `extra_content`, so by creating a file named `dataElement.html` we can create a template for including

extra relationship details.

This will then be rendered in the "Related content" section of the page.

The full path for a generic extra content template, would be:

```
templates/{app_name}/extra_content/{model_name}.html
```

1.4.5 Tags and filters available in aristotle templates

A number of convenience tags are available for performing common actions in custom templates.

Include the aristotle template tags in every template that uses them, like so:

```
{% load aristotle_tags %}
```

Available tags and filters

aristotle_mdr.templatetags.aristotle_tags.**adminEdit**(*item*)

A tag for easily generating the link to an admin page for editing an item. For example:

```
<a href="{% adminEdit item %}">Advanced editor for {{item.name}}</a>
```

aristotle_mdr.templatetags.aristotle_tags.**can_edit**(*item*, *user*)

A filter that acts as a wrapper around aristotle_mdr.perms.user_can_edit. Returns true if the user has permission to edit the item, otherwise it returns False. If calling user_can_edit throws an exception it safely returns False.

For example:

```
{% if myItem|can_edit:request.user %}
  {{ item }}
{% endif %}
```

aristotle_mdr.templatetags.aristotle_tags.**can_view**(*item*, *user*)

A filter that acts as a wrapper around aristotle_mdr.perms.user_can_view. Returns true if the user has permission to view the item, otherwise it returns False. If calling user_can_view throws an exception it safely returns False.

For example:

```
{% if myItem|can_view:request.user %}
  {{ item }}
{% endif %}
```

aristotle_mdr.templatetags.aristotle_tags.**can_view_iter**(*qs*, *user*)

A filter that is a simple wrapper that applies the aristotle_mdr.models.ConceptManager.visible(user) for use in templates. Filtering on a Django Queryset and passing in the current user as the argument returns a list (not a Queryset at this stage) of only the items from the Queryset the user can view.

If calling can_view_iter throws an exception it safely returns an empty list.

For example:

```
{% for item in myItems|can_view_iter:request.user %}
  {{ item }}
{% endfor %}
```

aristotle_mdr.templatetags.aristotle_tags.**doc**(*item*, *field=None*)
Gets the appropriate help text or docstring for a model or field. Accepts 1 or 2 string arguments: If 1, returns the docstring for the given model in the specified app. If 2, returns the help_text for the field on the given model in the specified app.

aristotle_mdr.templatetags.aristotle_tags.**downloadMenu**(*item*)
Returns the complete download menu for a partcular item. It accepts the id of the item to make a download menu for, and the id must be of an item that can be downloaded, otherwise the links will show, but not work.

For example:

```
{% downloadMenu item %}
```

aristotle_mdr.templatetags.aristotle_tags.**in_workgroup**(*user*, *workgroup*)
A filter that acts as a wrapper around `aristotle_mdr.perms.user_in_workgroup`. Returns true if the user has permission to administer the workgroup, otherwise it returns False. If calling `user_in_workgroup` throws an exception it safely returns False.

For example:

```
{% if request.user|in_workgroup:workgroup %}
  {{ something }}
{% endif %}
```

aristotle_mdr.templatetags.aristotle_tags.**public_standards**(*regAuth*, *itemType='aristotle_mdr._concept'*)
This is a filter that accepts a registration Authority and an item type and returns a list of tuples that contain all *public* items with a status of "Standard" or "Preferred Standard" *in that Registration Authority only*, as well as a the status object for that Authority.

The item type should consist of the name of the app the item is from and the name of the item itself separated by a period (`.`).

This requires the django `django.contrib.contenttypes` app is installed.

If calling `public_standards` throws an exception or the item type requested is not found it safely returns an empty list.

For example:

```
{% for item, status in registrationAuthority|public_standards:'aristotle_mdr.
↪DataElement' %}
  {{ item }} - made standard on {{ status.registrationDate }}.
{% endfor %}
```

aristotle_mdr.templatetags.aristotle_tags.**stateToText**(*state*)
This tag takes the integer value of a state for a registration status and converts it to its text equivilent.

aristotle_mdr.templatetags.aristotle_tags.**ternary**(*condition*, *a*, *b*)
A simple ternary tag - it beats verbose if/else tags in templates for simple strings If the `condition` is 'truthy' return `a` otherwise return `b`. For example:

```
<a class="{% ternary item.is_public 'public' 'private' %}">{{item.name}}</a>
```

aristotle_mdr.templatetags.aristotle_tags.**zws**(*string*)
zws or "zero width space" is used to insert a soft break near em-dashed. Since em-dashs are commonly used in Data Element Concept names, this helps them wrap in the right places.

For example:

```
<h1>{% zws item.name %}</h1>
```

1.4.6 Using Aristotle permissions in custom code

One of the key features in Aristotle is specific access control to items based on a rich matrix of user groups. To make creating extension easier these are exposed through the code in a number of easy to use ways.

Permissions in `perms.py`

aristotle_mdr.perms.**user_can_change_status**(*user*, *item*)
Can the user change the status of the item?

aristotle_mdr.perms.**user_can_edit**(*user*, *item*)
Can the user edit the item?

aristotle_mdr.perms.**user_can_view**(*user*, *item*)
Can the user view the item?

Permissions-based `ConceptManager`

All correctly derived `concept` items should have their default manager set to the `aristotle.models.ConceptManager`. For more information on how this works see the full documentation on the *ConceptManager and ConceptQuerySet*.

class aristotle_mdr.models.**ConceptManager**
The `ConceptManager` is the default object manager for `concept` and `_concept` items, and extends from the django-model-utils `InheritanceManager`.

It provides access to the `ConceptQuerySet` to allow for easy permissions-based filtering of ISO 11179 Concept-based items.

Permissions template tags

Tags and filters available in aristotle templates

A number of convenience tags are available for performing common actions in custom templates.

Include the aristotle template tags in every template that uses them, like so:

```
{% load aristotle_tags %}
```

Available tags and filters

aristotle_mdr.templatetags.aristotle_tags.**can_edit**(*item*, *user*)
A filter that acts as a wrapper around `aristotle_mdr.perms.user_can_edit`. Returns true if the user

has permission to edit the item, otherwise it returns False. If calling `user_can_edit` throws an exception it safely returns False.

For example:

```
{% if myItem|can_edit:request.user %}
  {{ item }}
{% endif %}
```

`aristotle_mdr.templatetags.aristotle_tags.`**`can_view`**(*item*, *user*)

A filter that acts as a wrapper around `aristotle_mdr.perms.user_can_view`. Returns true if the user has permission to view the item, otherwise it returns False. If calling `user_can_view` throws an exception it safely returns False.

For example:

```
{% if myItem|can_view:request.user %}
  {{ item }}
{% endif %}
```

`aristotle_mdr.templatetags.aristotle_tags.`**`can_view_iter`**(*qs*, *user*)

A filter that is a simple wrapper that applies the `aristotle_mdr.models.ConceptManager.visible(user)` for use in templates. Filtering on a Django `Queryset` and passing in the current user as the argument returns a list (not a `Queryset` at this stage) of only the items from the `Queryset` the user can view.

If calling `can_view_iter` throws an exception it safely returns an empty list.

For example:

```
{% for item in myItems|can_view_iter:request.user %}
  {{ item }}
{% endfor %}
```

There are more *template tags available in Aristotle*

1.4.7 Testing

Aristotle uses tox and django's unit test framework for testing

Running tests locally

Docker

For running tests on a local docker environemnt refer to https://github.com/aristotle-mdr/aristotle-metadata-registry/docker

Tox

Tests can be run locally running tox with an optional environment argument e.g. `tox -e dj1.11-test-linux-db-sqlite-search-whoosh`.

Virtualenv

- To run tests in a virtualenv, first set the `DJANGO_SETTINGS_MODULE` enviroment variable to the settings module you want to use
- Install dev requirements with `pipenv install --dev`
- Run tests with `pipenv run django-admin test aristotle_mdr`. Replacing aristotle_mdr with a full test path if needed

Adding extension modules to our automated testing

When adding an extension package to the system it is important to integrate this with the automated testing process to ensure it is tested alongside the rest of the system

Once the extension has been added to the `/python` directory follow these steps to enable automated testing

1. Add a `setup.py` for your package with dependancies defined in install_requires
2. Add the package to the Pipfile at the base directory of the repo
3. Run `pipenv lock` to update the lock file
4. Add a new model extension to the envlist in `tox.ini` at the base directory of the repo
5. Define your settings module, module name and module path in the setenv section of `tox.ini`
6. Add a new stage in `.travis.yml` with your new module extension

Done, your module will now be tested by travis automatically using the command `django-admin test modulename`

1.5 Customing the Aristotle Metadata Registry

1.5.1 Customising templates

Aristotle-MDR builds pages using Django tempaltes.

Which means almost every part of the website can be customised using the django template overriding order.

But default, when building pages Aristotle will try to load templates from the `site` directory first before using templates from Aristolte and extensions or other Django apps.

1.5.2 Customising the browse pages

Making metadata-specific browse lists

To make a metadata specific browse page, add a directory and template into your custom templates directory for that specific app of the form:

```
'aristotle_mdr_browse/<app_label>/<model_name>_list.html
```

1.6 Creating and deploying user help

1.6.1 Aristotle Help models

class aristotle_mdr.contrib.help.models.**ConceptHelp**(**args*, ***kwargs*)

A Concept help page documents a given model that inherits from an 11179 concept.

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **slug** (*AutoSlugField*) – Slug
- **app_label** (*CharField*) – Add an app for app specific help, required for concept help
- **title** (*TextField*) – A short title for the help page
- **body** (*RichTextUploadingField*) – A long help definition for an object or topic
- **language** (*CharField*) – Language
- **is_public** (*BooleanField*) – Indicates if a help topic is available to non-registered users.
- **helpbase_ptr_id** (*OneToOneField*) – Helpbase ptr
- **concept_type** (*CharField*) – Concept type
- **brief** (*TextField*) – A short description of the concept
- **offical_definition** (*TextField*) – An official description of the concept, e.g. the ISO/IEC definition for an Object Class
- **official_reference** (*TextField*) – The reference document that describes this concept type
- **official_link** (*TextField*) – An link to an official source for a description of the concept
- **creation_tip** (*RichTextUploadingField*) – Instructions for creating good content of this type

class aristotle_mdr.contrib.help.models.**HelpBase**(**args*, ***kwargs*)

The base help class for Aristotle help pages.

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **slug** (*AutoSlugField*) – Slug
- **app_label** (*CharField*) – Add an app for app specific help, required for concept help
- **title** (*TextField*) – A short title for the help page
- **body** (*RichTextUploadingField*) – A long help definition for an object or topic
- **language** (*CharField*) – Language

- **is_public** (*BooleanField*) – Indicates if a help topic is available to non-registered users.

class aristotle_mdr.contrib.help.models.**HelpPage**(**args*, ***kwargs*)

A help page is a generic way of providing help to a user on a topic.

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **slug** (*AutoSlugField*) – Slug
- **app_label** (*CharField*) – Add an app for app specific help, required for concept help
- **title** (*TextField*) – A short title for the help page
- **body** (*RichTextUploadingField*) – A long help definition for an object or topic
- **language** (*CharField*) – Language
- **is_public** (*BooleanField*) – Indicates if a help topic is available to non-registered users.
- **helpbase_ptr_id** (*OneToOneField*) – Helpbase ptr

1.6.2 Special syntax in user help files

As help files are just django fixture files all of the caveats there apply, with a few small conventions applied on top.

- For consistancy and readability, its encouraged to keep one help fixture per file.
- The body of the help file can be HTML, and this will be displayed to the user unchanged. It is up to documenters to ensure that help files are valid HTML that won't break layout.

A few additional

Below is an example help file:

```
- model: aristotle_mdr_help.helppage
  fields:
    title: Advanced Search
    language: en
    body: >
      <h2>Restricting search with the advanced search options</h2>
      <p>
          The <a href="/search/">search page</a> provides a form that gives
          users control to filter and sort search results.</p>
      <p>
      <img src="{static}/aristotle_mdr/search_filters.png">
          When searching, the "indexed text" refers to everything crawled by the␣
↪search engine.
```

1.6.3 Writing help files

To improve users abilities to self-help and self-manage within an instance the Aristotle Metadata Registry includes a help API that allows system administrators, and extension and download developers to write help files that are searchable by users.

At their core, these help files are similar to django fixture files with a few relatively minor differences.

- The subclassing of help files needed for indexing can be ignored
- One fixture per file is recommended to make writing easier, although multiple help pages can be parsed from one file

1.6.4 Importing help files

The Aristotle-MDR provides a django command line action similar to the `loadata` called `load_aristotle_help`. This adds an additional switch `--update` or `-U` that when attempting to insert, will instead override help files.

For example:

```
./manage.py load_aristotle_help
```

Will load all help files in the `./aristotle_help_files/` subdirectory of *all apps in ‘‘INSTALLED_APPS‘‘*.

1.6.5 Accessing help in extension and download templates

Aristotle provides a template tag to extract a number of different help types for 11179-derived concepts in templates.

This can be called using `help_doc` and passing the model class for the concept required along with the help field requested.

{% load aristotle_help %} {% help_doc model_class ‘brief’ %}

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

G

H

I

P

R

S

T

U

V

Z