Philo Documentation

Release 0.9.1

iThink Software

October 08, 2011

CONTENTS

Philo is a foundation for developing web content management systems. Please, read the `notes for our latest release`.

Prerequisites:

- Python 2.5.4+
- Django 1.3+
- django-mptt e734079+
- (optional) django-grappelli 2.0+
- (optional) south 0.7.2+
- (`philo.contrib.penfield`) django-taggit 0.9.3+
- (`philo.contrib.waldo`, optional) recaptcha-django r6+

CHAPTER

ONE

CONTENTS

1.1 What is Philo, anyway?

Philo allows the creation of site structures using Django's built-in admin interface. Like Django, Philo separates URL structure from backend code from display:

- `Node`s represent the URL hierarchy of the website.
- `View`s contain the logic for each `Node`, as simple as a `Redirect` or as complex as a `Blog`.
- `Page`s (the most commonly used `View`) render whatever context they are passed using database-driven `Template`s written with Django's template language.
- `Attribute`s are arbitrary key/value pairs which can be attached to most of the models that Philo provides. Attributes of a `Node` will be inherited by all of the `Node`'s descendants and will be available in the template's context.

The `container` template tag that Philo provides makes it easy to mark areas in a template which need to be editable page-by-page; every `Page` will have an additional field in the admin for each `container` in the template it uses.

1.1.1 How's that different than other CMSes?

Philo developed according to principles that grew out of the observation of the limitations and practices of other content management systems. For example, Philo believes that:

- **Designers are in charge of how content is displayed, not end users. For example, users should be able to embed images in**

  See Also:

  `embed`

- **Interpretation of content (as a django template, as markdown, as textile, etc.) is the responsibility of the template designer**

  See Also:

  `include_string`

- **Page content should be simple – not reorderable. Each piece of content should only be related to one page. Any other syst**

  See Also:

  `Contentlet`, `ContentReference`

- **Some pieces of information may be shared by an entire site, used in disparate places, and changed frequently enough that**

  See Also:

  Attribute

1.2 Tutorials

1.2.1 Getting started with philo

Note: This guide assumes that you have worked with Django's built-in administrative interface.

Once you've installed philo and mptt to your python path, there are only a few things that you need to do to get philo working.

1. Add philo and mptt to settings.INSTALLED_APPS:

```
INSTALLED_APPS = (
        ...
        'philo',
        'mptt',
        ...
)
```

2. Syncdb or run migrations to set up your database.

3. Add philo.middleware.RequestNodeMiddleware to settings.MIDDLEWARE_CLASSES:

```
MIDDLEWARE_CLASSES = (
        ...
        'philo.middleware.RequestNodeMiddleware',
        ...
)
```

4. Include philo.urls somewhere in your urls.py file. For example:

```
from django.conf.urls.defaults import patterns, include, url
urlpatterns = patterns('',
        url(r'^', include('philo.urls')),
)
```

Philo should be ready to go! (Almost.)

Hello world

Now that you've got everything configured, it's time to set up your first page! Easy peasy. Open up the admin and add a new Template. Call it "Hello World Template". The code can be something like this:

```
<html>
        <head>
                <title>Hello world!</title>
        </head>
        <body>
                <p>Hello world!</p>
                <p>The time is {% now %}.</p>
        </body>
</html>
```

Next, add a philo Page - let's call it "Hello World Page" and use the template you just made.

Now make a philo Node. Give it the slug hello-world. Set the view_content_type to "Page" and the view_object_id to the id of the page that you just made - probably 1. If you navigate to /hello-world, you will see the results of rendering the page!

Setting the root node

So what's at /? If you try to load it, you'll get a 404 error. This is because there's no Node located there - and since Node.slug is a required field, getting a node there is not as simple as leaving the slug blank.

In philo, the node that is displayed at / is called the "root node" of the current Site. To represent this idea cleanly in the database, philo adds a ForeignKey to Node to the django.contrib.sites.models.Site model.

Since there's only one Node in your Site, we probably want hello-world to be the root node. All you have to do is edit the current Site and set its root node to hello-world. Now you can see the page rendered at /!

Editing page contents

Great! We've got a page that says "Hello World". But what if we want it to say something else? Should we really have to edit the Template to change the content of the Page? And what if we want to share the Template but have different content? Adjust the Template to look like this:

```
<html>
    <head>
        <title>{% container page_title %}</title>
    </head>
    <body>
        {% container page_body as content %}
        {% if content %}
            <p>{{ content }}</p>
        {% endif %}
        <p>The time is {% now "jS F Y H:i" %}.</p>
    </body>
</html>
```

Now go edit your Page. Two new fields called "Page title" and "Page body" have shown up! You can put anything you like in here and have it show up in the appropriate places when the page is rendered.

See Also:

philo.templatetags.containers.container

Congrats! You've done it!

1.2.2 Using Shipherd in the Admin

The navigation mechanism is fairly complex; unfortunately, there's no real way around that - without a lot of equally complex code that you are quite welcome to write and contribute! ;-)

For this guide, we'll assume that you have the setup described in *Getting started with philo*. We'll be adding a main Navigation to the root Node and making it display as part of the Template.

Before getting started, make sure that you've added philo.contrib.shipherd to your INSTALLED_APPS. shipherd template tags also require the request context processor, so make sure to set TEMPLATE_CONTEXT_PROCESSORS appropriately:

```
TEMPLATE_CONTEXT_PROCESSORS = (
        # Defaults
        "django.contrib.auth.context_processors.auth",
        "django.core.context_processors.debug",
        "django.core.context_processors.i18n",
        "django.core.context_processors.media",
        "django.core.context_processors.static",
        "django.contrib.messages.context_processors.messages"
        ...
        "django.core.context_processors.request"
)
```

Creating the Navigation

Start off by adding a new Navigation instance with node set to the good ole' root node and key set to main. The default depth of 3 is fine.

Now open up that first inline NavigationItem. Make the text Hello World and set the target Node to, again, root. (Of course, this is a special case. If we had another node that we wanted to point to, we would choose that.)

Press save and you've created your first navigation.

Displaying the Navigation

All you need to do now is show the navigation in the template! This is quite easy, using the recursenavigation templatetag. For now we'll keep it simple. Adjust the "Hello World Template" to look like this:

```
<html>{% load shipherd %}
    <head>
        <title>{% container page_title %}</title>
    </head>
    <body>
        <ul>
            {% recursenavigation node "main" %}
                <li{% if navloop.active %} class="active"{% endif %}>
                    <a href="{{ item.get_target_url }}">{{ item.text }}</a>
                </li>
            {% endrecursenavigation %}
        </ul>
        {% container page_body as content %}
        {% if content %}
            <p>{{ content }}</p>
        {% endif %}
        <p>The time is {% now %}.</p>
    </body>
</html>
```

Now have a look at the page - your navigation is there!

Linking to google

Edit the `main` `Navigation` again to add another `NavigationItem`. This time give it the `text` `Google` and set the `url_or_subpath` field to `http://google.com`. A navigation item will show up on the Hello World page that points to `google.com`! Granted, your navigation probably shouldn't do that, because confusing navigation is confusing; the point is that it is possible to provide navigation to arbitrary URLs.

`url_or_subpath` can also be used in conjuction with a `Node` to link to a subpath beyond that `Node`'s url.

1.3 Philo's models

Contents:

1.3.1 Entities and Attributes

One of the core concepts in Philo is the relationship between the `Entity` and `Attribute` classes. `Attribute`s represent an arbitrary key/value pair by having one `GenericForeignKey` to an `Entity` and another to an `AttributeValue`.

Attributes

class `philo.models.base.`**`Attribute`**(**args*, ***kwargs*)

`Attribute`s exist primarily to let arbitrary data be attached to arbitrary model instances without altering the database schema and without guaranteeing that the data will be available on every instance of that model.

Generally, `Attribute`s will not be accessed as models; instead, they will be accessed through the `Entity.attributes` property, which allows direct dictionary getting and setting of the value of an `Attribute` with its key.

`entity`

`GenericForeignKey` to anything (generally an instance of an Entity subclass).

`value`

`GenericForeignKey` to an instance of a subclass of `AttributeValue` as determined by the `attribute_value_limiter`.

`key`

`CharField` containing a key (up to 255 characters) consisting of alphanumeric characters and underscores.

`set_value`(*value*, *value_class=<class 'philo.models.base.JSONValue'>*)

Given a value and a value class, sets up self.value appropriately.

class `philo.models.base.`**`AttributeValue`**(**args*, ***kwargs*)

This is an abstract base class for models that can be used as values for `Attribute`s.

AttributeValue subclasses are expected to supply access to a clean version of their value through an attribute called "value".

`set_value`(*value*)

Given a `value`, sets the appropriate fields so that it can be correctly stored in the database.

`value_formfields`(***kwargs*)

Returns any formfields that would be used to construct an instance of this value.

Returns A dictionary mapping field names to formfields.

construct_instance(***kwargs*)
Applies cleaned data from the formfields generated by valid_formfields to oneself.

philo.models.base.**attribute_value_limiter**
An instance of ContentTypeSubclassLimiter which is used to track the content types which are considered valid value models for an Attribute.

class philo.models.base.**JSONValue**(**args*, ***kwargs*)
Bases: philo.models.base.AttributeValue

Stores a python object as a json string.

class philo.models.base.**ForeignKeyValue**(**args*, ***kwargs*)
Bases: philo.models.base.AttributeValue

Stores a generic relationship to an instance of any value content type (as defined by the value_content_type_limiter).

class philo.models.base.**ManyToManyValue**(**args*, ***kwargs*)
Bases: philo.models.base.AttributeValue

Stores a generic relationship to many instances of any value content type (as defined by the value_content_type_limiter).

philo.models.base.**value_content_type_limiter**
An instance of ContentTypeRegistryLimiter which is used to track the content types which can be related to by ForeignKeyValues and ManyToManyValues.

philo.models.base.**register_value_model**(*model*)
Registers a model as a valid content type for a ForeignKeyValue or ManyToManyValue through the value_content_type_limiter.

philo.models.base.**unregister_value_model**(*model*)
Registers a model as a valid content type for a ForeignKeyValue or ManyToManyValue through the value_content_type_limiter.

Entities

class philo.models.base.**Entity**(**args*, ***kwargs*)
An abstract class that simplifies access to related attributes. Most models provided by Philo subclass Entity.

class philo.models.base.**TreeEntityManager**

get_with_path(*path*, *root=None*, *absolute_result=True*, *pathsep='/'*, *field='pk'*)
If absolute_result is True, returns the object at path (starting at root) or raises an ObjectDoesNotExist exception. Otherwise, returns a tuple containing the deepest object found along path (or root if no deeper object is found) and the remainder of the path after that object as a string (or None if there is no remaining path).

Note: If you are looking for something with an exact path, it is faster to use absolute_result=True, unless the path depth is over ~40, in which case the high cost of the absolute query may make a binary search (i.e. non-absolute) faster.

Note: SQLite allows max of 64 tables in one join. That means the binary search will only work on paths with a max depth of 127 and the absolute fetch will only work to a max depth of (surprise!) 63. Larger depths could be handled, but since the common use case will not have a tree structure that deep, they are not.

Parameters

- **path** – The path of the object
- **root** – The object which will be considered the root of the search
- **absolute_result** – Whether to return an absolute result or do a binary search
- **pathsep** – The path separator used in `path`
- **field** – The field on the model which should be queried for `path` segment matching.

Returns An instance if `absolute_result` is `True` or an (instance, remaining_path) tuple otherwise.

Raises django.core.exceptions.ObjectDoesNotExist if no object can be found matching the input parameters.

class `philo.models.base.`**`TreeEntity`**(**args*, ***kwargs*)

Bases: `philo.models.base.Entity`, `mptt.models.MPTTModel`

An abstract subclass of Entity which represents a tree relationship.

`objects`

An instance of `TreeEntityManager`.

`get_path`(*root=None*, *pathsep='/'*, *field='pk'*, *memoize=True*)

Parameters

- **root** – Only return the path since this object.
- **pathsep** – The path separator to use when constructing an instance's path
- **field** – The field to pull path information from for each ancestor.
- **memoize** – Whether to use memoized results. Since, in most cases, the ancestors of a TreeEntity will not change over the course of an instance's lifetime, this defaults to `True`.

Returns A string representation of an object's path.

`get_path`(*root=None*, *pathsep='/'*, *field='pk'*, *memoize=True*)

Parameters

- **root** – Only return the path since this object.
- **pathsep** – The path separator to use when constructing an instance's path
- **field** – The field to pull path information from for each ancestor.
- **memoize** – Whether to use memoized results. Since, in most cases, the ancestors of a TreeEntity will not change over the course of an instance's lifetime, this defaults to `True`.

Returns A string representation of an object's path.

`path`

Parameters

- **root** – Only return the path since this object.
- **pathsep** – The path separator to use when constructing an instance's path
- **field** – The field to pull path information from for each ancestor.

- **memoize** – Whether to use memoized results. Since, in most cases, the ancestors of a TreeEntity will not change over the course of an instance's lifetime, this defaults to `True`.

Returns A string representation of an object's path.

1.3.2 Nodes and Views: Building Website structure

Nodes

class `philo.models.nodes.`**`Node`**`(`**args*, ***kwargs*`)`
Bases: `philo.models.base.SlugTreeEntity`

`Node`s are the basic building blocks of a website using Philo. They define the URL hierarchy and connect each URL to a `View` subclass instance which is used to generate an HttpResponse.

`view`
`GenericForeignKey` to a non-abstract subclass of `View`

`accepts_subpath`
A property shortcut for `self.view.accepts_subpath`

`render_to_response``(`*request*, *extra_context=None*`)`
This is a shortcut method for `View.render_to_response()`

`get_absolute_url``(`**moreargs*, ***morekwargs*`)`
This is essentially a shortcut for calling `construct_url()` without a subpath.

Returns The absolute url of the node on the current site.

`construct_url``(`*subpath='/'*, *request=None*, *with_domain=False*, *secure=False*`)`
This method will do its best to construct a URL based on the Node's location. If with_domain is True, that URL will include a domain and a protocol; if secure is True as well, the protocol will be https. The request will be used to construct a domain in cases where a call to `Site.objects.get_current()` fails.

Node urls will not contain a trailing slash unless a subpath is provided which ends with a trailing slash. Subpaths are expected to begin with a slash, as if returned by `django.core.urlresolvers.reverse()`.

Because this method will be called frequently and will always try to reverse `philo-root`, the results of that reversal will be cached by default. This can be disabled by setting `PHILO_CACHE_PHILO_ROOT` to `False`.

`construct_url()` may raise the following exceptions:

- `NoReverseMatch` if "philo-root" is not reversable – for example, if `philo.urls` is not included anywhere in your urlpatterns.
- `Site.DoesNotExist` if with_domain is True but no `Site` or `RequestSite` can be built.
- `AncestorDoesNotExist` if the root node of the site isn't an ancestor of the node constructing the URL.

Parameters

- **subpath** (*string*) – The subpath to be constructed beyond beyond the node's URL.
- **request** – `HttpRequest` instance. Will be used to construct a `RequestSite` if `Site.objects.get_current()` fails.
- **with_domain** – Whether the constructed URL should include a domain name and protocol.

- **secure** – Whether the protocol, if included, should be http:// or https://.

Returns A constructed url for accessing the given subpath of the current node instance.

Views

Abstract View Models

class `philo.models.nodes.View`(**args*, ***kwargs*)

Bases: `philo.models.base.Entity`

`View` is an abstract model that represents an item which can be "rendered", generally in response to an `HttpRequest`.

`accepts_subpath`

An attribute on the class which defines whether this `View` can handle subpaths. Default: `False`

classmethod `handles_subpath`(*subpath*)

Returns True if the `View` handles the given subpath, and False otherwise.

`reverse`(*view_name=None*, *args=None*, *kwargs=None*, *node=None*, *obj=None*)

If `accepts_subpath` is True, try to reverse a URL using the given parameters using `self` as the urlconf.

If `obj` is provided, `get_reverse_params()` will be called and the results will be combined with any `view_name`, `args`, and `kwargs` that may have been passed in.

Parameters

- **view_name** – The name of the view to be reversed.
- **args** – Extra args for reversing the view.
- **kwargs** – A dictionary of arguments for reversing the view.
- **node** – The node whose subpath this is.
- **obj** – An object to be passed to `get_reverse_params()` to generate a view_name, args, and kwargs for reversal.

Returns A subpath beyond the node that reverses the view, or an absolute url that reverses the view if a node was passed in.

Raises

- **philo.exceptions.ViewDoesNotProvideSubpaths** – if `accepts_subpath` is False
- **philo.exceptions.ViewCanNotProvideSubpath** – if a reversal is not possible.

`get_reverse_params`(*obj*)

This method is not implemented on the base class. It should return a (`view_name`, `args`, `kwargs`) tuple suitable for reversing a url for the given `obj` using `self` as the urlconf. If a reversal will not be possible, this method should raise `ViewCanNotProvideSubpath`.

`attributes_with_node`(*node*, *mapper=<class 'philo.utils.entities.LazyPassthroughAttributeMapper'>*)

Returns a `LazyPassthroughAttributeMapper` which can be used to directly retrieve the values of `Attributes` related to the `View`, falling back on the `Attributes` of the passed-in `Node` and its ancestors.

`render_to_response`(*request*, *extra_context=None*)

Renders the `View` as an `HttpResponse`. This will raise `MIDDLEWARE_NOT_CONFIGURED` if the *request* doesn't have an attached `Node`. This can happen if the `RequestNodeMiddleware` is not in `settings.MIDDLEWARE_CLASSES` or if it is not functioning correctly.

`render_to_response()` will send the `view_about_to_render` signal, then call `actually_render_to_response()`, and finally send the `view_finished_rendering` signal before returning the `response`.

`actually_render_to_response`(*request*, *extra_context=None*)
Concrete subclasses must override this method to provide the business logic for turning a `request` and `extra_context` into an `HttpResponse`.

class `philo.models.nodes.`**`MultiView`**(**args*, ***kwargs*)
Bases: `philo.models.nodes.View`

`MultiView` is an abstract model which represents a section of related pages - for example, a `BlogView` might have a foreign key to `Pages` for an index, an entry detail, an entry archive by day, and so on. `MultiView` subclasses `View`, and defines the following additional methods and attributes:

`accepts_subpath`
Same as `View.accepts_subpath`. Default: `True`

`urlpatterns`
Returns urlpatterns that point to views (generally methods on the class). `MultiView`s can be thought of as "managing" these subpaths.

`actually_render_to_response`(*request*, *extra_context=None*)
Resolves the remaining subpath left after finding this `View`'s node using `self.urlpatterns` and renders the view function (or method) found with the appropriate args and kwargs.

`get_context`()
Hook for providing instance-specific context - such as the value of a Field - to any view methods on the instance.

`basic_view`(*field_name*)
Given the name of a field on the class, accesses the value of that field and treats it as a `View` instance. Creates a basic context based on self.get_context() and any extra_context that was passed in, then calls the `View` instance's render_to_response() method. This method is meant to be called to return a view function appropriate for urlpatterns.

Parameters **field_name** – The name of a field on the instance which contains a `View` subclass instance.

Returns A simple view function.

Example:

```
class Foo(Multiview):
        page = models.ForeignKey(Page)

        @property
        def urlpatterns(self):
                urlpatterns = patterns('',
                        url(r'^$', self.basic_view('page'))
                )
                return urlpatterns
```

Concrete View Subclasses

class `philo.models.nodes.`**`Redirect`**(**args*, ***kwargs*)
Bases: `philo.models.nodes.TargetURLModel`, `philo.models.nodes.View`

Represents a 301 or 302 redirect to a different url on an absolute or relative path.

STATUS_CODES
A choices tuple of redirect status codes (temporary or permanent).

status_code
An `IntegerField` which uses `STATUS_CODES` as its choices. Determines whether the redirect is considered temporary or permanent.

actually_render_to_response(*request*, *extra_context=None*)
Returns an `HttpResponseRedirect` to `self.target_url`.

class `philo.models.nodes.`**File**(**args*, ***kwargs*)
Bases: `philo.models.nodes.View`

Stores an arbitrary file.

name
The name of the uploaded file. This is meant for finding the file again later, not for display.

mimetype
Defines the mimetype of the uploaded file. This will not be validated. If no mimetype is provided, it will be automatically generated based on the filename.

file
Contains the uploaded file. Files are uploaded to `philo/files/%Y/%m/%d`.

Pages `Page`s are the most frequently used `View` subclass. They define a basic HTML page and its associated content. Each `Page` renders itself according to a `Template`. The `Template` may contain `container` tags, which define related `Contentlet`s and `ContentReference`s for any page using that `Template`.

class `philo.models.pages.`**Page**(**args*, ***kwargs*)
Bases: `philo.models.nodes.View`

Represents a page - something which is rendered according to a `Template`. The page will have a number of related `Contentlet`s and `ContentReference`s depending on the template selected - but these will appear only after the page has been saved with that template.

template
A `ForeignKey` to the `Template` used to render this `Page`.

title
The name of this page. Chances are this will be used for organization - i.e. finding the page in a list of pages - rather than for display.

get_containers()
Returns the results `containers` for the related template. This is a tuple containing the specs of all `containers` in the `Template`'s code. The value will be cached on the instance so that multiple accesses will be less expensive.

containers
Returns the results `containers` for the related template. This is a tuple containing the specs of all `containers` in the `Template`'s code. The value will be cached on the instance so that multiple accesses will be less expensive.

render_to_string(*request=None*, *extra_context=None*)
In addition to rendering as an `HttpResponse`, a `Page` can also render as a string. This means, for example, that `Page`s can be used to render emails or other non-HTML content with the same `container`-based functionality as is used for HTML.

The `Page` will add itself to the context as `page` and its `attributes` as `attributes`. If a request is provided, then `request.node` will also be added to the context as `node` and `attributes` will be set to the result of calling `attributes_with_node()` with that `Node`.

`actually_render_to_response`(*request*, *extra_context=None*)
Returns an `HttpResponse` with the content of the `render_to_string()` method and the mimetype set to the `mimetype` of the related `Template`.

`clean_fields`(*exclude=None*)
This is an override of the default model clean_fields method. Essentially, in addition to validating the fields, this method validates the `Template` instance that is used to render this `Page`. This is useful for catching template errors before they show up as 500 errors on a live site.

class `philo.models.pages.`**`Template`**(**args*, ***kwargs*)
Bases: `philo.models.base.SlugTreeEntity`

Represents a database-driven django template.

See Also:

`philo.loaders.database`

`name`
The name of the template. Used for organization and debugging.

`documentation`
Can be used to let users know what the template is meant to be used for.

`mimetype`
Defines the mimetype of the template. This is not validated. Default: `text/html`.

`code`
An insecure `TemplateField` containing the django template code for this template.

`get_containers`()
Returns a tuple where the first item is a list of names of contentlets referenced by containers, and the second item is a list of tuples of names and contenttypes of contentreferences referenced by containers. This will break if there is a recursive extends or includes in the template code. Due to the use of an empty Context, any extends or include tags with dynamic arguments probably won't work.

`containers`
Returns a tuple where the first item is a list of names of contentlets referenced by containers, and the second item is a list of tuples of names and contenttypes of contentreferences referenced by containers. This will break if there is a recursive extends or includes in the template code. Due to the use of an empty Context, any extends or include tags with dynamic arguments probably won't work.

class `philo.models.pages.`**`Contentlet`**(**args*, ***kwargs*)
Represents a piece of content on a page. This content is treated as a secure `TemplateField`.

`page`
The page which this `Contentlet` is related to.

`name`
This represents the name of the container as defined by a `container` tag.

`content`
A secure `TemplateField` holding the content for this `Contentlet`. Note that actually using this field as a template requires use of the `include_string` template tag.

class `philo.models.pages.`**`ContentReference`**(**args*, ***kwargs*)
Represents a model instance related to a page.

page
The page which this ContentReference is related to.

name
This represents the name of the container as defined by a `container` tag.

content
A `GenericForeignKey` to a model instance. The content type of this instance is defined by the `container` tag which defines this ContentReference.

1.3.3 Collections

class `philo.models.collections.`**`Collection`**(**args*, ***kwargs*)
Collections are curated ordered groupings of arbitrary models.

name
`CharField` with max_length 255

description
Optional `TextField`

get_count()
Returns the number of items in the collection.

class `philo.models.collections.`**`CollectionMember`**(**args*, ***kwargs*)
The collection member model represents a generic link from a Collection to an arbitrary model instance with an attached order.

objects
A CollectionMemberManager instance

collection
`ForeignKey` to a Collection instance.

index
The numerical index of the item within the collection (optional).

member
`GenericForeignKey` to an arbitrary model instance.

class `philo.models.collections.`**`CollectionMemberManager`**

with_model(*model*)
Given a model class or instance, returns a queryset of all instances of that model which have collection members in this manager's scope.

Example:

```
>>> from philo.models import Collection
>>> from django.contrib.auth.models import User
>>> collection = Collection.objects.get(name="Foo")
>>> collection.members.all()
[<CollectionMember: Foo - user1>, <CollectionMember: Foo - user2>, <CollectionMember: Foo -
>>> collection.members.with_model(User)
[<User: user1>, <User: user2>]
```

1.3.4 Miscellaneous Models

class `philo.models.nodes.`**`TargetURLModel`**(**args*, ***kwargs*)

An abstract parent class for models which deal in targeting a url.

`target_node`

An optional `ForeignKey` to a `Node`. If provided, that node will be used as the basis for the redirect.

`url_or_subpath`

A `CharField` which may contain an absolute or relative URL, or the name of a node's subpath.

`reversing_parameters`

A `JSONField` instance. If the value of `reversing_parameters` is not None, the `url_or_subpath` will be treated as the name of a view to be reversed. The value of `reversing_parameters` will be passed into the reversal as args if it is a list or as kwargs if it is a dictionary. Otherwise it will be ignored.

`target_url`

Calculates and returns the target url based on the `target_node`, `url_or_subpath`, and `reversing_parameters`. The results will be memoized by default; this can be prevented by passing in `memoize=False`.

1.3.5 Custom Fields

class `philo.models.fields.`**`JSONField`**

A `TextField` which stores its value on the model instance as a python object and stores its value in the database as JSON. Validated with `json_validator()`.

class `philo.models.fields.`**`SlugMultipleChoiceField`**

Stores a selection of multiple items with unique slugs in the form of a comma-separated list. Also knows how to correctly handle `RegistryIterators` passed in as choices.

class `philo.models.fields.`**`TemplateField`**(*allow=None*, *disallow=None*, *secure=True*, **args*, ***kwargs*)

A `TextField` which is validated with a `TemplateValidator`. `allow`, `disallow`, and `secure` will be passed into the validator's construction.

AttributeProxyFields

class `philo.models.fields.entities.`**`AttributeProxyField`**(*attribute_key=None*, *verbose_name=None*, *help_text=None*, *default=NOT_PROVIDED*, *editable=True*, *choices=None*, **args*, ***kwargs*)

`AttributeProxyField`s can be assigned as fields on a subclass of `philo.models.base.Entity`. They act like any other model fields, but instead of saving their data to the model's table, they save it to `Attribute`s related to a model instance. Additionally, a new `Attribute` will be created for an instance if and only if the field's value has been set. This is relevant i.e. for `PassthroughAttributeMapper`s and `TreeAttributeMapper`s, where even an `Attribute` with a value of `None` will prevent a passthrough.

Example:

```
class Thing(Entity):
        numbers = models.PositiveIntegerField()
        improvised = JSONAttribute(models.BooleanField)
```

Parameters **attribute_key** – The key of the attribute that will be used to store this field's value, if it is different than the field's name.

The remaining parameters have the same meaning as for ordinary model fields.

`formfield`(*form_class=<class 'django.forms.fields.CharField'>*, ***kwargs*)
Returns a form field capable of accepting values for the `AttributeProxyField`.

`value_from_object`(*obj*)
Returns the value of this field in the given model instance.

`get_storage_value`(*value*)
Final conversion of `value` before it gets stored on an `Entity` instance. This will be called during `EntityForm.save()`.

`validate_value`(*value*)
Raise an appropriate exception if `value` is not valid for this `AttributeProxyField`.

`has_default`()
Returns `True` if a default value was provided and `False` otherwise.

`choices`
Returns the choices passed into the constructor.

`value_class`
Each `AttributeProxyField` subclass can define a value_class to use for creation of new `AttributeValue`s

class `philo.models.fields.entities.`**`JSONAttribute`**(*field_template=None*, ***kwargs*)
Handles an `Attribute` with a `JSONValue`.

Parameters **field_template** – A django form field instance that will be used to guide rendering and interpret values. For example, using `django.forms.BooleanField` will make this field render as a checkbox.

`value_class`
alias of `JSONValue`

`value_from_object`(*obj*)
If the field template is a `DateField` or a `DateTimeField`, this will convert the default return value to a datetime instance.

`get_storage_value`(*value*)
If `value` is a `datetime.datetime` instance, this will convert it to a format which can be stored as correct JSON.

class `philo.models.fields.entities.`**`ForeignKeyAttribute`**(*model*, *limit_choices_to=None*, ***kwargs*)
Handles an `Attribute` with a `ForeignKeyValue`.

Parameters **limit_choices_to** – A `Q` object, dictionary, or `ContentTypeLimiter` to restrict the queryset for the `ForeignKeyAttribute`.

`value_class`
alias of `ForeignKeyValue`

`value_from_object`(*obj*)
Converts the default value type (a model instance) to a pk.

class `philo.models.fields.entities.`**`ManyToManyAttribute`**(*model*, *limit_choices_to=None*, ***kwargs*)
Handles an `Attribute` with a `ManyToManyValue`.

Parameters **limit_choices_to** – A `Q` object, dictionary, or `ContentTypeLimiter` to restrict the queryset for the `ManyToManyAttribute`.

`value_class`
alias of `ManyToManyValue`

`value_from_object`(*obj*)
Converts the default value type (a queryset) to a list of pks.

1.4 Exceptions

`philo.exceptions.`**`MIDDLEWARE_NOT_CONFIGURED`**
Raised if `request.node` is required but not present. For example, this can be raised by `philo.views.node_view()`. `MIDDLEWARE_NOT_CONFIGURED` is an instance of `django.core.exceptions.ImproperlyConfigured`.

exception `philo.exceptions.`**`ViewDoesNotProvideSubpaths`**
Raised by `View.reverse()` when the `View` does not provide subpaths (the default).

exception `philo.exceptions.`**`ViewCanNotProvideSubpath`**
Raised by `View.reverse()` when the `View` can not provide a subpath for the supplied arguments.

exception `philo.exceptions.`**`AncestorDoesNotExist`**
Raised by `TreeEntity.get_path()` if the root instance is not an ancestor of the current instance.

1.5 Handling Requests

`philo.middleware.`**`get_node`**(*path*)
Returns a `Node` instance at `path` (relative to the current site) or `None`.

class `philo.middleware.`**`RequestNodeMiddleware`**
Adds a `node` attribute, representing the currently-viewed `Node`, to every incoming `HttpRequest` object. This is required by `philo.views.node_view()`.

`RequestNodeMiddleware` also catches all exceptions raised while handling requests that have attached `Node`s if `settings.DEBUG` is `True`. If a `django.http.Http404` error was caught, `RequestNodeMiddleware` will look for an "Http404" `Attribute` on the request's `Node`; otherwise it will look for an "Http500" `Attribute`. If an appropriate `Attribute` is found, and the value of the attribute is a `View` instance, then the `View` will be rendered with the exception in the `extra_context`, bypassing any later handling of exceptions.

`philo.views.`**`node_view`**(*request*[, *path=None*, ***kwargs*])
`node_view()` handles incoming requests by checking to make sure that:

- the request has an attached `Node`.
- the attached `Node` handles any remaining path beyond its location.

If these conditions are not met, then `node_view()` will either raise `Http404` or, if it seems like the address was mistyped (for example missing a trailing slash), return an `HttpResponseRedirect` to the correct address.

Otherwise, `node_view()` will call the `Node`'s `render_to_response()` method, passing `kwargs` in as the `extra_context`.

1.6 Signals

`philo.signals.`**`entity_class_prepared`**

Sent whenever an Entity subclass has been "prepared" – that is, after the processing necessary to make `AttributeProxyFields` work has been completed. This will fire after `django.db.models.signals.class_prepared`.

Arguments that are sent with this signal:

`sender` The model class.

`philo.signals.`**`view_about_to_render`**

Sent when a `View` instance is about to render. This allows you, for example, to modify the `extra_context` dictionary used in rendering.

Arguments that are sent with this signal:

`sender` The `View` instance

`request` The `HttpRequest` instance which the `View` is rendering in response to.

`extra_context` A dictionary which will be passed into `actually_render_to_response()`.

`philo.signals.`**`view_finished_rendering`**

Sent when a view instance has finished rendering.

Arguments that are sent with this signal:

`sender` The `View` instance

`response` The `HttpResponse` instance which `View` view has rendered to.

`philo.signals.`**`page_about_to_render_to_string`**

Sent when a `Page` instance is about to render as a string. If the `Page` is rendering as a response, this signal is sent after `view_about_to_render` and serves a similar function. However, there are situations where a `Page` may be rendered as a string without being rendered as a response afterwards.

Arguments that are sent with this signal:

`sender` The `Page` instance

`request` The `HttpRequest` instance which the `Page` is rendering in response to (if any).

`extra_context` A dictionary which will be passed into the `Template` context.

`philo.signals.`**`page_finished_rendering_to_string`**

Sent when a `Page` instance has just finished rendering as a string. If the `Page` is rendering as a response, this signal is sent before `view_finished_rendering` and serves a similar function. However, there are situations where a `Page` may be rendered as a string without being rendered as a response afterwards.

Arguments that are sent with this signal:

`sender` The `Page` instance

`string` The string which the `Page` has rendered to.

1.7 Validators

`philo.validators.`**`INSECURE_TAGS`**

Tags which are considered insecure and are therefore always disallowed by secure `TemplateValidator` instances.

`philo.validators.`**`json_validator`**(*value*)
Validates whether `value` is a valid json string.

class `philo.validators.`**`TemplateValidator`**(*allow=None*, *disallow=None*, *secure=True*)
Validates whether a string represents valid Django template code.

Parameters

- **allow** – `None` or an iterable of tag names which are explicitly allowed. If provided, tags whose names are not in the iterable will cause a ValidationError to be raised if they are used in the template code.
- **disallow** – `None` or an iterable of tag names which are explicitly allowed. If provided, tags whose names are in the iterable will cause a ValidationError to be raised if they are used in the template code. If a tag's name is in `allow` and `disallow`, it will be disallowed.
- **secure** – If the validator is set to secure, it will automatically disallow the tag names listed in `INSECURE_TAGS`. Defaults to `True`.

1.8 Utilities

`philo.utils.`**`fattr`**(**args*, ***kwargs*)
Returns a wrapper which takes a function as its only argument and sets the key/value pairs passed in with kwargs as attributes on that function. This can be used as a decorator.

Example:

```
>>> from philo.utils import fattr
>>> @fattr(short_description="Hello World!")
... def x():
...     pass
...
>>> x.short_description
'Hello World!'
```

class `philo.utils.`**`ContentTypeRegistryLimiter`**
Can be used to limit the choices for a `ForeignKey` or `ManyToManyField` to the `ContentTypes` which have been registered with this limiter.

`register_class`(*cls*)
Registers a model class with this limiter.

`unregister_class`(*cls*)
Unregisters a model class from this limiter.

class `philo.utils.`**`ContentTypeSubclassLimiter`**(*cls*, *inclusive=False*)
Can be used to limit the choices for a `ForeignKey` or `ManyToManyField` to the `ContentTypes` for all non-abstract models which subclass the class passed in on instantiation.

Parameters

- **cls** – The class whose non-abstract subclasses will be valid choices.
- **inclusive** – Whether `cls` should also be considered a valid choice (if it is a non-abstract subclass of `models.Model`)

`philo.utils.`**`paginate`**(*objects*, *per_page=None*, *page_number=1*)
Given a list of objects, return a (`paginator`, `page`, `objects`) tuple.

Parameters

- **objects** – The list of objects to be paginated.
- **per_page** – The number of objects per page.
- **page_number** – The number of the current page.

Returns tuple (`paginator, page, objects`) where `paginator` is a `django.core.paginator.Paginator` instance, `page` is the result of calling `Paginator.page()` with `page_number`, and objects is `page.objects`. Any of the return values which can't be calculated will be returned as `None`.

1.8.1 AttributeMappers

class `philo.utils.entities.`**`AttributeMapper`**(*entity*)

Given an `Entity` subclass instance, this class allows dictionary-style access to the `Entity`'s `Attribute`s. In order to prevent unnecessary queries, the `AttributeMapper` will cache all `Attribute`s and the associated python values when it is first accessed.

Parameters entity – The `Entity` subclass instance whose `Attribute`s will be made accessible.

`get_attributes`()

Returns an iterable of all of the `Entity`'s `Attribute`s.

`get_attribute`(*key*, *default=None*)

Returns the `Attribute` instance with the given `key` from the cache, populating the cache if necessary, or `default` if no such attribute is found.

`keys`()

Returns the keys from the cache, first populating the cache if necessary.

`items`()

Returns the items from the cache, first populating the cache if necessary.

`values`()

Returns the values from the cache, first populating the cache if necessary.

`clear_cache`()

Clears the cache.

class `philo.utils.entities.`**`TreeAttributeMapper`**(*entity*)

Bases: `philo.utils.entities.AttributeMapper`

The `TreeEntity` class allows the inheritance of `Attribute`s down the tree. This mapper will return the most recently declared `Attribute` among the `TreeEntity`'s ancestors or set an attribute on the `Entity` it is attached to.

`get_attributes`()

Returns a list of `Attribute`s sorted by increasing parent level. When used to populate the cache, this will cause `Attribute`s on the root to be overwritten by those on its children, etc.

class `philo.utils.entities.`**`PassthroughAttributeMapper`**(*entities*)

Bases: `philo.utils.entities.AttributeMapper`

Given an iterable of `Entities`, this mapper will fetch an `AttributeMapper` for each one. Lookups will return the value from the first `AttributeMapper` which has an entry for a given key. Assignments will be made to the first `Entity` in the iterable.

Parameters entities – An iterable of `Entity` subclass instances.

LazyAttributeMappers

class `philo.utils.entities.`**`LazyAttributeMapperMixin`**

In some cases, it may be that only one attribute value needs to be fetched. In this case, it is more efficient to avoid populating the cache whenever possible. This mixin overrides the `__getitem__()` and `get_attribute()` methods to prevent their populating the cache. If the cache has been populated (i.e. through `keys()`, `values()`, etc.), then the value or attribute will simply be returned from the cache.

class `philo.utils.entities.`**`LazyAttributeMapper`**(*entity*)

Bases: `philo.utils.entities.LazyAttributeMapperMixin`, `philo.utils.entities.AttributeMapper`

class `philo.utils.entities.`**`LazyTreeAttributeMapper`**(*entity*)

Bases: `philo.utils.entities.LazyAttributeMapperMixin`, `philo.utils.entities.TreeAttributeMapper`

class `philo.utils.entities.`**`LazyPassthroughAttributeMapper`**(*entities*)

Bases: `philo.utils.entities.LazyAttributeMapperMixin`, `philo.utils.entities.PassthroughAttributeMapper`

The `LazyPassthroughAttributeMapper` is lazy in that it tries to avoid accessing the `AttributeMapper`s that it uses for lookups. However, those `AttributeMapper`s may or may not be lazy themselves.

1.9 Template Tags

1.9.1 Collections

The collection template tags are automatically included as builtins if `philo` is an installed app.

templatetag `collections.`**`membersof`**

Given a collection and a content type, sets the results of `collection.members.with_model` as a variable in the context.

Usage:

```
{% membersof <collection> with <app_label>.<model_name> as <var> %}
```

1.9.2 Containers

The container template tags are automatically included as builtins if `philo` is an installed app.

templatetag `containers.`**`container`**

If a template using this tag is used to render a `Page`, that `Page` will have associated content which can be set in the admin interface. If a content type is referenced, then a `ContentReference` object will be created; otherwise, a `Contentlet` object will be created.

Usage:

```
{% container <name> [[references <app_label>.<model_name>] as <variable>] %}
```

1.9.3 Embedding

The embed template tags are automatically included as builtins if `philo` is an installed app.

templatetag embed.**embed**

The {% embed %} tag can be used in two ways.

First, to set which template will be used to render a particular model. This declaration can be placed in a base template and will propagate into all templates that extend that template.

Syntax:

```
{% embed <app_label>.<model_name> with <template> %}
```

Second, to embed a specific model instance in the document with a template specified earlier in the template or in a parent template using the first syntax. The instance can be specified as a content type and pk or as a context variable. Any kwargs provided will be passed into the context of the template.

Syntax:

```
{% embed (<app_label>.<model_name> <object_pk> || <instance>) [<argname>=<value> ...] %}
```

1.9.4 Nodes

The node template tags are automatically included as builtins if `philo` is an installed app.

templatetag nodes.**node_url**

The `node_url` tag allows access to `View.reverse()` from a template for a `Node`. By default, the `Node` that is used for the call is pulled from the context variable `node`; however, this can be overridden with the `[for <node>]` option.

Usage:

```
{% node_url [for <node>] [as <var>] %}
{% node_url with <obj> [for <node>] [as <var>] %}
{% node_url <view_name> [<arg1> [<arg2> ...] ] [for <node>] [as <var>] %}
{% node_url <view_name> [<key1>=<value1> [<key2>=<value2> ...] ] [for <node>] [as <var>] %}
```

1.9.5 String inclusion

templatetag include_string.**include_string**

Include a flat string by interpreting it as a template. The compiled template will be rendered with the current context.

Usage:

```
{% include_string <template_code> %}
```

1.10 Forms

class philo.forms.entities.**EntityForm**(**args*, ***kwargs*)

`EntityForm` knows how to handle `Entity` instances - specifically, how to set initial values for `AttributeProxyField`s and save cleaned values to an instance on save.

1.10.1 Fields

class `philo.forms.fields.`**`JSONFormField`**(*required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, *error_messages=None*, *show_hidden_initial=False*, *validators=[ ]*, *localize=False*)

A form field which is validated by `philo.validators.json_validator()`.

1.11 Database Template Loader

class `philo.loaders.database.`**`Loader`**(**args*, ***kwargs*)

`philo.loaders.database.Loader` enables loading of template code from `Template`s. This would let `Template`s be used with `{% include %}` and `{% extends %}` tags, as well as any other features that use template loading.

1.12 Contrib apps

1.12.1 Penfield

Blogs

Newsletters

Template filters

Penfield supplies two template filters to handle common use cases for blogs and newsletters.

templatefilter `penfield.`**`monthname`**(*value*)

Returns the name of a month with the supplied numeric value.

templatefilter `penfield.`**`apmonthname`**(*value*)

Returns the Associated Press abbreviated month name for the supplied numeric value.

1.12.2 Shipherd

`Node`s are useful for structuring a website; however, they are inherently unsuitable for creating site navigation.

The most glaring problem is that a navigation tree based on `Node`s would have one `Node` as the root, whereas navigation usually has multiple objects at the top level.

Additionally, navigation needs to have display text that is relevant to the current context; however, `Node`s do not have a field for that, and `View` subclasses with a name or title field will generally need to use it for database-searchable names.

Finally, `Node` structures are inherently unordered, while navigation is inherently ordered.

`shipherd` exists to resolve these issues by separating navigation structures from `Node` structures. It is instead structured around the way that site navigation works in the wild:

- A site may have one or more independent navigation bars (Main navigation, side navigation, etc.)
- A navigation bar may be shared by sections of the website, or even by the entire site.
- A navigation bar has a certain depth that it displays to.

The `Navigation` model supplies these features by attaching itself to a `Node` via `ForeignKey` and adding a `navigation` property to `Node` which provides access to a `Node` instance's inherited `Navigation`s.

Each entry in the navigation bar is then represented by a `NavigationItem`, which stores information such as the `order` and `text` for the entry. Given an `HttpRequest`, a `NavigationItem` can also tell whether it `is_active()` or `has_active_descendants()`.

Since the common pattern is to recurse through a navigation tree and render each part similarly, `shipherd` also ships with the `recursenavigation` template tag.

Models

class `philo.contrib.shipherd.models.`**`NavigationMapper`**`(`*node*`)`

Bases: `object`, `UserDict.DictMixin`

The `NavigationMapper` is a dictionary-like object which allows easy fetching of the root items of a navigation for a node according to a key. A `NavigationMapper` instance will be available on each node instance as `Node.navigation` if `shipherd` is in the `INSTALLED_APPS`

class `philo.contrib.shipherd.models.`**`Navigation`**`(`**args*, ***kwargs*`)`

Bases: `philo.models.base.Entity`

`Navigation` represents a group of `NavigationItem`s that have an intrinsic relationship in terms of navigating a website. For example, a `main` navigation versus a `side` navigation, or a `authenticated` navigation versus an `anonymous` navigation.

A `Navigation`'s `NavigationItem`s will be accessible from its related `Node` and that `Node`'s descendants through a `NavigationMapper` instance at `Node.navigation`. Example:

```
>>> node.navigation_set.all()
[]
>>> parent = node.parent
>>> items = parent.navigation_set.get(key='main').roots.all()
>>> parent.navigation["main"] == node.navigation["main"] == list(items)
True
```

`objects`

A `NavigationManager` instance.

`node`

The `Node` which the `Navigation` is attached to. The `Navigation` will also be available to all the `Node`'s descendants and will override any `Navigation` with the same key on any of the `Node`'s ancestors.

`key`

Each `Navigation` has a `key` which consists of one or more word characters so that it can easily be accessed in a template as `{{ node.navigation.this_key }}`.

`depth`

There is no limit to the depth of a tree of `NavigationItem`s, but `depth` will limit how much of the tree will be displayed.

class `philo.contrib.shipherd.models.`**`NavigationItem`**`(`**args*, ***kwargs*`)`

Bases: `philo.models.base.TreeEntity`, `philo.models.nodes.TargetURLModel`

NavigationItem(id, parent_id, lft, rght, tree_id, level, target_node_id, url_or_subpath, reversing_parameters_json, navigation_id, text, order)

navigation
A ForeignKey to a Navigation instance. If this is not null, then the NavigationItem will be a root node of the Navigation instance.

text
The text which will be displayed in the navigation. This is a CharField instance with max length 50.

order
The order in which the NavigationItem will be displayed.

is_active (*request*)
Returns True if the NavigationItem is considered active for a given request and False otherwise.

has_active_descendants (*request*)
Returns True if the NavigationItem has active descendants and False otherwise.

class philo.contrib.shipherd.models.**NavigationManager**

Template tags

templatetag shipherd.**recursenavigation**
The recursenavigation templatetag takes two arguments:

- the Node for which the Navigation should be found
- the Navigation's key.

It will then recursively loop over each NavigationItem in the Navigation and render the template chunk within the block. recursenavigation sets the following variables in the context:

Variable	Description
navloop.depth	The current depth of the loop (1 is the top level)
navloop.depth0	The current depth of the loop (0 is the top level)
navloop.counter	The current iteration of the current level(1-indexed)
navloop.counter0	The current iteration of the current level(0-indexed)
navloop.first	True if this is the first time through the current level
navloop.last	True if this is the last time through the current level
navloop.parentloop	This is the loop one level "above" the current one
item	The current item in the loop (a NavigationItem instance)
children	If accessed, performs the next level of recursion.
navloop.active	True if the item is active for this request
navloop.active_descendants	True if the item has active descendants for this request

Example:

```
<ul>
    {% recursenavigation node "main" %}
        <li{% if navloop.active %} class='active'{% endif %}>
            <a href="{{ item.get_target_url }}">{{ item.text }}</a>
            {% if item.get_children %}
                <ul>
                    {{ children }}
                </ul>
            {% endif %}
        </li>
    {% endrecursenavigation %}
</ul>
```

Note: `{% recursenavigation %}` requires that the current `HttpRequest` be present in the context as `request`. The simplest way to do this is with the request context processor. Simply make sure that `django.core.context_processors.request` is included in your `TEMPLATE_CONTEXT_PROCESSORS` setting.

templatefilter `shipherd.`**`has_navigation`**(*node*, *key=None*)
Returns `True` if the node has a `Navigation` with the given key and `False` otherwise. If `key` is `None`, returns whether the node has any `Navigation`s at all.

templatefilter `shipherd.`**`navigation_host`**(*node*, *key*)
Returns the `Node` which hosts the `Navigation` which `node` has inherited for `key`. Returns `node` if any exceptions are encountered.

1.12.3 Sobol

Sobol implements a generic search interface, which can be used to search databases or websites. No assumptions are made about the search method. If SOBOL_USE_CACHE is `True` (default), the results will be cached using django's cache framework. Be aware that this may use a large number of cache entries, as a unique entry will be made for each search string for each type of search.

Settings

`SOBOL_USE_CACHE` Whether sobol will use django's cache framework. Defaults to `True`; this may cause a lot of entries in the cache.

`SOBOL_USE_EVENTLET` If `eventlet` is installed and this setting is `True`, sobol web searches will use `eventlet.green.urllib2` instead of the built-in `urllib2` module. Default: `False`.

Templates

For convenience, `sobol` provides a template at `sobol/search/_list.html` which can be used with an `{% include %}` tag inside a full search page template to list the search results. The `_list.html` template also uses a basic jQuery script (`static/sobol/ajax_search.js`) to handle AJAX search result loading if the AJAX API of the current `SearchView` is enabled. If you want to use `_list.html`, but want to provide your own version of jQuery or your own AJAX loading script, or if you want to include the basic script somewhere else (like inside the `<head>`) simply do the following:

```
{% include "sobol/search/_list.html" with suppress_scripts=1 %}
```

Models

class `philo.contrib.sobol.models.`**`Search`**(**args*, ***kwargs*)
Represents all attempts to search for a unique string.

`string`
The string which was searched for.

`get_weighted_results`(*threshhold=None*)
Returns a list of `ResultURL` instances related to the search and ordered by decreasing weight. This will be cached on the instance.

Parameters threshhold – The earliest datetime that a `Click` can have been made on a related `ResultURL` in order to be included in the weighted results (or `None` to include all `Click`s and `ResultURL`s).

`get_favored_results`(*error=5*, *threshhold=None*)
Calculates the set of most-favored results based on their weight. Evenly-weighted results will be grouped together and either added or excluded as a group.

Parameters

- **error** – An arbitrary number; higher values will cause this method to be more reticent about adding new items to the favored results.
- **threshhold** – Will be passed directly into `get_weighted_results()`

class `philo.contrib.sobol.models.`**`ResultURL`**(**args*, ***kwargs*)
Represents a URL which has been selected one or more times for a `Search`.

`search`
A `ForeignKey` to the `Search` which the `ResultURL` is related to.

`url`
The URL which was selected.

`get_weight`(*threshhold=None*)
Calculates, caches, and returns the weight of the `ResultURL`.

Parameters threshhold – The datetime limit before which `Click`s will not contribute to the weight of the `ResultURL`.

`weight`
Calculates, caches, and returns the weight of the `ResultURL`.

Parameters threshhold – The datetime limit before which `Click`s will not contribute to the weight of the `ResultURL`.

class `philo.contrib.sobol.models.`**`Click`**(**args*, ***kwargs*)
Represents a click on a `ResultURL`.

`result`
A `ForeignKey` to the `ResultURL` which the `Click` is related to.

`datetime`
The datetime when the click was registered in the system.

`get_weight`(*default=1*, *weighted=<function <lambda> at 0x31f31b8>*)
Calculates and returns the weight of the `Click`.

`weight`
Calculates and returns the weight of the `Click`.

class `philo.contrib.sobol.models.`**`SearchView`**(**args*, ***kwargs*)
Handles a view for the results of a search, anonymously tracks the selections made by end users, and provides an AJAX API for asynchronous search result loading. This can be particularly useful if some searches are slow.

`results_page`
`ForeignKey` to a `Page` which will be used to render the search results.

`searches`
A `SlugMultipleChoiceField` whose choices are the contents of `sobol.search.registry`

`enable_ajax_api`
A `BooleanField` which controls whether or not the AJAX API is enabled.

Note: If the AJAX API is enabled, a `ajax_api_url` attribute will be added to each search instance containing the url and get parameters for an AJAX request to retrieve results for that search.

Note: Be careful not to access `search_instance.results` if the AJAX API is enabled - otherwise the search will be run immediately rather than on the AJAX request.

`placeholder_text`
A `CharField` containing the placeholder text which is intended to be used for the search box for the `SearchView`. It is the template author's responsibility to make use of this information.

`search_form`
The form which will be used to validate the input to the search box for this `SearchView`.

`results_view`(*request*, *extra_context=None*)
Renders `results_page` with a context containing an instance of `search_form`. If the form was submitted and was valid, then one of two things has happened:

- A search has been initiated. In this case, a list of search instances will be added to the context as `searches`. If `enable_ajax_api` is enabled, each instance will have an `ajax_api_url` attribute containing the url needed to make an AJAX request for the search results.
- A link has been chosen. In this case, corresponding `Search`, `ResultURL`, and `Click` instances will be created and the user will be redirected to the link's actual url.

`ajax_api_view`(*request*, *slug*, *extra_context=None*)
Returns a JSON object containing the following variables:

search Contains the slug for the search.

results Contains the results of `Result.get_context()` for each result.

rendered Contains the results of `Result.render()` for each result.

hasMoreResults `True` or `False` whether the search has more results according to `BaseSearch.has_more_results()`

moreResultsURL Contains `None` or a querystring which, once accessed, will note the `Click` and redirect the user to a page containing more results.

Search API

`philo.contrib.sobol.search.`**`registry`**
A registry for `BaseSearch` subclasses that should be available in the admin.

`philo.contrib.sobol.search.`**`get_search_instance`**(*slug*, *search_arg*)
Returns a search instance for the given slug, either from the cache or newly-instantiated.

class `philo.contrib.sobol.search.`**`Result`**(*search*, *result*)
`Result` is a helper class that, given a search and a result of that search, is able to correctly render itself with a template defined by the search. Every `Result` will pass a `title`, a `url` (if applicable), and the raw `result` returned by the search into the template context when rendering.

Parameters

- **search** – An instance of a `BaseSearch` subclass or an object that implements the same API.
- **result** – An arbitrary result from the `search`.

get_title()
Returns the title of the result by calling `BaseSearch.get_result_title()` on the raw result.

get_url()
Returns the url of the result or `None` by calling `BaseSearch.get_result_url()` on the raw result. This url will contain a querystring which, if used, will track a `Click` for the actual url.

get_actual_url()
Returns the actual url of the result by calling `BaseSearch.get_actual_result_url()` on the raw result.

get_content()
Returns the content of the result by calling `BaseSearch.get_result_content()` on the raw result.

get_template()
Returns the template which will be used to render the `Result` by calling `BaseSearch.get_result_template()` on the raw result.

get_context()
Returns the context dictionary for the result. This is used both in rendering the result and in the AJAX return value for `SearchView.ajax_api_view()`. The context will contain the following keys:

title The result of calling `get_title()`

url The result of calling `get_url()`

content The result of calling `get_content()`

render()
Returns the template from `get_template()` rendered with the context from `get_context()`.

class `philo.contrib.sobol.search.`**BaseSearch**(*search_arg*)
Defines a generic search api. Accessing `results` will attempt to retrieve cached results and, if that fails, will initiate a new search and store the results in the cache. Each search has a `verbose_name` and a `slug`. If these are not provided as attributes, they will be automatically generated based on the name of the class.

Parameters **search_arg** – The string which is being searched for.

result_limit
The number of results to return from the complete list. Default: 5

result_template
The path to the template which will be used to render the `Result`s for this search. If this is `None`, then the framework will try `sobol/search/<slug>/result.html` and `sobol/search/result.html`.

title_template
The path to the template which will be used to generate the title of the `Result`s for this search. If this is `None`, then the framework will try `sobol/search/<slug>/title.html` and `sobol/search/title.html`.

content_template
The path to the template which will be used to generate the content of the `Result`s for this search. If this is `None`, then the framework will try `sobol/search/<slug>/content.html` and `sobol/search/content.html`.

results
Retrieves cached results or initiates a new search via `get_results()` and caches the results.

get_results(*limit=None*, *result_class=<class 'philo.contrib.sobol.search.Result'>*)
Calls `search()` and parses the return value into `Result` instances.

Parameters

- **limit** – Passed directly to `search()`.
- **result_class** – The class used to represent the results. This will be instantiated with the `BaseSearch` instance and the raw result from the search.

search(*limit=None*)
Returns an iterable of up to `limit` results. The `get_result_title()`, `get_result_url()`, `get_result_template()`, and `get_result_extra_context()` methods will be used to interpret the individual items that this function returns, so the result can be an object with attributes as easily as a dictionary with keys. However, keep in mind that the raw results will be stored with django's caching mechanisms and will be converted to JSON.

get_actual_result_url(*result*)
Returns the actual URL for the `result` or `None` if there is no URL. Must be implemented by subclasses.

get_result_querydict(*result*)
Returns a querydict for tracking selection of the result, or `None` if there is no URL for the result.

get_result_url(*result*)
Returns `None` or a url which, when accessed, will register a `Click` for that url.

get_result_title(*result*)
Returns the title of the `result`. By default, renders `sobol/search/<slug>/title.html` or `sobol/search/title.html` with the result in the context. This can be overridden by setting `title_template` or simply overriding `get_result_title()`. If no template can be found, this will raise `TemplateDoesNotExist`.

get_result_content(*result*)
Returns the content for the `result`. By default, renders `sobol/search/<slug>/content.html` or `sobol/search/content.html` with the result in the context. This can be overridden by setting `content_template` or simply overriding `get_result_content()`. If no template is found, this will return an empty string.

get_result_template(*result*)
Returns the template to be used for rendering the `result`. For a search with slug `google`, this would first try `sobol/search/google/result.html`, then fall back on `sobol/search/result.html`. Subclasses can override this by setting `result_template` to the path of another template.

has_more_results
Returns `True` if there are more results than `result_limit` and `False` otherwise.

get_actual_more_results_url()
Returns the actual url for more results. By default, simply returns `None`.

get_more_results_querydict()
Returns a `QueryDict` for tracking whether people click on a 'more results' link.

more_results_url
Returns a URL which consists of a querystring which, when accessed, will log a `Click` for the actual URL.

class `philo.contrib.sobol.search.`**DatabaseSearch**(*search_arg*)
Implements `search()` and `get_queryset()` methods to handle database queries.

model
The model which should be searched by the `DatabaseSearch`.

get_queryset()
Returns a `QuerySet` of all instances of `model`. This method should be overridden by subclasses to specify how the search should actually be implemented for the model.

class `philo.contrib.sobol.search.`**`URLSearch`**(*search_arg*)

Defines a generic interface for searches that require accessing a certain url to get search results.

`search_url`

The base URL which will be accessed to get the search results.

`query_format_str`

The url-encoded query string to be used for fetching search results from `search_url`. Must have one `%s` to contain the search argument.

`url`

The URL where the search gets its results. Composed from `search_url` and `query_format_str`.

`parse_response`(*response*, *limit=None*)

Handles the `response` from accessing `url` (with `urllib2.urlopen()`) and returns a list of up to `limit` results.

class `philo.contrib.sobol.search.`**`JSONSearch`**(*search_arg*)

Makes a GET request and parses the results as JSON. The default behavior assumes that the response contains a list of results.

class `philo.contrib.sobol.search.`**`GoogleSearch`**(*search_arg*)

An example implementation of a `JSONSearch`.

`default_args`

Unquoted default arguments for the `GoogleSearch`.

1.12.4 Waldo

Models

Waldo provides abstract `MultiView`s to handle several levels of common authentication:

- `LoginMultiView` handles the case where users only need to be able to log in and out.
- `PasswordMultiView` handles the case where users will also need to change their password.
- `RegistrationMultiView` builds on top of `PasswordMultiView` to handle user registration, as well.
- `AccountMultiView` adds account-handling functionality to the `RegistrationMultiView`.

class `philo.contrib.waldo.models.`**`LoginMultiView`**(**args*, ***kwargs*)

Handles exclusively methods and views related to logging users in and out.

`login_page`

A `ForeignKey` to the `Page` which will be used to render the login form.

`login_form`

A django form class which will be used for the authentication process. Default: `WaldoAuthenticationForm`.

`set_requirement_redirect`(*request*, *redirect=None*)

Figures out and stores where a user should end up after landing on a page (like the login page) because they have not fulfilled some kind of requirement.

`get_requirement_redirect`(*request*, *default=None*)

Returns the location which a user should be redirected to after fulfilling a requirement (like logging in).

`login`(*request*, **args*, ***kwargs*)

Renders the `login_page` with an instance of the `login_form` for the given `HttpRequest`.

`logout`(*request*, **args*, ***kwargs*)
Logs the given `HttpRequest` out, redirecting the user to the page they just left or to the `get_absolute_url()` for the `request.node`.

`login_required`(*view*)
Wraps a view function to require that the user be logged in.

class `philo.contrib.waldo.models.`**`PasswordMultiView`**(**args*, ***kwargs*)
Adds support for password setting, resetting, and changing to the `LoginMultiView`. Password reset support includes handling of a confirmation email.

`password_reset_page`
A `ForeignKey` to the `Page` which will be used to render the password reset request form.

`password_reset_confirmation_email`
A `ForeignKey` to the `Page` which will be used to render the password reset confirmation email.

`password_set_page`
A `ForeignKey` to the `Page` which will be used to render the password setting form (i.e. the page that users will see after confirming a password reset).

`password_change_page`
A `ForeignKey` to the `Page` which will be used to render the password change form.

`password_change_form`
The password change form class. Default: `django.contrib.auth.forms.PasswordChangeForm`.

`password_set_form`
The password set form class. Default: `django.contrib.auth.forms.SetPasswordForm`.

`password_reset_form`
The password reset request form class. Default: `django.contrib.auth.forms.PasswordResetForm`.

`make_confirmation_link`(*confirmation_view*, *token_generator*, *user*, *node*, *token_args=None*, *reverse_kwargs=None*, *secure=False*)
Generates a confirmation link for an arbitrary action, such as a password reset.

Parameters

- **confirmation_view** – The view function which needs to be linked to.
- **token_generator** – Generates a confirmable token for the action.
- **user** – The user who is trying to take the action.
- **node** – The node which is providing the basis for the confirmation URL.
- **token_args** – A list of additional arguments (i.e. besides the user) to be used for token creation.
- **reverse_kwargs** – A dictionary of any additional keyword arguments necessary for correctly reversing the view.
- **secure** – Whether the link should use the https:// or http://.

`send_confirmation_email`(*subject*, *email*, *page*, *extra_context*)
Sends a confirmation email for an arbitrary action, such as a password reset. If the `page`'s `Template` has a mimetype of `text/html`, then the email will be sent with an HTML alternative version.

Parameters

- **subject** – The subject line of the email.
- **email** – The recipient's address.
- **page** – The page which will be used to render the email body.

- **extra_context** – The context for rendering the `page`.

`password_reset`(*request*, *extra_context=None*, *token_generator=<django.contrib.auth.tokens.PasswordResetTokenGenerato object at 0x3041350>*)

Handles the process by which users request a password reset, and generates the context for the confirmation email. That context will contain:

link The confirmation link for the password reset.

user The user requesting the reset.

site The current `Site`.

request The current `HttpRequest` instance.

Parameters **token_generator** – The token generator to use for the confirmation link.

`password_reset_confirm`(*request*, *extra_context=None*, *uidb36=None*, *token=None*, *token_generator=<django.contrib.auth.tokens.PasswordResetTokenGenerator object at 0x3041350>*)

Checks that `token`` is valid, and if so, renders an instance of `password_set_form` with `password_set_page`.

Parameters **token_generator** – The token generator used to check the `token`.

`password_change`(*request*, *extra_context=None*)

Renders an instance of `password_change_form` with `password_change_page`.

class `philo.contrib.waldo.models.`**`RegistrationMultiView`**(**args*, ***kwargs*)

Adds support for user registration to the `PasswordMultiView`.

`register_page`

A `ForeignKey` to the `Page` which will be used to display the registration form.

`register_confirmation_email`

A `ForeignKey` to the `Page` which will be used to render the registration confirmation email.

`registration_form`

The registration form class. Default: `RegistrationForm`.

`register`(*request*, *extra_context=None*, *token_generator=<philo.contrib.waldo.tokens.RegistrationTokenGenerator object at 0x41d6b90>*)

Renders the `register_page` with an instance of `registration_form` in the context as `form`. If the form has been submitted, sends a confirmation email using `register_confirmation_email` and the same context as `PasswordMultiView.password_reset()`.

Parameters **token_generator** – The token generator to use for the confirmation link.

`register_confirm`(*request*, *extra_context=None*, *uidb36=None*, *token=None*, *token_generator=<philo.contrib.waldo.tokens.RegistrationTokenGenerator object at 0x41d6b90>*)

Checks that `token` is valid, and if so, logs the user in and redirects them to `post_register_confirm_redirect()`.

Parameters **token_generator** – The token generator used to check the `token`.

`post_register_confirm_redirect`(*request*)

Returns an `HttpResponseRedirect` for post-registration-confirmation. Default: `Node.get_absolute_url()` for `request.node`.

class `philo.contrib.waldo.models.`**`AccountMultiView`**(**args*, ***kwargs*)

Adds support for user accounts on top of the `RegistrationMultiView`. By default, the account consists of the first_name, last_name, and email fields of the User model. Using a different account model is as simple as replacing `account_form` with any form class that takes an `auth.User` instance as the first argument.

`manage_account_page`
A `ForeignKey` to the `Page` which will be used to render the account management form.

`email_change_confirmation_email`
A `ForeignKey` to a `Page` which will be used to render an email change confirmation email. This is optional; if it is left blank, then email changes will be performed without confirmation.

`account_form`
A django form class which will be used to manage the user's account. Default: `UserAccountForm`

`account_view`(*request*, *extra_context=None*, *token_generator=<philo.contrib.waldo.tokens.EmailTokenGenerator object at 0x3d4ec10>*, **args*, ***kwargs*)
Renders the `manage_account_page` with an instance of `account_form` in the context as `form`. If the form has been posted, the user's email was changed, and `email_change_confirmation_email` is not `None`, sends a confirmation email to the new email to make sure it exists before making the change. The email will have the same context as `PasswordMultiView.password_reset()`.

Parameters **token_generator** – The token generator to use for the confirmation link.

`has_valid_account`(*user*)
Returns `True` if the `user` has a valid account and `False` otherwise.

`account_required`(*view*)
Wraps a view function to allow access only to users with valid accounts and otherwise redirect them to the `account_view()`.

`post_register_confirm_redirect`(*request*)
Automatically redirects users to the `account_view()` after registration.

`email_change_confirm`(*request*, *extra_context=None*, *uidb36=None*, *token=None*, *email=None*, *token_generator=<philo.contrib.waldo.tokens.EmailTokenGenerator object at 0x3d4ec10>*)
Checks that `token` is valid, and if so, changes the user's email.

Parameters **token_generator** – The token generator used to check the `token`.

Forms

class `philo.contrib.waldo.forms.`**`EmailInput`**(*attrs=None*)
Displays an HTML5 email input on browsers which support it and a normal text input on other browsers.

class `philo.contrib.waldo.forms.`**`RegistrationForm`**(*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*)
Handles user registration. If `recaptcha_django` is installed on the system and `recaptcha_django.middleware.ReCaptchaMiddleware` is in `settings.MIDDLEWARE_CLASSES`, then a recaptcha field will automatically be added to the registration form.

See Also:

recaptcha-django

email
An EmailField using the EmailInput widget.

class philo.contrib.waldo.forms.**UserAccountForm**(*user*, **args*, ***kwargs*)
Handles a user's account - by default, auth.User.first_name, auth.User.last_name, auth.User.email.

email_changed()
Returns True if the email field changed value and False if it did not, or if there is no email field on the form. This method must be supplied by account forms used with waldo.

reset_email()
ModelForms modify their instances in-place during _post_clean(); this method resets the email value to its initial state and returns the altered value. This is a method on the form to allow unusual behavior such as storing email on a UserProfile.

classmethod set_email(*user*, *email*)
Given a valid instance and an email address, correctly set the email address for that instance and save the changes. This is a class method in order to allow unusual behavior such as storing email on a UserProfile.

class philo.contrib.waldo.forms.**WaldoAuthenticationForm**(*request=None*, **args*, ***kwargs*)
Handles user authentication. Checks that the user has not mistakenly entered their email address (like django.contrib.admin.forms.AdminAuthenticationForm) but does not require that the user be staff.

Token generators

Based on django.contrib.auth.tokens. Supports the following settings:

WALDO_REGISTRATION_TIMEOUT_DAYS The number of days a registration link will be valid before expiring. Default: 1.

WALDO_EMAIL_TIMEOUT_DAYS The number of days an email change link will be valid before expiring. Default: 1.

philo.contrib.waldo.tokens.**registration_token_generator**
Strategy object used to generate and check tokens for the user registration mechanism.

philo.contrib.waldo.tokens.**email_token_generator**
Strategy object used to generate and check tokens for a user email change mechanism.

1.12.5 Winer

Winer provides the same API as django's syndication Feed class, adapted to a Philo-style MultiView for easy database management. Apps that need syndication can simply subclass FeedView, override a few methods, and start serving RSS and Atom feeds. See BlogView for a concrete implementation example.

class philo.contrib.winer.models.**FeedView**(**args*, ***kwargs*)
FeedView is an abstract model which handles a number of pages and related feeds for a single object such as a blog or newsletter. In addition to all other methods and attributes, FeedView supports the same generic API as django.contrib.syndication.views.Feed.

feed_type
The type of feed which should be served by the FeedView.

feed_suffix
The suffix which will be appended to a page URL for a feed_type feed of its items. Default: "feed". Note that RSS and Atom feeds will always be available at <page_url>/rss and <page_url>/atom regardless of the value of this setting.

See Also:

get_feed_type(), feed_patterns()

feeds_enabled
A BooleanField - whether or not feeds are enabled.

feed_length
A PositiveIntegerField - the maximum number of items to return for this feed. All items will be returned if this field is blank. Default: 15.

item_title_template
A ForeignKey to a Template which will be used to render the title of each item in the feed if provided.

item_description_template
A ForeignKey to a Template which will be used to render the description of each item in the feed if provided.

item_context_var
An attribute holding the name of the context variable to be populated with the items managed by the FeedView. Default: "items"

object_attr
An attribute holding the name of the attribute on a subclass of FeedView which will contain the main object of a feed (such as a Blog.) Default: "object"

Example:

```
class BlogView(FeedView):
    blog = models.ForeignKey(Blog)

    object_attr = 'blog'
    item_context_var = 'entries'
```

description
An attribute holding a description of the feeds served by the FeedView. This is a required part of the django.contrib.syndication.view.Feed API.

feed_patterns(*base*, *get_items_attr*, *page_attr*, *reverse_name*)
Given the name to be used to reverse this view and the names of the attributes for the function that fetches the objects, returns patterns suitable for inclusion in urlpatterns. In addition to base (which will serve the page at page_attr) and base + feed_suffix (which will serve a feed_type feed), patterns will be provided for each registered feed type as base + slug.

Parameters

- **base** – The base of the returned patterns - that is, the subpath pattern which will reference the page for the items. The feed_suffix will be appended to this subpath.
- **get_items_attr** – A callable or the name of a callable on the FeedView which will return an (items, extra_context) tuple. This will be passed directly to feed_view() and page_view().
- **page_attr** – A Page instance or the name of an attribute on the FeedView which contains a Page instance. This will be passed directly to page_view() and will be rendered with the items from get_items_attr.

- **reverse_name** – The string which is considered the "name" of the view function returned by `page_view()` for the given parameters.

Returns Patterns suitable for use in urlpatterns.

Example:

```
class BlogView(FeedView):
    blog = models.ForeignKey(Blog)
    entry_archive_page = models.ForeignKey(Page)

    @property
    def urlpatterns(self):
        urlpatterns = self.feed_patterns(r'^', 'get_all_entries', 'index_page', 'index')
        urlpatterns += self.feed_patterns(r'^(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})
        return urlpatterns

    def get_entries_by_ymd(request, year, month, day, extra_context=None):
        entries = Blog.entries.all()
        # filter entries based on the year, month, and day.
        return entries, extra_context
```

See Also:

`get_feed_type()`

`get_object`(*request*, ***kwargs*)
By default, returns the object stored in the attribute named by `object_attr`. This can be overridden for subclasses that publish different data for different URL parameters. It is part of the `django.contrib.syndication.views.Feed` API.

`feed_view`(*get_items_attr*, *reverse_name*, *feed_type=None*)
Returns a view function that renders a list of items as a feed.

Parameters

- **get_items_attr** – A callable or the name of a callable on the `FeedView` that will return a (items, extra_context) tuple when called with the object for the feed and view arguments.
- **reverse_name** – The name which can be used reverse the page for this feed using the `FeedView` as the urlconf.
- **feed_type** – The slug used to render the feed class which will be used by the returned view function.

Returns A view function that renders a list of items as a feed.

`page_view`(*get_items_attr*, *page_attr*)

Parameters

- **get_items_attr** – A callable or the name of a callable on the `FeedView` that will return a (items, extra_context) tuple when called with view arguments.
- **page_attr** – A `Page` instance or the name of an attribute on the `FeedView` which contains a `Page` instance. This will be rendered with the items from `get_items_attr`.

Returns A view function that renders a list of items as an `HttpResponse`.

`process_page_items`(*request*, *items*)
Hook for handling any extra processing of `items` based on an `HttpRequest`, such as pagination or searching. This method is expected to return a list of items and a dictionary to be added to the page context.

get_feed_type(*request*, *feed_type=None*)

If `feed_type` is not `None`, returns the corresponding class from the registry or raises `HttpNotAcceptable`.

Otherwise, intelligently chooses a feed type for a given request. Tries to return `feed_type`, but if the Accept header does not include that mimetype, tries to return the best match from the feed types that are offered by the `FeedView`. If none of the offered feed types are accepted by the `HttpRequest`, raises `HttpNotAcceptable`.

If mimeparse is installed, it will be used to select the best matching accepted format; otherwise, the first available format that is accepted will be selected.

get_feed(*obj*, *request*, *reverse_name*, *feed_type=None*, **args*, ***kwargs*)

Returns an unpopulated `django.utils.feedgenerator.DefaultFeed` object for this object.

Parameters

- **obj** – The object for which the feed should be generated.
- **request** – The current request.
- **reverse_name** – The name which can be used to reverse the URL of the page corresponding to this feed.
- **feed_type** – The slug used to register the feed class that will be instantiated and returned.

Returns An instance of the feed class registered as `feed_type`, falling back to `feed_type` if `feed_type` is `None`.

populate_feed(*feed*, *items*, *request*)

Populates a `django.utils.feedgenerator.DefaultFeed` instance as is returned by `get_feed()` with the passed-in `items`.

feed_extra_kwargs(*obj*)

Returns an extra keyword arguments dictionary that is used when initializing the feed generator.

item_extra_kwargs(*item*)

Returns an extra keyword arguments dictionary that is used with the *add_item* call of the feed generator.

exception `philo.contrib.winer.exceptions.`**HttpNotAcceptable**

This will be raised in `FeedView.get_feed_type()` if an Http-Accept header will not accept any of the feed content types that are available.

class `philo.contrib.winer.middleware.`**HttpNotAcceptableMiddleware**

Middleware to catch `HttpNotAcceptable` and return an `HttpResponse` with a 406 response code. See **RFC 2616**.

Following Python and Django's "batteries included" philosophy, Philo includes a number of optional packages that simplify common website structures:

- `penfield` — Basic blog and newsletter management.
- `shipherd` — Powerful site navigation.
- `sobol` — Custom web and database searches.
- `waldo` — Custom authentication systems.
- `winer` — Abstract framework for Philo-based syndication.

1.13 Contributing to Philo

So you want to contribute to Philo? That's great! Here's some ways you can get started:

- **Report bugs and request features** using the issue tracker at the project site.
- **Contribute code** using git. You can fork philo's repository either on GitHub or Gitorious. If you are contributing to Philo, you will need to submit a *Contributor License Agreement*.
- **Join the discussion** on IRC at irc://irc.oftc.net/#philo if you have any questions or suggestions or just want to chat about the project. You can also keep in touch using the project mailing lists: philo@ithinksw.org and philo-devel@ithinksw.org.

1.13.1 Branches and Code Style

We use A successful Git branching model with the blessed repository. To make things easier, you probably should too. This means that you should work on and against the develop branch in most cases, and leave it to the release manager to create the commits on the master branch if and when necessary. When pulling changes into the blessed repository at your request, the release manager will usually merge them into the develop branch unless you explicitly note they be treated otherwise.

Philo adheres to PEP8 for its code style, with two exceptions: tabs are used rather than spaces, and lines are not truncated at 79 characters.

1.13.2 Licensing and Legal

In order for the release manager to merge your changes into the blessed repository, you will need to have already submitted a signed CLA. Our CLAs are based on the Apache Software Foundation's CLAs, which is the same source as the Django Project's CLAs. You might, therefore, find the Django Project's CLA FAQ. helpful.

If you are an individual not doing work for an employer, then you can simply submit the `Individual CLA`.

If you are doing work for an employer, they will need to submit the `Corporate CLA` and you will need to submit the Individual CLA `Individual CLA` as well.

Both documents include information on how to submit them.

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX