DataMYNE Documentation

Release 1.0.0

The New School

May 16, 2011

CONTENTS

Contents:

CHAPTER

ONE

PROFILES: MODELS

class `apps.profiles.models.`**`ActivePersonManager`**

Bases: `django.db.models.manager.Manager`

The `ActivePersonManager` supports the `Person` class by restricting queries to only those people who have active user accounts.

class `apps.profiles.models.`**`AreaOfStudy`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

An `AreaOfStudy` is a more amorphous grouping of courses that are not necessarily organized under the traditional hierarchy.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`BaseModel`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`

BaseModel is the root of almost all other objects within the DataMYNE system. It establishes the creation and modification date fields, as well as indicating which user last changed the given model. It also stubs out the permissions for a `auth.User` and other objects based upon their associations within the university's hierarchy of "units" (e.g. `profiles.Division`, `profiles.School`, etc.)

`get_unit``()`

get_unit returns the organizational unit that hold responsibility for this object. For example, a committee executing get_unit could return the division that has authority over it.

`has_unit_permission``(`*user*`)`

Given a user, this method will check to see if that user is attached to the appropriate organizational unit to have access to edit this object. This allows people in higher levels of the university's hierarchy the ability to edit a greater number of objects, while restricting those lower down to objects that only affect their unit.

`save``(`**args*, ***kwargs*`)`

Every DataMyne object has a `created_by` field that contains the `User` object of the user who was logged in when the system created the object. The method uses the datamining.middleware module to access another thread and check on the current user.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`ContactEmail`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

`ContactEmail` is legacy code that needs to be refactored out``

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Course`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `Course` is an object that contains course information across time. A `Section` object is connected to a course and shows only the information that is specific to a particular semester and set of faculty.

The distinction between `Course` and `Section` is subtle but crucial. A course effectively lives outside of time. Faculty are never associated with a course. Nor are semesters. A section, by contrast, has associated with it a particular `Semester` as well as zero or many `FacultyMember` objects (it is possible to have zero faculty for a `Section` in cases where the faculty assignments are TBD.)

It is Mike Edwards's **strong** recommendation that the `taken` field by replaced with code that defines an `Affiliation`. See the documentation of the `reporting` app for a complete explanation of the design decision to favor `Affiliation` objects over simple ManyToManyField fields.

Also, the `projects` field should be refactored and removed, since the `Project` model is deprecated. It should be replaced with code that relates to `Work` objects instead, either as a ManyToMany or something similar to what the `Affiliation` object seeks to achieve between `Person` objects and every other DataMYNE class.

`get_unit``()`

A `Course` is assumed to be under the authority of whatever organizational unit manages the course's `Subject` object.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`CourseImage`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `CourseImage` connects a `Course` to an image file and associated metadata.

It is Mike Edwards's **strong** recommendation that this be refactored and migrated into a generic `Image` class that can relate the same data to a GenericForeignKey. This would allow images to decorate any other object within the DataMYNE system, allowing for a common set of code to handle image assets, rather than duplicating the effort. In addition, it may also be worth considering how a generic `Image` object could inherit from a generic `Media` object, leaving open the possibility of achieving similar efficiency for video, audio, and other unforeseen media.

Of course, some of this addresses issues at the heart of what DataMYNE is and what it ought to be. The degree to which this system stores (or references) other data depends largely on how much data needs to be internally understood by the system (for the purposes of searching, cross-referencing, etc.) and how much ought to be offloaded to the rest of the Web (e.g. Flickr, YouTube, etc.) At the time of writing, this issue is still very much in flux. As such, designing for flexibility (instead of performance or simplicity) is paramount.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Department`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `Department` is one of the the highest "units" within the university. Below it are programs. It sits at the same level as a `Department`

Historically, all divisions except Parsons the New School for Design have Departments.

`authorities`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Division`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `Division` is the highest "unit" within the university. Below it are `Department` , `School`, etc.

`authorities`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Expertise`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

An `Expertise` (sometimes called an area of expertise) is a canonical keyword that users can select to describe them in their profiles.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`FacultyMember`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.Person`

A `FacultyMember` is a `Person` (believe it or not) that teaches one or more `Section` objects of a `Course` object. Their association with a `Section` is their key distinction with other `Person` objects.

class `apps.profiles.models.`**`Invitation`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`

An `Invitation` object allows a `host` user to invite a `guest` user to be part of another object within the DataMYNE system. The guest can either be a registered user or someone with an email address outside of the university.

Once a guest receives his or her invitation email, they are directed to follow a url (composed of a unique `slug`) that will do one of the following:

- if the user is a member of DataMYNE and is signed in, he or she will be taken directly to the object in question
- if the user is a member of DataMYNE and is not signed it, he or she will be taken first to a login screen
- if the user is not yet a member of DataMYNE, he or she will still be directed to the login screen.
  - If he or she is able to join (e.g. has a listing within LDAP), a profile will automatically be created.
  - If he or shee is not able to join, at this point, tough luck.

This last case is an interesting one, since DataMYNE still represents a closed community. The best way to deal with this is most likely best handled within the `profiles.backends` module. This will be most important for alumni who no longer can authenticate through LDAP, as well as incoming students who are not yet established within LDAP. In any case, the `Invitation` model should remain more or less agnostic to this.

`content_object`
Provides a generic relation to any object through content-type/object-id fields.

class `apps.profiles.models.`**`Link`**(**args*, ***kwargs*)
Bases: `apps.profiles.models.BaseModel`

A `Link` allows for adding a URL, plus description and other metadata, to any object in the system.

It should succeed `WorkURL` for `Person` links and be used for other DataMyne objects should links become useful with them.

`content_object`
Provides a generic relation to any object through content-type/object-id fields.

`unit_permissions`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Organization`**(**args*, ***kwargs*)
Bases: `apps.profiles.models.BaseModel`

An `Organization` is an object that represents any number of kinds of groups that may appear within the university. It comprises everything from officially designated labs to ad hoc student groups.

The `projects` field should be refactored out and, most likely, replaced with some kind of relationship to the `Work` model (either via a new ManyToMany field or something akin to how the `Person` models relates to other objects via the `Affiliation`

`affiliations`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`has_unit_permission`(*user*)
Organization models don't have a parent unit, therefore there are no unit restrictions on this. This situation may change if the `Sponsorship` object becomes more widely used.

meetings

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

unit_permissions

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`OrganizationType`**(**args*, ***kwargs*)

Bases: `apps.profiles.models.BaseModel`

An `OrganizationType` defines a canonical category for `Organization` objects (e.g. lab, center, institute, etc.)

unit_permissions

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Person`**(**args*, ***kwargs*)

Bases: `apps.profiles.models.BaseModel`

A `Person` represents any number of roles within the university, but provides a unifying class for dealing with all of them. The `Person` has fields for first and last name, N Number, etc. as well as several crucial class and object methods that pertain to all members of the university.

A `Student`, `Staff`, and `FacultyMember` object are all of the `Person` type.

activate(*email*)

An email-based activation method. This is deprecated since the introduction of the LDAP authentication

cv_text

Docstring

deactivate()

Deactivates a person's user account

group_perms_set

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

unit_permissions

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

user_perms_set

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Program`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `Program` is typically the lowest organizational unit in the university. It sits below either a `School` or a `Department`.

`affiliations`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`authorities`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Project`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

Project is a now defunct way of expressing what we now use Organizations and Works to accomplish. This should be retired.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Requirement`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `Requirement` connects a `Program` to zero or more `Course` objects in order to define a program's requirements.

Although this relationship exists in the model code, there are currently no tools to manage this outside of the admin tool. Considerable thought and planning needs to go into how to manage this, as well as other models such as `AreaOfStudy`, etc.

`unit_permissions`

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`School`**`(`**args*, ***kwargs*`)`

Bases: `apps.profiles.models.BaseModel`

A `School` is one of the the highest "units" within the university. Below it are programs. It sits at the same level as a `Department`

Historically, only Parsons the New School for Design has Schools.

`authorities`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`unit_permissions`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Section`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.BaseModel`

A `Section` object is connected to a course and shows only the information that is specific to a particular semester and set of faculty. A `Course` is an object that contains course information across time.

Refer to the `Course` documentation for a more complete explanation of this distinction.

`get_display_title``()`
Not all sections need to have their own title, but there are enough cases where the section of a course has a meaningfully different title that this ought to be shown instead (e.g. Parsons AMT Collab studios)

`unit_permissions`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Semester`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.BaseModel`

A `Semester` object represents the season and year in which a `Section` of a `Course` is held. Note that, in Banner exports, spring and summer semesters are list as part of the previous year. For example:

- 201010 is the fall semester of 2010
- 201030 is the spring semester of 2011

For simplicity's sake, however, we convert the banner code to the correct calendar year. Therefore, fall 2010 has the term "fa" and the year "2010" while spring 2011 has the term "sp" and the year "2011". All the conversions for this are done in the import script and should never need to be considered outside of that.

`unit_permissions`
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Sponsorship`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.BaseModel`

A `Sponsorship` ties an organization to a bureaucratic unit, etc.

At the time of writing, this relationship exists only in the model code. The necessity of an organization's sponsorship will need to be determined as the `Organization` use cases and code achieve more maturity.

See the `Authority` class within the `reporting` documentation for an analog to this between `Committee` objects and organizational units.

content_object
Provides a generic relation to any object through content-type/object-id fields.

unit_permissions
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`Staff`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.Person`

A `Staff` member is a `Person` that works for the university in a role other than faculty. They are assigned to a `Division` and have a job description.

class `apps.profiles.models.`**`Student`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.Person`

A `Student` is a `Person` that has a graduation year and home program.

Students need to be addressed with the utmost sensitivity and security, since we must follow FERPA guidelines and cannot divulge any more data than is absolutely necessary (i.e. directory information). See:

http://www2.ed.gov/policy/gen/guid/fpco/ferpa/mndirectoryinfo.html

class `apps.profiles.models.`**`Subject`**`(`**args*, ***kwargs*`)`
Bases: `apps.profiles.models.BaseModel`

A `Subject` typically represents a group of `Course` objects below a `Program`. Recent changes in the way courses are constructed, though, may require this simple relationship to be expanded somewhat. A `Subject` also contains the four-letter abbreviation that precedes a couse number in the course directories.

unit_permissions
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.profiles.models.`**`UnitPermission`**`(`**args*, ***kwargs*`)`
Bases: `django.db.models.base.Model`

A `UnitPermission` object attaches itself to a user and another DataMYNE object, most suitably an organizational unit (e.g. `Division`, `Program`).

Unit permissions allow for an expanded authorization framework that goes beyond the stock permissions built into Django which are based solely on `ContentType`. For example, a user may:

- have permission to change only `Course` objects
- have permission to change only objects that falls under the Parsons `Division`

The effect of this is to create a matrix that constrains administrators to only edit certain kind of objects and only object that fall under their purview.

At the time of writing, this can be managed from within the admin tool by DataMYNE administrators. The administrator need to go to a unit's admin page (e.g. Parsons) and, under the Unit Permissions section, add only those users who are able to act on behalf of the unit (e.g. an operations manager for Parsons, a program director of Communication Design, etc.)

While this addresses the problem of how to restrict editorial permissions to both object (via Django's system) and unit, more work needs to be done on creating an effective set of admin interfaces that could allow the assignment of unit permissions by people within the university, rather than the rather rough way DataMYNE administrators need to work now within the admin tool. This will become especially important as the system expands to cover the entire university.

content_object
Provides a generic relation to any object through content-type/object-id fields.

class apps.profiles.models.**Work**(**args*, ***kwargs*)
Bases: apps.profiles.models.BaseModel

A Work represents any work of art or any other product/document/etc. created within the system. We do NOT associate these with a Person directly. Instead, these relationships are managed with the Affiliation class from the reporting application. The allows for many-to-many relationships, as well as providing subtle variations in the role, date of participation, etc.

affiliations
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

unit_permissions
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class apps.profiles.models.**WorkURL**(**args*, ***kwargs*)
Bases: apps.profiles.models.BaseModel

A WorkURL provides a URL associated with a persons work (e.g. portfolio links, etc.).

This should be deprecated in favor of the more flexible Link class.

unit_permissions
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

CHAPTER

TWO

PROFILES: VIEWS

`apps.profiles.views.`**`accept_invitation`**`(`*request*, **args*, ***kwargs*`)`

If the user if logged in, this view simply redirects that user to the appropriate view and marks the invite as accepted.

This view requires that an object that can have an accepted invitation possess an `accept_invitiation` method.

`apps.profiles.views.`**`accept_organization_invitation`**`(`*request*, **args*, ***kwargs*`)`

This view marks an invitation to an organization as accepted.

Note that this should be deprecated in favor of the more general `accept_organization` view, but requires that the `Organization` model have an `accept_organization` model built into it.

`apps.profiles.views.`**`activate`**`(`*request*, **args*, ***kwargs*`)`

This is a deprecated method that preceded the move to LDAP authentication.

`apps.profiles.views.`**`add_syllabus`**`(`*request*, **args*, ***kwargs*`)`

This view allows specific users the ability to upload a syllabus and attach it to the `Section` object.

`apps.profiles.views.`**`admin`**`(`*request*, **args*, ***kwargs*`)`

Deprecated. A new and more complete admin section needs to be created to handle the move from DataMYNE as a Parsons-based experiment to university- wide infrastrucure.

`apps.profiles.views.`**`api`**`(`*request*`)`

Deprecated

`apps.profiles.views.`**`browse`**`(`*request*`)`

This is a deprecated view. It served primarily as the *browse* page, which has since been temporarily removed from the main nav pending a redesign.

`apps.profiles.views.`**`contact`**`(`*request*`)`

A simple contact-form processor.

`apps.profiles.views.`**`contact_student`**`(`*request*, *person_id*`)`

This view allows users to send students email without revealing the student's email address publicly.

There most likely should be a setting on the `Student` profile page which allows the student to disable this. At time of writing, this does not exist in either model or view code and should be considered.

`apps.profiles.views.`**`decline_invitation`**`(`*request*, **args*, ***kwargs*`)`

If the user if logged in, this view simply redirects that user back to his or her previous view and deletes the invitation. The `Invitation` is one of the very few objects that get deleted in the DataMYNE system. In the future, it may prove useful to only "logically" delete these, but still retain the connection in order to mine more data on social dynamics.

`apps.profiles.views.`**`delete_work`**(*request*, *work_id*, *person_id*)

This view deletes a `Work` object. It is one of the few models that we allow to be publicly deleted. Since this is user-contributed content, however, it only seems fair that we respect their wishes about the display and use of their own work.

`apps.profiles.views.`**`download_syllabus`**(*request*, *section_id*)

This view generates a document based on the uploaded syllabus. It tries to create a name for the file that matches the university's guidelines for naming syllabus documents rather than defaulting to the user's file naming scheme.

`apps.profiles.views.`**`edit_course_profile`**(*request*, **args*, ***kwargs*)

This view allows for the editing of courses on the public site. Like the `Person` profile views, it has both admin and non-admin versions, so be careful with that distinction. It also uses the unit permissions and Django permissions code. While simpler here than in other parts of the site (cf. `edit_organization`), the ability to edit is probably best determined with a object method written into the model and not in the view code, as this makes the permissions system overly brittle.

`apps.profiles.views.`**`edit_organization`**(*request*, **args*, ***kwargs*)

This view edits an `Organization` object. See the model documentation for a more complete description of what an organization represents.

Along with the `Committee` model, organizations represent one of the most complicated objects in terms of security. This model should be refactored first as part of a general clean up to remove edittable from the view code and port it into the model code itself.

Also note the use of the `current` and `past` managers for the member and leader affiliations. This is one of the benefits of using the `Affiliation` object over a `ManyToManyField`: we can retain historical information even after the connection is no longer active. For example, we can know all of the previous leaders of an organization while still allowing the current ones to be the only recipients of security clearance, public display, etc.

In addition, the `Affiliation` managers have the `begin` and `retire` methods that allow connections to see easily set to current or past without having to rewrite complicated code. "Retirement" is the preferred way for disposing of a current affiliation. It has the same effect as a deletion, while still retaining the connections for historical and data-mining purposes.

`apps.profiles.views.`**`edit_person_profile`**(*request*, *person_id*)

This view serves to unify the `FacultyMember`, `Student`, and `Staff` profile edit views under a single edit view. Within the `urls` module, any person's role-specific profile editing page can be reached from here. This allows a common "gateway" to profiles editors without needing to decide, within the templates, which role a `Person` occupies.

This should be the destination for all profile edit requests going forward.

`apps.profiles.views.`**`edit_profile`**(*request*, **args*, ***kwargs*)

This is the profile edit view for `FacultyMember` objects. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `edit_person_profile` view to reduce repetition.
- edittable defines whether a user can edit this object. This really needs to be refactored out of the view code and into the model code, either at the level of `FacultyMember` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- There are really two edit views: one for regular users and one for admins. Be sensitive to the distinction, since admins have access to many fields that we do not want users to be able to update (e.g. field provided by Banner, etc.)

`apps.profiles.views.`**`edit_section_profile`**(*request*, **args*, ***kwargs*)

This view allows for the editing of sections on the public site. Like the `Person` profile views, it has both admin and non-admin versions, so be careful with that distinction. It also uses the unit permissions and Django permissions code. While simpler here than in other parts of the site (cf. `edit_organization`), the ability to edit is probably best determined with a object method written into the model and not in the view code, as this makes the permissions system overly brittle.

One important consideration for the editting permissions is that we allow assigned faculty to edit this page, as well as the usual administrators.

`apps.profiles.views.`**`edit_staff_profile`**(*request*, **args*, ***kwargs*)

This is the profile edit view for `Staff` objects. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `edit_person_profile` view to reduce repetition.
- edittable defines whether a user can edit this object. This really needs to be refactored out of the view code and into the model code, either at the level of `Staff` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- There are really two edit views: one for regular users and one for admins. Be sensitive to the distinction, since admins have access to many fields that we do not want users to be able to update (e.g. field provided by Banner, etc.)

`apps.profiles.views.`**`edit_student_profile`**(*request*, **args*, ***kwargs*)

This is the profile edit view for `Student` objects. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `edit_person_profile` view to reduce repetition.
- edittable defines whether a user can edit this object. This really needs to be refactored out of the view code and into the model code, either at the level of `Student` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- There are really two edit views: one for regular users and one for admins. Be sensitive to the distinction, since admins have access to many fields that we do not want users to be able to update (e.g. field provided by Banner, etc.)

`apps.profiles.views.`**`edit_work`**(*request*, **args*, ***kwargs*)

This view edits a `Work` object and its affiliated creators.

As with other model views, this should be refactored to move the security code into the model itself rather than re-writing the “edittable” flag within the view code.

`apps.profiles.views.`**`filter`**(*request*)

This view has been deprecated since the introduction of new `Student` and `Staff` models. A more comprehensive browse/filter view is in the planning stages at the time of writing.

`apps.profiles.views.`**`home`**(*request*)

This is the home page of DataMYNE. It calls the `_random_images` function to generate a set of images that contain a mix of user profile images and work images.

`apps.profiles.views.`**`list_profiles`**(*request*, *tag=''*)

This is a deprecated view. It served primarily as destination for tag links, but has since been temporarily removed in favor of a direct link to the search results for a tag.

`apps.profiles.views.`**`stats_report`**`(`*request*, **args*, ***kwargs*`)`
Deprecated. A new admin (with stats) needs to be created.

`apps.profiles.views.`**`view_course`**`(`*request*, *course_id*`)`
This view displays the course. It's surprisingly simple, given how much work has been done with courses over time. In fact, a lot of the work of displaying the page appears on the template itself, for good or ill.

Like other simple model views, this view could benefit from being refactored so that the security logic exists in the Course model itself and not in the course view.

`apps.profiles.views.`**`view_invitation`**`(`*request*, **args*, ***kwargs*`)`
If the user if logged in, this view simply redirects that user to the appropriate view and marks the invite as received.

`apps.profiles.views.`**`view_organization`**`(`*request*, *organization_id*`)`
This view displays an `Organization` object. See the model documentation for a more complete description of what an organization represents.

Along with the `Committee` model, organizations represent one of the most complicated objects in terms of security. This model should be refactored first as part of a general clean up to remove edittable from the view code and port it into the model code itself.

It would also be a good idea to add an "invite" permission that could allow members with fewer security privileges to still invite new members.

Also note the use of the `current` and `past` managers for the member and leader affiliations. This is one of the benefits of using the `Affiliation` object over a `ManyToManyField`: we can retain historical information even after the connection is no longer active. For example, we can know all of the previous leaders of an organization while still allowing the current ones to be the only recipients of security clearance, public display, etc.

`apps.profiles.views.`**`view_person_profile`**`(`*request*, *person_id*`)`
This view serves to unify the `FacultyMember`, `Student`, and `Staff` profile views under a single view. Within the `urls` module, any person's role-specific profile can be reached from here. This allows a common "gateway" to profiles without needing to decide, within the templates, which role a `Person` occupies.

This should be the destination for all profile requests going forward.

`apps.profiles.views.`**`view_profile`**`(`*request*, *person_id*`)`
This is the profile view for `FacultyMembers`. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `view_person_profile` view to reduce repetition.
- edittable defines whether the Edit button appears. This really needs to be refactored out of the view code and into the model code, either at the level of `FacultyMember` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- mlt refers to the `more_like_this` method available in `haystack` (and, in turn, `Solr`). It produces what appears to be a fairly interesting list of related documents (i.e. other DataMYNE objects with sufficient document similarity). However, this could be improved with a faceted search. For example, perhaps only other people and/or works are shown, but not courses or committees. Further user testing is necessary to determine this.
- OpenCalais code exists here which updates the OpenCalais objects in the background. The OC code is still experimental within DataMYNE and should probably either be removed or made subject to a debug setting.

`apps.profiles.views.`**`view_program`**`(`*request*, *program_id*`)`

This view is more or less a stub of pag for displaying the `Program` organizational unit. Ultimately, this should be a place where admins and other authorized users can go to work with program-related functions, like managing `Committee` and `Authority` relationships, etc. Other org units should probably get similar pages.

`apps.profiles.views.`**`view_project_profile`**`(`*request*, *project_id*`)`

This view, and its associated `Project` model, is deprecated and should be refactored out.

`apps.profiles.views.`**`view_section`**`(`*request*, *section_id*`)`

This view displays a `Section` of a `Course` (see the model documentation for a complete description of the distinction between the two.)

This view is notable as being one of the places in which the object-based permissions are used. Object-based permissions are applied at the level of a specific object – in this case, whether the user is allowed to read the syllabus of a specific instructor.

As with other view, the edittable flag here would best be refactored into the model code itself rather than remain in the view.

`apps.profiles.views.`**`view_staff_profile`**`(`*request*, *person_id*`)`

This is the profile view for `Staff` objects. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `view_person_profile` view to reduce repetition.
- edittable defines whether the Edit button appears. This really needs to be refactored out of the view code and into the model code, either at the level of `Staff` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- mlt refers to the `more_like_this` method available in `haystack` (and, in turn, `Solr`). It produces what appears to be a fairly interesting list of related documents (i.e. other DataMYNE objects with sufficient document similarity). However, this could be improved with a faceted search. For example, perhaps only other people and/or works are shown, but not courses or committees. Further user testing is necessary to determine this.
- OpenCalais code exists here which updates the OpenCalais objects in the background. The OC code is still experimental within DataMYNE and should probably either be removed or made subject to a debug setting.

`apps.profiles.views.`**`view_student_profile`**`(`*request*, *person_id*`)`

This is the profile view for `Student` objects. Its function is pretty straightforward, but there are some tricky areas:

- much of this code is similar to other `Person` objects. This could be refactored into the `view_person_profile` view to reduce repetition.
- edittable defines whether the Edit button appears. This really needs to be refactored out of the view code and into the model code, either at the level of `Student` or, at least in part, at the level of `Person` itself. The reason for this is that the combination of Django permissions and unit permissions is becoming increasingly brittle. Unifying this with a well-written object method would improve security, reusability, and performance.
- mlt refers to the `more_like_this` method available in `haystack` (and, in turn, `Solr`). It produces what appears to be a fairly interesting list of related documents (i.e. other DataMYNE objects with sufficient document similarity). However, this could be improved with a faceted search. For example, perhaps only other people and/or works are shown, but not courses or committees. Further user testing is necessary to determine this.

- OpenCalais code exists here which updates the OpenCalais objects in the background. The OC code is still experimental within DataMYNE and should probably either be removed or made subject to a debug setting.

`apps.profiles.views.`**`view_work`**`(`*request*, *work_id*`)`
This view displays a `Work` object and its affiliated creators.

`apps.profiles.views.`**`wordpress`**`(`*request*, *faculty_id*`)`

CHAPTER

THREE

PROFILES: FIELDS

Created on Apr 27, 2011

@author: Mike_Edwards

class `apps.profiles.fields.`**`DataMyneSplitDateTimeField`**`(`**args*, ***kwargs*`)`

Bases: `django.forms.fields.MultiValueField`

based on: http://copiesofcopies.org/webl/2010/04/26/a-better-datetime-widget-for-django/

This field allows for split date/time form entries with ajax-powered widgets

`compress``(`*data_list*`)`

Takes the values from the MultiWidget and passes them as a list to this function. This function needs to compress the list into a single object to save.

`widget`

alias of `DataMyneSplitDateTimeWidget`

CHAPTER

FOUR

PROFILES: FORMS

class `apps.profiles.forms.`**`AdminCourseForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Course`

`AdminCourseForm.`**`media`**

class `apps.profiles.forms.`**`AdminFacultyForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `FacultyMember`

`AdminFacultyForm.`**`clean_bio`**()

`AdminFacultyForm.`**`clean_expertise`**()

`AdminFacultyForm.`**`media`**

class `apps.profiles.forms.`**`AdminSectionForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Section`

`AdminSectionForm.`**`media`**

class `apps.profiles.forms.`**`AdminStaffForm`**`(`*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*`)`

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`

alias of `Staff`

`AdminStaffForm.`**`clean_bio`**`()`

`AdminStaffForm.`**`clean_expertise`**`()`

`AdminStaffForm.`**`media`**

class `apps.profiles.forms.`**`AdminStudentForm`**`(`*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*`)`

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`

alias of `Student`

`AdminStudentForm.`**`clean_bio`**`()`

`AdminStudentForm.`**`clean_expertise`**`()`

`AdminStudentForm.`**`media`**

class `apps.profiles.forms.`**`ContactForm`**`(`*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False*`)`

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`ContactStudentForm`**`(`*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False*`)`

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`CourseForm`**`(`*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*`)`

Bases: `apps.profiles.forms.AdminCourseForm`

class **`Meta`**

Bases: `apps.profiles.forms.Meta`

`CourseForm.`**`media`**

class `apps.profiles.forms.`**`FacultyForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `apps.profiles.forms.AdminFacultyForm`

class **`Meta`**

Bases: `apps.profiles.forms.Meta`

`FacultyForm.`**`media`**

class `apps.profiles.forms.`**`FilterForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`InvitationForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`OrganizationForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`

alias of `Organization`

`OrganizationForm.`**`media`**

class `apps.profiles.forms.`**`PersonActivateForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`SearchForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Bases: `django.forms.forms.Form`

`media`

class `apps.profiles.forms.`**`SectionForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `apps.profiles.forms.AdminSectionForm`

class `Meta` (*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `apps.profiles.forms.AdminSectionForm`

`media`

`SectionForm.`**`media`**

class `apps.profiles.forms.`**`StaffForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `apps.profiles.forms.AdminStaffForm`

class **`Meta`**
Bases: `apps.profiles.forms.Meta`

`StaffForm.`**`media`**

class `apps.profiles.forms.`**`StudentForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `apps.profiles.forms.AdminStudentForm`

class **`Meta`**
Bases: `apps.profiles.forms.Meta`

`StudentForm.`**`media`**

class `apps.profiles.forms.`**`SyllabusForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Section`

`SyllabusForm.`**`media`**

class `apps.profiles.forms.`**`WorkForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Work`

`WorkForm.`**`media`**

class `apps.profiles.forms.`**`WorkURLForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Bases: `django.forms.models.ModelForm`

class Meta

model
alias of `WorkURL`

`WorkURLForm.`**`media`**

CHAPTER

FIVE

PROFILES: HANDLERS

Created on Aug 18, 2010

@author: edwards

class `apps.profiles.handlers.`**`CourseHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns courses.

`model`
alias of `Course`

`queryset`(*request*)

class `apps.profiles.handlers.`**`ExpertiseHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns areas of expertise.

`model`
alias of `Expertise`

`queryset`(*request*)

class `apps.profiles.handlers.`**`FacultyHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns faculty members.

`model`
alias of `FacultyMember`

`queryset`(*request*)

class `apps.profiles.handlers.`**`FacultyResultHandler`**
Bases: `apps.profiles.handlers.ResultHandler`

This `ResultHandler` allows for searches to be faceted to just return the faculty members within a search result set.

`read`(*request*, *model=None*, **args*, ***kwargs*)

class `apps.profiles.handlers.`**`OrganizationHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns organizations.

`model`
alias of `Organization`

queryset (*request*)

class `apps.profiles.handlers.`**PersonHandler**

Bases: `piston.handler.BaseHandler`

This handler is a generic handler for all people. It's most useful in queries like with affiliations where people of multiple roles could be returned (e.g. FacultyMember, Student, Staff)

model

alias of `Person`

queryset (*request*)

class `apps.profiles.handlers.`**ProjectHandler**

Bases: `piston.handler.BaseHandler`

model

alias of `Project`

queryset (*request*)

class `apps.profiles.handlers.`**RecentFacultyHandler**

Bases: `apps.profiles.handlers.FacultyHandler`

This handler returns the 50 most recently updated faculty members.

queryset (*request*)

class `apps.profiles.handlers.`**RecentStudentHandler**

Bases: `apps.profiles.handlers.StudentHandler`

This handler returns the 50 most recently updated students.

queryset (*request*)

class `apps.profiles.handlers.`**RecentWorkHandler**

Bases: `apps.profiles.handlers.WorkHandler`

This handler returns the 50 most recently updated works.

queryset (*request*)

class `apps.profiles.handlers.`**ResultHandler**

Bases: `piston.handler.BaseHandler`

This handler is unique in that is uses a `SearchQuerySet` from `haystack` rather than the typical Django `QuerySet`. This requires a bit of customization with the read output, but allows us to treat `haystack` search results the same way as any other set coming out of the `piston` API application.

convert_search_queryset (*queryset*, *mlt=False*)

model

alias of `SearchResult`

read (*request*, *model=None*, **args*, ***kwargs*)

class `apps.profiles.handlers.`**StudentHandler**

Bases: `piston.handler.BaseHandler`

This handler returns students.

model

alias of `Student`

queryset (*request*)

class `apps.profiles.handlers.`**`StudentResultHandler`**
Bases: `apps.profiles.handlers.ResultHandler`

This `ResultHandler` allows for searches to be faceted to just return the students within a search result set.

`read`(*request*, *model=None*, **args*, ***kwargs*)

class `apps.profiles.handlers.`**`TagHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns tags.

`model`
alias of `Tag`

class `apps.profiles.handlers.`**`TaggedPersonHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns tagged items, but restricts the results to `Student` and `FacultyMember` objects.

`model`
alias of `TaggedItem`

`queryset`(*request*)

class `apps.profiles.handlers.`**`TaggedWorkHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns tagged items, but restricts the results to `Work` objects.

`model`
alias of `TaggedItem`

`queryset`(*request*)

class `apps.profiles.handlers.`**`WorkHandler`**
Bases: `piston.handler.BaseHandler`

This handler returns works.

`model`
alias of `Work`

class `apps.profiles.handlers.`**`WorkResultHandler`**
Bases: `apps.profiles.handlers.ResultHandler`

This `ResultHandler` allows for searches to be faceted to just return the works within a search result set.

`read`(*request*, *model=None*, **args*, ***kwargs*)

CHAPTER

SIX

PROFILES: LOOKUPS

class `apps.profiles.lookups.`**`CourseLookup`**

Bases: `object`

This lookup pulls in `Course` objects to complete ajax-powered form fields.

`format_item`(*course*)

`format_result`(*course*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`DepartmentLookup`**

Bases: `object`

This lookup pulls in `Department` objects to complete ajax-powered form fields.

`format_item`(*object*)

`format_result`(*object*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`DivisionLookup`**

Bases: `object`

This lookup pulls in `Division` objects to complete ajax-powered form fields.

`format_item`(*object*)

`format_result`(*object*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`PersonLookup`**

Bases: `object`

This lookup pulls in `Person` objects to complete ajax-powered form fields.

`format_item`(*person*)

`format_result`(*person*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`ProgramLookup`**
Bases: `object`

This lookup pulls in `Program` objects to complete ajax-powered form fields.

`format_item`(*object*)

`format_result`(*object*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`SchoolLookup`**
Bases: `object`

This lookup pulls in `School` objects to complete ajax-powered form fields.

`format_item`(*object*)

`format_result`(*object*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

class `apps.profiles.lookups.`**`WorkLookup`**
Bases: `object`

This lookup pulls in `Work` objects to complete ajax-powered form fields.

`format_item`(*object*)

`format_result`(*object*)

`get_objects`(*ids*)

`get_query`(*q*, *request*)

CHAPTER

SEVEN

PROFILES: BACKENDS

Created on Mar 2, 2011

@author: edwards

class `apps.profiles.backends.`**`EmailModelBackend`**

Bases: `django.contrib.auth.backends.ModelBackend`

Authenticates against django.contrib.auth.models.User. This is an older backend whose use preceded the NewSchoolLDAPBackend authentication.

`authenticate`(*username=None*, *password=None*)

class `apps.profiles.backends.`**`NewSchoolLDAPBackend`**

Bases: `django_auth_ldap.backend.LDAPBackend`

This is the LDAP authentication for New School faculty, students and staff. At present, it does its best to figure out whether a new person is one of the three roles contained in the system. In the future, we will need to find ways to create hybrid profiles for users who occupy multiple roles.

To look into the LDAP itself, you can use the following commands (replace the example text with actual values)

```
>>> import ldap
>>> conn = ldap.initialize("ldaps://your.ldap.server.edu")
>>> conn.bind_s("cn=commonLoginName,o=organization","password",ldap.AUTH_SIMPLE)
>>> conn.search_s("o=organization",ldap.SCOPE_SUBTREE, "(&(objectclass=user)(sn=Lastname)(givenN
```

`get_or_create_user`(*username*, *ldap_user*)

CHAPTER

EIGHT

REPORTING: MODELS

class `apps.reporting.models.`**`Affiliation`**`(`**args*, ***kwargs*`)`

Bases: `datamining.apps.profiles.models.BaseModel`

The `Affiliation` objects are both useful and pervasive in the current DataMYNE system. In almost all cases, they have superseded the use of the `ManyToManyField` for the *‘‘Person‘‘ to other objects* relationships. They have the following advantages:

- They are generic. This means we do not need to redefine the relationship field on every new object we make. Instead, we can assume that **any** new model will be able to form a many-to-many relationship to a person via an `Affiliation`.
- They have a `Role`. We can therefore create several different kinds of relationships between a `Person` and an object. For example, a `Committee` can have both a chairperson and a member.
- They have a start and end date. This allows us to maintain old relationships, and embargo new relationships, without have to delete links. This is useful historically and for data-mining purposes. For example, we can create a history of all of a `Committee`‘s chairs as far back as we like.
  - The use of the `begin` and `retire` methods is encouraged in maintaining current and past affiliations. An `embargo` method would probably also be useful for maintaining future affiliations (e.g. an incoming committee chair.)

exception **`DoesNotExist`**

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `Affiliation.`**`MultipleObjectsReturned`**

Bases: `django.core.exceptions.MultipleObjectsReturned`

`Affiliation.`**`begin`**`()`

`Affiliation.`**`content_object`**

Provides a generic relation to any object through content-type/object-id fields.

`Affiliation.`**`content_type`**

`Affiliation.`**`created_by`**

`Affiliation.`**`get_next_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Affiliation.`**`get_next_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Affiliation.`**`get_previous_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Affiliation.`**`get_previous_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Affiliation.`**`person`**

`Affiliation.`**`retire`**`()`

Affiliation.**role**

Affiliation.**unit_permissions**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class apps.reporting.models.**AffiliationCurrentManager**

Bases: django.db.models.manager.Manager

This manager show affiliations that exist between two dates OR without any dates OR where there is a previous start but not an end OR where there is no start date but a future end date.

This affiliation manager, like all the others, allows for affiliations to be begun or retired en masse.

begin_all(*role*, *content_type*, *object_id*)

get_query_set()

retire_all(*role*, *content_type*, *object_id*)

class apps.reporting.models.**AffiliationFutureManager**

Bases: apps.reporting.models.AffiliationCurrentManager

This manager only lists affiliations whose start date is in the future.

get_query_set()

class apps.reporting.models.**AffiliationPastManager**

Bases: apps.reporting.models.AffiliationCurrentManager

This manager only lists affiliations whose end date is in the past.

get_query_set()

class apps.reporting.models.**Authority**(**args*, ***kwargs*)

Bases: datamining.apps.profiles.models.BaseModel

An Authority defines the control of a committee by an organizational unit within the university (e.g. a Division).

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception Authority.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

Authority.**committee**

Authority.**content_object**

Provides a generic relation to any object through content-type/object-id fields.

Authority.**content_type**

Authority.**created_by**

Authority.**get_next_by_created_at**(**moreargs*, ***morekwargs*)

Authority.**get_next_by_updated_at**(**moreargs*, ***morekwargs*)

Authority.**get_previous_by_created_at**(**moreargs*, ***morekwargs*)

Authority.**get_previous_by_updated_at**(**moreargs*, ***morekwargs*)

`Authority.`**`unit_permissions`**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.reporting.models.`**`Committee`**`(`**args*, ***kwargs*`)`

Bases: `datamining.apps.profiles.models.BaseModel`

A `Committee` is an official designated group of people who have a mandate and who presumably meet regularly. A committee may have a parent committee to which it reports. It is more structured than an `Organization` and is attached to organization units such as `Division` or `Program` via the `Authority` objects.

exception **`DoesNotExist`**

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `Committee.`**`MultipleObjectsReturned`**

Bases: `django.core.exceptions.MultipleObjectsReturned`

`Committee.`**`accept_invitation`**`(`*invitation*`)`

`Committee.`**`affiliations`**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`Committee.`**`authorities`**

`Committee.`**`created_by`**

`Committee.`**`get_absolute_url`**`(`**moreargs*, ***morekwargs*`)`

`Committee.`**`get_next_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Committee.`**`get_next_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Committee.`**`get_previous_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Committee.`**`get_previous_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Committee.`**`get_unit`**`()`

`Committee.`**`meetings`**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`Committee.`**`parent`**

`Committee.`**`subcommittees`**

`Committee.`**`unit_permissions`**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.reporting.models.`**`Meeting`**`(`**args*, ***kwargs*`)`

Bases: `datamining.apps.profiles.models.BaseModel`

A `Meeting` generically connects itself to any other object. This allows other models like `Committee` and `Organization` to use the same meeting code.

exception **`DoesNotExist`**
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `Meeting.`**`MultipleObjectsReturned`**
Bases: `django.core.exceptions.MultipleObjectsReturned`

`Meeting.`**`accept_invitation`**(*invitation*)

`Meeting.`**`content_object`**
Provides a generic relation to any object through content-type/object-id fields.

`Meeting.`**`content_type`**

`Meeting.`**`created_by`**

`Meeting.`**`get_absolute_url`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_next_by_created_at`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_next_by_end_time`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_next_by_start_time`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_next_by_updated_at`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_previous_by_created_at`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_previous_by_end_time`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_previous_by_start_time`**(**moreargs*, ***morekwargs*)

`Meeting.`**`get_previous_by_updated_at`**(**moreargs*, ***morekwargs*)

`Meeting.`**`unit_permissions`**
This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

class `apps.reporting.models.`**`MeetingManager`**
Bases: `django.db.models.manager.Manager`

`daily_occurrences`(*dt=None*, *content_type=None*, *object_id=None*)
Returns a queryset of for instances that have any overlap with a particular day.

- `dt` may be either a datetime.datetime, datetime.date object, or `None`. If `None`, default to the current day.
- `event` can be an `Event` instance for further filtering.

`monthly_occurrences`(*dt=None*, *content_type=None*, *object_id=None*)
Returns a queryset of for instances that have any overlap with a particular day.

- `dt` may be either a datetime.datetime, datetime.date object, or `None`. If `None`, default to the current day.
- `event` can be an `Event` instance for further filtering.

`range_occurences`(*start=None*, *end=None*, *content_type=None*, *object_id=None*)

`weekly_occurrences`(*dt=None*, *content_type=None*, *object_id=None*)
Returns a queryset of for instances that have any overlap with a particular day.

- `dt` may be either a datetime.datetime, datetime.date object, or `None`. If `None`, default to the current day.
- `event` can be an `Event` instance for further filtering.

class `apps.reporting.models.`**`Role`**`(`**args*, ***kwargs*`)`

Bases: `datamining.apps.profiles.models.BaseModel`

A `Role` is a refinement of `Affiliation` between a `Person` and another object. A roles has:

title A role's title is simply that. Good examples include "creator," as in "Jane Doe is the creator of Artwork X" and "chairperson," as in "John Does is the chairperson of Committee ABC"

content_type A role must also have a content type. This allows for there to be a distinct "member" role, for example, in affiliations to both a `Committee` and an `Organization`.

exception **`DoesNotExist`**

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `Role.`**`MultipleObjectsReturned`**

Bases: `django.core.exceptions.MultipleObjectsReturned`

`Role.`**`affiliations`**

`Role.`**`content_type`**

`Role.`**`created_by`**

`Role.`**`get_next_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Role.`**`get_next_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Role.`**`get_previous_by_created_at`**`(`**moreargs*, ***morekwargs*`)`

`Role.`**`get_previous_by_updated_at`**`(`**moreargs*, ***morekwargs*`)`

`Role.`**`unit_permissions`**

This class provides the functionality that makes the related-object managers available as attributes on a model class, for fields that have multiple "remote" values and have a GenericRelation defined in their model (rather than having another model pointed *at* them). In the example "article.publications", the publications attribute is a ReverseGenericRelatedObjectsDescriptor instance.

`apps.reporting.models.`**`delete_area_of_study_affiliations`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_committee_affiliations`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_committee_meetings`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_department_authorities`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_division_authorities`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_organization_affiliations`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_organization_meetings`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_program_affiliations`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_program_authorities`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`delete_school_authorities`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`index_committee`**`(`*sender*, **args*, ***kwargs*`)`

`apps.reporting.models.`**`index_meeting`**`(`*sender*, **args*, ***kwargs*`)`

CHAPTER

NINE

REPORTING: VIEWS

`apps.reporting.views.`**`edit_committee`**`(`*request*, **args*, ***kwargs*`)`

This view edits a `Committee` object. See the model documentation for a more complete description of what an organization represents.

Along with the `Organization` model, committees represent one of the most complicated objects in terms of security. This model should be refactored first as part of a general clean up to remove edittable from the view code and port it into the model code itself.

Also note the use of the `current` and `past` managers for the member and chairperson affiliations. This is one of the benefits of using the `Affiliation` object over a `ManyToManyField`: we can retain historical information even after the connection is no longer active. For example, we can know all of the previous chairs of a committee while still allowing the current ones to be the only recipients of security clearance, public display, etc.

In addition, the `Affiliation` managers have the `begin` and `retire` methods that allow connections to see easily set to current or past without having to rewrite complicated code. “Retirement” is the preferred way for disposing of a current affiliation. It has the same effect as a deletion, while still retaining the connections for historical and data-mining purposes.

`apps.reporting.views.`**`edit_meeting`**`(`*request*, **args*, ***kwargs*`)`

This view edits a meeting.

Note that meetings, currently, can be attached to any object, specifically the `Organization` and `Committee` objects. This creates a rather complex security situation, in that the `Meeting` mode is not only capable of having permissions assigned to it directly but, logically, is also subject to the admin permissions for the `Organization` and `Committee`. As with other models, this should be refactored, to as great an extent as possible, into the models themselves.

`apps.reporting.views.`**`list_committees_by_school`**`(`*request*`)`

This view lists all of the committees of all the schools.

`apps.reporting.views.`**`view_committee`**`(`*request*, *committee_id*`)`

This view displays an `Committee` object. See the model documentation for a more complete description of what an organization represents.

Along with the `Organization` model, committees represent one of the most complicated objects in terms of security. This model should be refactored first as part of a general clean up to remove edittable from the view code and port it into the model code itself.

It would be good to add a “admin” role to the committees, in addition to the current “chairperson” and “member” roles.

Also note the use of the `current` and `past` managers for the member and chairperson affiliations. This is one of the benefits of using the `Affiliation` object over a `ManyToManyField`: we can retain historical information even after the connection is no longer active. For example, we can know all of the previous chairs of a committee while still allowing the current ones to be the only recipients of security clearance, public display, etc.

`apps.reporting.views.`**`view_meeting`**`(`*request*, *meeting_id*`)`

This view displays a meeting.

Note that meetings, currently, can be attached to any object, specifically the `Organization` and `Committee` objects. This creates a rather complex security situation, in that the `Meeting` mode is not only capable of having permissions assigned to it directly but, logically, is also subject to the admin permissions for the `Organization` and `Committee`. As with other models, this should be refactored, to as great an extent as possible, into the models themselves.

CHAPTER

TEN

REPORTING: FORMS

Created on Apr 7, 2011

@author: Mike_Edwards

class `apps.reporting.forms.`**`CommitteeAffiliationForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Affiliation`

`CommitteeAffiliationForm.`**`media`**

class `apps.reporting.forms.`**`CommitteeForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Committee`

`CommitteeForm.`**`media`**

class `apps.reporting.forms.`**`MeetingForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)

Bases: `django.forms.models.ModelForm`

class **`Meta`**

`model`
alias of `Meeting`

`MeetingForm.`**`media`**

CHAPTER

ELEVEN

REPORTING: HANDLERS

Created on Aug 18, 2010

@author: edwards

class `apps.reporting.handlers.`**`CommitteeHandler`**

Bases: `piston.handler.BaseHandler`

This handler returns committees.

`model`

alias of `Committee`

`queryset`(*request*)

class `apps.reporting.handlers.`**`StaffHandler`**

Bases: `piston.handler.BaseHandler`

This handler returns staff members.

`model`

alias of `Staff`

`queryset`(*request*)

CHAPTER

TWELVE

MOBILE: VIEWS

`apps.mobile.views.`**`home`**`(`*request*`)`

CHAPTER

THIRTEEN

INDICES AND TABLES

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

PYTHON MODULE INDEX