Zadig Documentation

Release 0.1.1

Antonis Christofides

August 08, 2016

Contents

1 Contents 3

2 Indices and tables 21

Zadig is a system for collaborative management of a web site (also called "content management system", but the latter is a misnomer).

There is no user documentation here; we have documentation for administrators and developers only. The most important piece of documentation is the *Concepts* document, which you must read carefully if you are going to administer a Zadig installation or do any development; but you probably want to read the *Administrator's Guide* first in order to install Zadig.

CHAPTER 1

Contents

1.1 Administrator's Guide

1.1.1 Installation

Prerequisites

You need Python 2.7.

This is the usual way to install the prerequisites (but it may be better to install Pillow and TinyMCE through your operating system's packages first):

```
mkvirtualenv [--system-site-packages] zadig
pip install -r requirements.txt
```

Installing a development/testing instance

You need to:

1. Make copies of three files: `zadig/settings/local-example.py`, `zadig/core/templates/base-example.html`, and `zadig/core/static/style-example.css`; the copies should be in the same location as the originals and have the same name but without the `-example` part.
2. Symlink the tinymce www directory (the one containing the `tiny_mce.js` file) to `static/tinymce`.
3. If needed, edit your copied `zadig/settings/local.py` (although it will work as it is, using SQLite to create a database called `testdb`).
4. Create the empty database, depending on the RDBMS you are using (you can skip this step for SQLite).
5. Run **`python manage.py migrate`**.
6. Run **`python manage.py createrootpage`**.
7. Run **`python manage.py runserver`**.

After doing all that, point your browser to http://localhost:8000/.

Note

It is now a very good time to read the *Concepts* document.

Installing a production instance

There are the following differences from installing a development/testing instance:

1. You should keep your configuration, custom template files, and custom CSS in a separate directory, such as `/etc/zadig`. In addition, you should store binary files in a separate directory, such as `/var/local/lib/zadig`, rather than in the `storage` subdirectory of the Zadig distribution, as it is configured by default to do.
2. You should precompile the Zadig distribution Python files, since your web server (or your Django WSGI process, or whatever it is) should not have permission to write the `.pyc` or `.pyo` files in the Zadig distribution directory.
3. You should use a RDBMS that is suitable for a production server. I use **PostgreSQL_**.
4. You should deploy Django as described in Deploying Django, and configure your web server to serve the static files properly.

The details depend on your setup.

1.1.2 Settings reference

These settings are specific to Zadig; the Django settings are also applicable.

ZADIG_LANGUAGES
A tuple of (language_id, language_description) pairs, indicating the languages available in this installation. The first of these languages is the default language of the site. Example:

```
ZADIG_LANGUAGES = (
    ('en', 'English'),
    ('el', 'Ελληνικ'),
)
```

ZADIG_WORKFLOW_ID
This is the id of the *Workflow* object that is used in the site.

1.2 Concepts

1.2.1 Object containment

In Zadig, objects can be contained in other objects. For example, you can have page `/mypage/hobbies`, and it can contain `/mypage/hobbies/django`. There is no separate "folder" object; instead, any object can act as a container of other objects, in addition to having its own content. If a page has no content, then it shows an index of its contained pages (like Plone does for folders when no page has been set as the folder's default view).

Philosophy

Why don't we specifically use folders, as Plone does? The reason is that it conflicts with the URL scheme, where any resource can have subresources. Because of this conflict, it is difficult for users to understand the notion of a folder and a page. For example, a user would normally create a page `/mypage/hobbies`, and then typically ask: how do I now create `/mypage/hobbies/django`? The Plone reply (hobbies must be made a folder, the current hobbies page must become a page inside the folder, and must be set as default, and a django [page or folder?] must be created inside the folder) is complicated and unintuitive. It also poses a practical problem: the user needs to decide a priori whether the page he is about to create will contain subpages or whether it is likely to remain a single page.

On the other hand, other web management systems, like MoinMoin, have pages and subpages, and each page can have attachments. This is also unintuitive, because, for example, an image does not always belong to a "page"; it rather belongs to a folder. `/myfolder/myimage` is much better and intuitive than `/mypage?attachment=myimage`.

Also note that, in Moin, `/mypage/mysubpage` is a completely different and unrelated page to `/mypage`: it's just a name that contains a slash. But in Zadig, `/mypage/mysubpage` is a page that is contained in `/mypage`, as if the latter were a folder. Moin is even simpler and maps even better on the URL scheme, but Zadig makes property inheritance (e.g. permissions and skins) much simpler and foolproof: any object simply inherits the properties of its container objects.

1.2.2 Entries and vobjects

The end user uses the word *page* (and the more general *object*) with two meanings, owing to the fact that we do automatic versioning like that of a wiki. When we say "delete this page", we mean delete an entire sequence of pages; when we talk about the content of a page, we mean the content of a specific page version.

When we need to distinguish between the two (and this is when we are writing code), we use the word **entry** for a sequence of versioned objects, and the word **vobject** for each specific version of an entry. An entry can be a page entry, an image entry, etc., and a vobject can be a vpage, a vimage, etc. Developers should be very clear about that. The end users should not be bothered, and the word **object** should be used for everything.

1.2.3 Metatags

Each object has a **name**, which is what is shown at the URL. Each object has, in addition, a **long title** and a **short title**. The long title shows everywhere except where there is a shortage of space, namely in the navigation, breadcrumbs, and possibly other applets. The short title can be omitted, in which case the long title is used. We use **title pair** to refer to both titles together.

Because of multilinguality, each object can have several title pairs, one for each language. Although this sounds useless for pages, because a page is in a given language, and therefore its title pair could also be in that given language, multilingual title pairs are useful in the following cases:

- For objects, such as images, that are not bound to a specific language.
- For objects that act as folders, and whose short title therefore shows in the breadcrumbs when you view a contained object.

As an example, consider the English pages `/mypage` and `/mypage/hobbies`, with titles "My Page" and "Hobbies". Suppose `/mypage/loisirs` is the French version of `/mypage/hobbies`, and has title "Loisirs". The breadcrumbs for that page would show "My Page -> Loisirs". However, "Mon Page -> Loisirs" would be better. For this, we add a French title to `/mypage`, although the page is English.

If `/mypage/fr` is the French version of `/mypage`, then you don't want the breadcrumbs to show "Mon Page -> Mon Page"; you want simply "Mon Page". For this, each object has the option to not show in the breadcrumbs, and this is generally for pages that are a kind of "folder defaults" in languages different from the containing object's language.

Except for the name and title pairs, an object can also have a **description**. For pages, the description is a summary of the page, displayed below the title in bold. Objects such as images may have the description in many languages. For uniformity, this applies to all kinds of object, although for some, such as pages, it is mostly useless.

The titles and description of each object are collectively named **metatags**.

In this section we have been talking of "objects". To be precise, each *entry* has a name, and each *vobject* has a set of metatags (i.e. a set of title pairs and a set of descriptions).

1.2.4 Languages

The user can select the **preferred language** from the language selector, which on the default template is on the top left, below login. However, the **effective language** is determined by the object being viewed. If the object being viewed is in a specific language (e.g. it is a page in Greek), then the effective language is the object's language (Greek in our case); but if the object has no associated language (e.g. it is an image), then the effective language is the preferred language.

The translatable strings (i.e. the elements of the user interface) are shown in the effective language, except for messages that intend to notify the user about the languages chosen and why what he sees is in another language (this is so that the user understands at least that message).

When two or more objects contain the same content in different languages, we say that they form one **multilingual group**. In that case, when the user changes the preferred language, the system transfers him to the alternative page (which, thus, also becomes the effective language). When an object exists in the preferred language and in another language, the navigation shows only the preferred language; but if an object exists in a language which is not the preferred language and does not have an equivalent in the preferred language, the navigation shows it anyway.

1.2.5 Deletions

There are two ways to "delete" an object: one is to **mark it as deleted**. In this case, the entry history is retained, and the "deletion" is an event recorded in the entry history. Marking an entry as deleted is creating a new vobject. Since the entry exists, another entry with the same name cannot be created.

The second way to delete is to **remove the entry**, in which case the entry, including the history, is permanently and irrevocably deleted.

Not implemented yet

Removing the entry is not yet implemented in the user interface.

1.2.6 Actions

There are several things you can do with an entry or vobject: view it, edit it, view its contents or its history, delete it, change its state, and so on. We call these operations *actions*: there is the "show action", the "edit action", the "contents action", the "change state action", and so on.

The **show action** is the normal mode of viewing an entry; for an image, for example, the show action results in a "image/jpeg" or similar response that contains binary data. The **info action** results in a page that includes information about the image, buttons for initiating editing, and so on, and is intended mainly for users who have permissions to modify the image. For some object types, such as pages, the show action and the info action may be identical. http://localhost/myobject/ produces the show action for *myobject*, whereas http://localhost/myobject/__info__/ provides the info action.

In GET requests, the action is specified by an item in the URL path which has a prefix and a suffix of two underscores; for example, http://localhost/mypage/__edit__/ is a request to edit *mypage*. If there is no such item in the URL, then the request is to view the entry, that is, for the show action. In POST requests, the action is specified by the *action* parameter.

1.2.7 Permissions and workflows

There are **users**, and each user can belong to one or more **groups**. When someone has not logged on then we say that they are the **anonymous user**. Each user and group has certain **permissions** on each object. There are five kinds of

permissions: *view*, *edit*, *admin*, *search*, and *delete*. Someone with *edit* permission, besides editing the entry, can also mark the entry as deleted, view the list of all subobjects regardless of whether they have any permission on them, and add subobjects to the entry; you also need *edit* permission to view the entry history; with *view* permission you can only view the current entry version (the last vobject). Someone with *admin* permissions can modify the object's permissions and state. Someone with *delete* permission can remove the entry, recursively including its subentries (irrespective of their permissions). Someone with *search* permission can see the entry in listings; e.g. the anonymous user has view permission on a public draft, but not search permission; you can view it if you know its URL, but it won't show in indexes or searches. Each entry has an **owner**, and the owner has all permissions on the object.

Note

Why do you need *edit* permission to view the entry history? This can be useful if old versions contain confidential information. There are alternative ways of doing it, such as adding a *view_history* permission, or allowing individual vobjects to have separate *view* permissions. This, however, adds complication. Zadig is not primarily a wiki, and the possibility for unprivileged users to view the history is not considered essential. Therefore, we chose to keep it simple for the moment, and leave it for the future to find a way around this problem.

At a given time, each entry is in a **state**. A state is a collection of permissions. For example, these are common states:

Private Logged on users can view.

Public draft Anonymous user can view.

Published Anonymous user can view and search.

A **workflow** is a collection of states and state-transition rules. A **state-transition rule** is a many-to-one relationship between states: it shows which are the possible states that follow a given state. Each state-transition rule can be followed by a specific user or group. Usually the owner has permissions to follow any state-transition rule; but this is not true in all workflows. For example, in some blogs, the blog editor, and not the post author, has permission to publish the post.

In the default installation, two states are created, private and published, two state-transition rules (from private to published and vice-versa), and one workflow named "Simple publication workflow" containing all those. The private state has the *view* and *search* permissions on logged on users; the published state has the *view* and *search* permissions on the anonymous user. There are no other permissions defined, which means that only the owner can edit, admin and delete.

Not implemented yet

Currently only the owner and the state can be changed through the user interface. The user interface for directly assigning permissions to users has not been developed yet.

1.2.8 Applets and portlets

An **applet** is essentially a Django custom template tag. The breadcrumbs, for example, is an applet; when you include `{% breadcrumbs %}` in a template, this is replaced by the breadcrumbs. The reason we don't call them simply tags is that applets also have standard ways of storing configuration options; they can have configuration options per entry, or global. But don't worry if you don't understand this yet; just think of "applet" as a synonym for a custom template tag.

A **portlet** is a special kind of applet. There are two things that are special about portlets: the first one is that they have a certain kind of look: they have a title and a body, and the body consists of items (some portlets, like navigation, have only one item in the body). The template tag of the portlet renders the portlet in a very specific manner:

```
<dl class="portlet">
  <dt>Portlet title</dt>
  <dd class="odd">
    First item title
    <span class="details">Details</span>
  </dd>
  <dd class="even">Second item</dd>
  ...
</dl>
```

Because portlets conform to that specification, the CSS style sheet can define a common look for all portlets. (Some portlets might contain only one item, and others might not contain items consisting of title and details; but all portlets are DL elements containing a DT and at least one DD.)

The second thing that is special about portlets is that they are registered: Zadig knows what portlets are available; this makes possible to have user interface where the user can select portlets from a list, instead of having to specify them in a Django template.

Not implemented yet

The portlets for recent changes, pending events, news, and calendar have not been implemented yet. In addition, no functionality that uses registered portlets has been implemented. I think it might be nice if the left and right panes, where portlets are shown, were applets themselves.

1.3 Developer's Guide

There are two things a Zadig developer is expected to do: (1) create new object types; and (2) create applets. We explain those in separate chapters below. It is essential to first read the *Concepts* document.

1.3.1 Creating Zadig applications

A Zadig application is a Django application, and, as such, is added in the `INSTALLED_APPS` setting. The application can reside anywhere in the Python path. The Zadig applications (e.g. `zadig.zstandard`) included in the Zadig distribution are under the `zadig` directory.

Such applications should have an `__init__.py` file and a `__models__.py`. If they have any extra templates, they should also have a `templates` directory. If they have applets, they should also have a `templatetags` directory. If they have any additional general actions, these should be in `views.py` (general actions will be examined below).

1.3.2 Creating new object types

New object types are declared in `models.py`. You need to subclass *`Entry`* and *`VObject`*, and create a form for editing the object.

The *`Entry`* subclass must define the `vobject_class` class attribute, whose value must be the *`VObject`* subclass. The *`Entry`* subclass should also define *`typename`*. Finally, when necessary, it should also override `edit_template_name`, *`edit_subform()`*, and *`process_edit_subform()`*.

The *`VObject`* subclass should have Django fields for storing the vobject content, and it should also define the `action_view()` and *`action_info()`* methods. If the subclass has related models (e.g. other models that have foreign keys to the subclass), you may also need to override the *`duplicate()`* method.

If you use the existing applications' object types implementation as an example, you should not have any problem understanding things better.

1.3.3 The action dispatcher

When the URL of a GET request is of the form */path_to_entry/__actionname__/remainder*, or the *action* parameter of a POST request is set, and the action is not known to the Zadig core code, then Zadig searches for such an action in three places:

1. It checks whether the *VObject* pointed by *path_to_entry* has a method called `action_actionname`.
2. Failing that, it searches for such a method in the *Entry* class.
3. Failing that, if *actionname* contains a dot, i.e. it is of the form *app.func*, it searches for a callable *func* in `zadig.app.views`.

If no appropriate callable is found, `Http404` is raised. If it is found, it is called, passing the vobject as the first argument. In GET requests, `vobject.request.parms` is set to the URL *remainder*.

An example of such a view is the "resized" action of an image. If you append `/__resized__/400` to the URL of an image, then you will see the image resized so that its largest dimension is 400 pixels. This is accomplished because the `zadig.zstandard.VImage` class has a `action_resized()` method.

1.3.4 Creating applets

To create an applet, create a custom template tag (read the Django documentation for that). If you want your applet to store information in the database, add models in `models.py`.

Incomplete

I've not yet gotten to write about entry option sets.

If the applet is a portlet, you need to do two more things. First, make sure that the tag output conforms to the portlet specification:

```
<dl class="portlet">
  <dt>Portlet title</dt>
  <dd>First item</dd>
  <dd>Second item</dd>
  ...
  <dd class="lastItem">Last item</dd>
</dl>
```

Second, in the top level `__init__.py` of your application, add the following:

```
from zadig.core import portlets
portlets.append({ 'name': _(u"My portlet"), 'tag': "myportlettag", },
                # Add more items to this list if your
                # application defines more than one portlets.
               )
```

1.3.5 The request object

Low level Zadig code needs to be aware of the *request* object all the time. It uses the *request* object for two purposes: First, it uses `request.user` to check whether the user has permissions to do various operations; for example, when

you call the `get_by_path()` method of the `default Entry manager`, it only returns an Entry if the user has permission to view it; and the default manager's querysets only include entries which the user has permission to search. Second, some additional information is stored for later usage in the *request* object, such as the preferred and effective languages.

In order to avoid passing the request object all the time and treat this at a low level, transparently for the high level developer, which is important for security, the low level code gets the request object by calling `zadig.core.utils.get_object()` (the Zadig middleware saves it at the start of execution by calling `zadig.core.utils.set_object()`). You should, however, avoid to use that at a high level; consider it a Zadig internal. Use the request object as it is normally being used in Django; just be aware that the Zadig core objects have access to it even if you don't provide it. In addition, *Entry* and *VObject* instances have a *request* attribute which you can use, particularly when you are subclassing them.

If you are not using the API in the context of a web server (for example if you are creating a command-line utility), and therefore you don't have a request object, low level code will sometimes fail and sometimes automatically consider that it is running with the permissions of the anonymous user. If this is insufficient, do something like the following:

```
from django.http import HttpRequest
from django.contrib.auth.models import User
from zadig.core.utils import set_request
request = HttpRequest()
request.user = User.objects.get(username=MY_USERNAME)
set_request(request)

# Do what you need at this point

set_request(None) # This is for cleanup and it is important in case you
                  # attempt to set it to a not None value again further
                  # below.
```

1.4 Internal API

It is important to read the *Concepts* document before studying the API, otherwise you will not understand what I'm talking about.

In the description of the database models below, it goes without saying that all models have an *id* field, which is usually not mentioned unless there is something unusual about it.

1.4.1 The Entry model

class `zadig.core.models.`**`Entry`**(*path*)

Stores an entry. It is always a subclass that is used, rather than *Entry* itself. The constructor of the subclass creates a new entry at the specified path. The parent entry must already exist and the logged on user must have appropriate permissions on the parent entry; otherwise, an exception is raised. If other entries with the same parent entry exist, the entry is added as the last sibling.

Entry objects have the following database attributes:

`btemplate`

A string with the filename of the base template. If empty, it means to use the same template as the parent entry; see *base_template* for more information.

`container`

A foreign key to *Entry* (that is, to self). It indicates which entry contains the entry. This attribute can be null; there is one and only one entry that has a null *container*, and it is the root entry.

The *related_name* of this attribute is `all_subentries`. Therefore, if you have an entry `myentry`, then `myentry.all_subentries` is the entries contained in `myentry`. You should not, however, use `all_subentries`, unless you have a good reason to do so; instead, use *subentries()*, which checks permissions.

multilingual_group
A foreign key to *MultilingualGroup*, indicating the multilingual group, if any, to which the entry belongs.

name
The name of the entry.

owner
A foreign key to `django.contrib.auth.models.User`, indicating the owner of the entry.

seq
The sequence of the entry in its container: 1 for the first entry, 2 for the second entry, and so on.

state
A foreign key to *State*, indicating the state of the entry.

vobject
A foreign key pointing to the current vobject, that is to the vobject that has the maximum *version_number*) for the entry.

This field is redundant, but has to be there to enable some queries to run properly; see these two posts for more information. You should never attempt to set it; Zadig automatically updates it upon saving a new vobject.

Entry has the following class attributes:

all_objects
This manager returns all objects. This is a copy of the default django manager. In general, you should avoid using this; you should instead use `objects`, which is an *EntryManager*.

Entry objects also have the following attributes, properties and methods:

absolute_uri
This read-only property returns the absolute URI for the show action of the entry.

add_details(*vobject*, *form*)
The *action_edit()* method is implemented in *Entry* and not in its subclasses. When creating a new vobject as a result of edit form submission, it doesn't know how to process all attributes submitted with the form. It therefore processes only those it can, and then calls this method (which should be implemented in the subclass), which should process any *form* attributes special to the subclass and modify *vobject* accordingly.

alt_lang_entries
A list with the alternative language entries. The order in this list is the order with which languages are listed in *ZADIG_LANGUAGES*.

base_template
A string with the filename the base template the entry should use. This is the base template used to render the page (the base template inherited by the actual template). The result is *btemplate*, except if this is empty, in which case the parent's base template is used; if the top level entry's `btemplate` attribute is also empty, then `base.html` is returned.

can_contain(*child*)

classmethod can_be_contained(*parent*)
Not all kinds of objects can contain all kinds of objects; for example, only blog objects can contain blog post objects; and blog post objects cannot contain other pages. Each `Entry` subclass has the

can_contain() method and the *can_be_contained()* class method that indicate what kind of objects can be added as subobjects.

Note

In this text, we use "subobjects" in the containment sense; that is, an object can contain subobjects. We use "subclass" in the usual hierarchical sense.

The *can_contain()* method indicates whether the object is willing to contain a *child* subobject, where *child* is an *Entry* subclass; and the *can_be_contained()* class method indicates whether the class is willing to create a new subobject of *parent*, where *parent* is an *Entry* subclass. You should only call the later, because it calls the former itself, whereas the former does not call the latter and therefore may miss additional restrictions.

The toplevel *can_contain()* returns `True`, provided that the user has appropriate permissions. The toplevel *can_be_contained()* returns *parent*`.can_contain(`*cls*`)`.

contains(*entry*)
Check whether *entry* is contained, directly or indirectly, in *self*. Returns `True` if *self* is the parent of *entry*, or the parent of its parent, or any ancenstor, and `False` otherwise.

action_contents()
Return a `django.http.HttpResponse` object with the contents of the entry.

creation_date
The date the entry was created, i.e. the date of its first vobject. See also *last_modification_date*.

descendant
Entry is always subclassed. If you get a grip on an object of type *Entry* when in fact you need that same object but in its descendant class, and you don't know which subclass it is, use this property, which gives you the same object but in the proper subclass.

edit_subform(*data=None*, *files=None*, *new=False*)
When the object is being edited, a form is shown to the user; this consists of items that are common to all *Entry* subclasses, plus items that are specific to the subclass. The items that are specific to the subclass are the *edit subform*, and they are a Django form, i.e. a `django.forms.Form` subclass. *edit_subform()* returns that form. If *data* (and optionally *files*) are specified, it returns a bound form (`data=request.POST` and `files=request.FILES` is normally specified upon a POST request); otherwise, if *new* is `True`, we are creating a new entry (normally an empty form should be returned); otherwise, we are editing an existing entry (normally an unbound form with the last vobject data as initial data should be returned).

The toplevel *edit_subform()* returns an empty form. Subclasses should redefine as needed.

action_edit([*new=False*])
Return a `django.http.HttpResponse` object. Depending on the contents of the *request object*, it either processes a submitted form (either modifying the entry or finding an error in the submitted form) and returns the response, or it returns a page with a form for editing the entry. *self* must be an *Entry* subclass.

If *new* is `True`, it means that the entry does not exist yet but is in the process of being created. When calling in this manner, *self* must be an *Entry* subclass, and *container* must have been set to the entry of which the new entry will be a child; the other attributes are irrelevant. Depending on the contents of the *request object*, the method will then either process a submitted form (either creating the entry or finding an error in the submitted form) and return the response, or return a page with a (mostly empty) form for filling in the entry.

get_vobject([*version_number*])
Return the entry's vobject; actually returns a *VObject* descendant. If *version_number* is not specified, it returns the latest vobject. An exception is raised if the user does not have permission to view the vobject.

last_modification_date()
The last modification date of the entry, i.e. the date of its latest vobject.

action_history()
Return a `django.http.HttpResponse` object with the history of the entry.

move(*target_entry*)
Move the entry from its current container to *target_entry*, which will be the new container. Also create a new redirection entry at the old location. Verifies permissions to do all that.

ownername
Return the owner full name, if available, otherwise the username.

path
This read-only property returns the URL path to the entry, not including a leading or trailing slash. See also *spath* and *absolute_uri*.

permissions
Return the permissions the logged on user has on the entry.

action_permissions()
Return a `django.http.HttpResponse` object with the permissions of the entry.

possible_target_states
A list of *State* objects, which are the possible states to which the workflow allows the current user to move the entry to.

process_edit_subform(*vobject*, *subform*)
After the user submits the edit form, the subform returned by *edit_subform()* must be processed. This method receives the newly created *vobject* and the submitted *subform* and processes as needed. The top level method does nothing; it is subclasses that must define how the processing is done.

rename(*newname*)
Rename the entry to the specified new name; it also creates a new redirection entry with the old name.

reorder(*source_seq*, *target_seq*)
Move the subentry with `seq=`*source_seq* before subentry with `seq=`*target_seq*. *source_seq* and *target_seq* are integers. The function changes the order of the children of the entry. The child that has `seq=`*source_seq* is moved before the child that has `seq=`*target_seq*, unless *target_seq* is one more than the number of children, in which case the child is moved to the end. The other children are renumbered as needed (i.e. their *seq* is modified accordingly). Raises an exception if *source_seq* or *target_seq* are inappropriate; for example, if *source_seq* is larger than the number of children, or if *source_seq* and *target_seq* are both the same number. Also raises an exception if the user does not have permission to do this.

request
A copy of the *request object*.

set_altlang(*altpath*)
Set *multilingual_group* so that the entry specified by *altpath* is an alternative language entry. *altpath* is a path to another entry.

Note that this method will rarely, if ever return an error: it tries to be smart and do the right thing. Maybe it's too smart. One case where it raises an exception is if this entry and the other entry are in the same language or they don't have a language specified.

If the entry specified by *altpath* does not exist or is inaccessible, the method does nothing.

If one of the entries involved (say A) is already in a multilingual group, and the other entry (say B) is not, it adds B to the multilingual group of A, unless there is already an entry with that language in the group; in that case, it removes A from the group and creates a new multilingual group for A and B.

If both entries are already in multilingual groups, it attempts to join these groups together; but again, if there are language conflicts, it removes one or both of the entries from their old groups as needed.

spath
This read-only property returns the full relative URL path to the entry, starting and ending in a slash. (This is more complicated than just prefixing and suffixing *path* with a slash, because then the root entry would be two slashes instead of one.) See also *path* and *absolute_uri*.

subentries
A query set of *Entry* objects, which are the subentries which the user has permission to search, in order.

template_name
This class attribute is the name of the template for editing the entry. Frequently the inherited value is OK.

touchable
`True` if the current user has either *edit* or *admin* permission on the entry. Primarily used by the template to check whether to show the editing buttons.

type
The class name, such as "PageEntry", or "ImageEntry".

typename
This is a class attribute, with a translatable, human readable name for the type, such as "Page" or "Image".

undelete()
When called on an entry whose last vobject is a `deletion_mark`, it creates an additional vobject identical in content to the vobject before undeletion, thereby reverting to that last vobject.

1.4.2 The default Entry manager

class `zadig.core.models.`**EntryManager**
Three things are important about the default *Entry* manager (accessible via `Entry.objects`):

1. The manager automatically filters objects and does not return those for which the user does not have search permission (see *request object* to understand how this is done), or objects which are deleted (see *deletion_mark*).
2. The manager is inherited by subclasses, contrary to normal django practice.
3. The manager has a few additional methods, described below.

See also *zadig.core.models.Entry.all_objects*.

get_by_path(*path*)
Return the entry at the specified path; raise `django.http.Http404` if such an entry does not exist or the user does not have permissions to view it (you normally don't need to handle that exception: it will result in a 404 page).

Generally you should much prefer to use the above method when retrieving Entries, because it will take care of permissions at a low level.

exclude_language_duplicates(*effective_language_id*)
Return a query set that does not contain items that are represented in the default query set in a more appropriate language. For example:

```
Entry.objects.exclude_language_duplicates(
                                    request.effective_language)
```

The resulting query set will not contain entries which are in the same multilingual group. If two entries are one the equivalent of the other in another language, only one of them will be included. If one of them is in the effective language, then it will be included, and the other will be excluded; if both are in a language which is not the effective language, then one of them will be included at random.

1.4.3 The VObject model

class `zadig.core.models.`**VObject**

This model is the parent of models that inherit it using multi-table inheritance, and stores a vobject. This model does not actually store the content of the vobject; this is stored by the descendant model. *VObject* provides the following attributes and methods:

entry
Foreign key to *Entry*.

version_number
An integer.

date
The date in which the vobject has been created.

deletion_mark
If this boolean attribute is `True`, then this vobject is a deletion mark, which means that the entry was marked as deleted when this vobject was created. The vobject's metatags and content are irrelevant in this case.

language
A foreign key to *Language* designating the language of the vobject.

descendant
VObject is always subclassed. If you get a grip on an object of type *VObject* when in fact you need that same object but in its descendant class, and you don't know which subclass it is, use this property, which gives you the same object but in the proper subclass.

request
A copy of the *request object*.

action_show()
Return a `django.http.HttpResponse` object that shows the vobject.

action_info()
Returns a `django.http.HttpResponse` object showing the info for the vobject.

view_deleted()
Used internally. When a vobject is deleted (i.e. has *deletion_mark* set), then this method is called instead of its other actions This method decides if it should return a response or whether it should raise a `django.http.Http404`.

duplicate()
Create and save an exact copy of the vobject; the only thing that is different in the newly created vobject is its id and date. Used for reverting an entry to an old vobject.

VObject has a custom manager:

objects
objects is similar to the default manager, except that it has an additional method:

get_by_path(*path*[, *version_number*])
Return the vobject that corresponds to the entry at the specified path. If *version_number* is not specified, it returns the latest vobject. `django.http.Http404` is raised if the entry does not exist or

the user does not have permissions to view it (you normally don't need to handle that exception: it will result in a 404 page).

1.4.4 Other core models

class zadig.core.models.**EntryPermission**
Permissions assigned to the entry besides those assigned to its state. Has three attributes: entry, lentity and permission, all foreign keys to the respective model.

class zadig.core.models.**Language**
Contains languages. It has only one field, id, storing the language id, as a 5-character long string, in the form "en" or "en_us" etc.

class zadig.core.models.**Permission**
A lookup that lists permissions: "view", "edit", "admin", "search", "delete". Has only a descr attribute.

class zadig.core.models.**Lentity**
A Lentity represents either a user or a group, and is used whenever any of the two can be used; for example, a certain permission can be given either to a user or to a group. It has three attributes: user, group and special. The former two are foreign keys to django.contrib.auth.models.User and django.contrib.auth.models.Group. Either user, or group, or special, must be not null; the other two must be null. If user is not null, the *Lentity* represents a user. If group is not null, the *Lentity* represents a group. special, besides null, can have one of the following values:

zadig.core.models.**EVERYONE**
zadig.core.models.**LOGGED_ON_USER**
These values denote any user (including the anonymous user), and any logged on user.

zadig.core.models.**OWNER**
This value denotes the owner of the entry. This is useful in cases where we need to generalize the owner; for example, in state-transition rules, such as "the owner of an object has permission to publish it".

zadig.core.models.**PERM_VIEW**
zadig.core.models.**PERM_EDIT**
zadig.core.models.**PERM_DELETE**
zadig.core.models.**PERM_ADMIN**
zadig.core.models.**PERM_SEARCH**
These values indicate the set of users who have the respective permission. This can also used in state-transitions, for example: "users who have edit permission may publish". Care should be taken to avoid circles, such as "a user with edit permission may edit", so *EntryPermission* and *StatePermission* should generally not refer to such special users.

There is the following method:

includes(*user*, *entry=None*)
Return True if the lentity represents *user* or a group that contains *user*. *user* is a django.contrib.auth.models.User instance. *entry* should also be supplied, in case the lentity is *OWNER*, *PERM_VIEW*, *PERM_EDIT*, *PERM_DELETE*, *PERM_ADMIN*, or *PERM_SEARCH*; in these cases, whether *user* is a member of the lentity depends on the entry.

class zadig.core.models.**State**
A list of states. Contains only a descr attribute.

class zadig.core.models.**StatePermission**
A state is a collection of permissions. This model stores the permissions that comprise the state. It has three attributes, state, lentity and permission, which are foreign keys to *State*, *Lentity* and *Permission*, and designate that said lentity has said permission on said state.

class zadig.core.models.**StateTransition**
A state transition. Has three attributes, source_state and target_state (both foreign keys to *State*), and lentity (foreign key to *Lentity*, the user or, more commonly, group who has permission to perform this transition).

class zadig.core.models.**Workflow**
A workflow is a collection of states and state-transition rules. The model has three attributes: name, which is a string, states, and state_transitions. The last two are many to many fields to *State* and *StateTransition*.

class zadig.core.models.**VObjectMetatags**
Stores the metatags of a vobject. Has five attributes: vobject and language are foreign keys to the respective models; title, short_title and description are strings. Also has method get_short_title(), which returns the short title, or the title if the short title is empty. The related name is *metatags*.

The default manager has an additional property, default, which returns an appropriate set of metatags. Normally the manager has access to the *request object*, in which case it returns the metatags in the effective language. Failing that (because the request object is unavailable, or because there are no metatags in the effective language), it returns the metatags in the language of the vobject; and if such a set does not exist, it returns a random set of metatags.

class zadig.core.models.**ContentFormat**
A lookup storing content formats, such as "text/html" or "text/x-rst". Has only a descr field.

class zadig.core.models.**MultilingualGroup**
Stores the multilingual groups. It has no field besides id, as it is only used to group multilingual entries together through their *multilingual_group* field. It has a check() method, which checks for integrity: it deletes the group if it contains less than two entries, checks that there are no multiple language occurrences in the group, makes other similar checks, and raises ValidationError if a check fails.

class zadig.core.models.**Page**(*VObject*)
Inherits *VObject* and designates a page. Has attributes format, a foreign key to *ContentFormat*, and content, a text field.

class zadig.core.models.**File**(*VObject*)
Inherits *VObject* and designates a file. Has attribute content, a file field.

class zadig.core.models.**Image**(*VObject*)
Inherits *VObject* and designates an image. Has attribute content, an image field.

1.4.5 Utility functions

The functions below are in module zadig.core.utils.

zadig.core.utils.**split_path**(*path*)
Return a list with the path items. Roughly path.split('/') but not confused by trailing slash, will ignore a starting slash, will work on an empty string or single slash, and it always includes an empty string as the first path item, which stands for the root entry.

zadig.core.utils.**join_path**(*path_items*)
Return a string with the path items joined into a path. *path_items* is a sequence of strings (supplied either as string arguments, or as a single sequence argument); each string may contain slashes. Roughly '/'.join(path_items) but will not result in duplicate slashes (strips leading and trailing slashes from each path item before joining), and the result never includes a trailing or leading slash.

zadig.core.utils.**get_current_path**(*request*)
Return the path to the current entry. This is something like request.path, but does not include any directive like __edit__ and so on.

zadig.core.utils.**including_lentities**(*user*)
Return a queryset of *Lentity* objects that either are or contain *user*, which is a django.contrib.auth.user object.

zadig.core.utils.**set_request**(*request*)
zadig.core.utils.**get_request**()
For information about these functions, see the *request object*.

zadig.core.utils.**sanitize_html**(*html*)
Sanitize the HTML provided and return a sanitized version. This is done for two reasons: (a) to avoid cross site scripting attacks; (b) to discourage users from using too much markup. It only allows specific (whitelisted) tags and specific (whitelisted) attributes, deleting all the rest, and it also sanitizes the content of *href* and *src* attributes, by allowing only specific (whitelisted) URL schemes. The whitelists are hardwired in this version.

1.4.6 Decorators

@**require_POST**
This is similar to the Django require_POST decorator, but it can be used to decorate any function or method and not only the views.

1.5 Migrating from Plone

1.5.1 Important things to have in mind

The migration script works by reading Plone's web site; it does not read the Zope database. I don't expect this change; people comfortable with the Plone/Zope API will probably not be interested in becoming Zadig developers, whereas Zadig developers are unlikely to learn the Plone/Zope API.

The fact that the migration script reads Plone's web site has several side-effects:

- Migration is slow.
- Parsing of the Plone pages could be unreliable and may depend on the skin Plone is using. For example, the main content of a page is assumed to be the content of a div with id='parent-fieldname-text', but this might not always be the case. (My Plone sites were using the classic skin.)
- Much of the information the migration script needs to dig out is in the Plone's "Edit" page. Therefore, the script (which must log on to Plone as an administrator) visits the Edit page; if the page is locked, it unlocks it; and since clicking the "Cancel" button afterwards is not trivial for the script and was not very important for my sites, the script currently leaves all pages locked. Since migration may take you days or weeks of tests, your users may be viewing locked pages all the time, and, worse, their locks may be unlocked. A workaround, besides making a copy of the entire Plone site, is to do tests at a small subdirectory each time; or, maybe better, modify the migration script to skip locked pages, and to unlock those it locked after it's done.

Another problem is that the migration is a one-off thing; once you do it, you don't care much about it again. I've migrated two of my sites, there are two remaining (and they are easier), and once I'm done, I don't think I will ever care about the migration script again; the same applies more or less to everyone, which means that the migration script will be poorly maintained. In addition, when I wrote the migration code, I did do careful work, but I did not put into it the care that I put in Zadig core (which I don't know how many times I've rewritten and reorganized).

That said, I did migrate my sites with the migration script, and it worked wonderfully. Consider it, therefore, a good start, which you can use further develop until it does what you want. Further below I document its internals in order to make it easier for you. Please consider submitting your patches.

1.5.2 Using the migration script

The migration script is in `bin/plone2zadig`. It converts a Plone site to a Zadig site. It does so by visiting the Plone site on the web. The script must log on the Plone site as a user who has edit permissions on all pages (the administrator for example). Use it like this:

```
bin/plone2zadig config_file
```

The *config_file* a collection of lines of the form `OPTION=value`. It is actually a Python program that defines several variables. The options are:

The options are:

PLONE_URL
: The topmost Plone url to start converting. That page and all its subpages (except those marked for exclusion) will be read and converted.

PLONE_EXCLUDE
: The URIs of this tuple will be excluded from conversion. The URIs can be full URLs or relative paths whose base is *PLONE_URL*.

PLONE_AUTH
: The value the Plone `__ac` cookie should have. It must be a user capable of editing all the objects from *PLONE_URL* downwards.

TARGET_PATH
: Will migrate *PLONE_URL* to this Zadig path, and subobjects to subpaths.

REMOVE_TARGET
: Specifies what happens if *TARGET_PATH* already exists. If True, the TARGET_PATH and its subentries are deleted before migration; if False (the default), its content is overwritten, as is the content of any already existing subentry. When such overwriting occurs, the existing object and the overwriting object must be of the same type, or an exception is raised.

OWNER
: The objects created in Zadig will belong to this user.

DEFAULT_LANGUAGE
: When a Plone page is "language neutral", it will be considered to be in this language.

LOG_LEVEL
: The logging level; one of DEBUG, INFO, WARNING, ERROR, and CRITICAL. Default is WARNING. Don't use CRITICAL, it will miss the errors.

All the above except *LOG_LEVEL* and *PLONE_EXCLUDE* are compulsory.

1.5.3 Migration script internal API

In case you haven't noticed, the migration script is quite small—530 lines at the time of this writing. This is because practically all the work is done by the Zadig core API and BeautifulSoup. You need to be comfortable with both.

The most important item in the migration script is the *PloneObject* class, which represents, unsurprisingly, a Plone object.

class PloneObject(*url*, *soup=None*)
: *PloneObject* is always subclassed, as a `PloneFolder`, a `PlonePage`, and so on. However, in order to create a *PloneObject* instance, you merely supply the object's URL, without needing to know what kind of object it is; the constructor will retrieve the data from the URL, find out what kind of object it is, and return an instance not of *PloneObject*, but of the appropriate subclass. It will visit and parse both the URL and the

equivalent "edit" page. It will unconditionally unlock the page if it is locked, and when it's finished it will leave it locked (this is a bug).

The *soup* argument must not be supplied, or it must be `None`. It is used internally, because the constructor reads the URL, decides what type of object it is, and calls the constructor of its appropriate subclass.

PloneObject instances have the following attributes:

soup
The `BeautifulSoup` object that represents the parsed HTML retrieved from the URL.

editsoup
The `BeautifulSoup` object that represents the parsed HTML of the Plone's "edit" view for the object.

title

short_title

description
These three attributes store the title and description of the Plone object. The *short_title* is always empty, as Plone does not have this feature; however, sometimes it is set by subclasses; in particular, when we have a Plone folder with a default view, we consider the default view's title to be the title and the folder's title to be the short title.

state
The state of the Plone object as a string, such as "Private" or "Published".

PloneObject instances of subclasses have the following method:

migrate(*request*, *path*)
This method does the migration. First it calls the same method of the superclass (which essentially creates and saves the entry), then it creates and saves the vobject, and the metatags, and finally it calls the `post_migrate()` method of the superclass, which does some common endwork (sets alternative language).

Now all the above is only the basic idea. It's not perfectly documented, and the code is a bit messy, but the above should be more than enough to get you find your way in the code.

1.6 Copyright and license

The copyright applies to all Zadig files, except those where something else is mentioned.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

A

absolute_uri (zadig.core.models.Entry attribute), 11
action, 6
    contents, 6
    edit, 6
    history, 6
    info, 6
    show, 6
action_contents() (zadig.core.models.Entry method), 12
action_edit() (zadig.core.models.Entry method), 12
action_history() (zadig.core.models.Entry method), 13
action_info() (zadig.core.models.VObject method), 15
action_permissions() (zadig.core.models.Entry method), 13
action_show() (zadig.core.models.VObject method), 15
add_details() (zadig.core.models.Entry method), 11
all_objects (zadig.core.models.Entry attribute), 11
alt_lang_entries (zadig.core.models.Entry attribute), 11
anonymous user, 6
applet, 7

B

base_template (zadig.core.models.Entry attribute), 11
breadcrumbs, 7
btemplate (zadig.core.models.Entry attribute), 10

C

can_be_contained() (zadig.core.models.Entry class method), 11
can_contain() (zadig.core.models.Entry method), 11
container (zadig.core.models.Entry attribute), 10
contains() (zadig.core.models.Entry method), 12
creation_date (zadig.core.models.Entry attribute), 12

D

date (zadig.core.models.VObject attribute), 15
DEFAULT_LANGUAGE (built-in variable), 19
delete object, 6
deletion_mark (zadig.core.models.VObject attribute), 15
descendant (zadig.core.models.Entry attribute), 12
descendant (zadig.core.models.VObject attribute), 15
description, 5
description (PloneObject attribute), 20
duplicate() (zadig.core.models.VObject method), 15

E

edit_subform() (zadig.core.models.Entry method), 12
editsoup (PloneObject attribute), 20
entry, 5
entry (zadig.core.models.VObject attribute), 15
exclude_language_duplicates() (zadig.core.models.EntryManager method), 14

F

folders, 4

G

get_by_path() (zadig.core.models.EntryManager method), 14
get_by_path() (zadig.core.models.VObject method), 15
get_vobject() (zadig.core.models.Entry method), 12
group, 6

I

includes() (zadig.core.models.Lentity method), 16

L

language, 5
    effective, 5
    preferred, 5
language (zadig.core.models.VObject attribute), 15
last_modification_date() (zadig.core.models.Entry method), 13
LOG_LEVEL (built-in variable), 19

M

mark object as deleted, 6
metatag, 5
migrate() (PloneObject method), 20
move() (zadig.core.models.Entry method), 13

multilingual group, 5
multilingual_group (zadig.core.models.Entry attribute), 11

N

name (zadig.core.models.Entry attribute), 11
navigation, 7

O

object, 5
objects (zadig.core.models.VObject attribute), 15
owner, 6
OWNER (built-in variable), 19
owner (zadig.core.models.Entry attribute), 11
ownername (zadig.core.models.Entry attribute), 13

P

path (zadig.core.models.Entry attribute), 13
permissions, 6
permissions (zadig.core.models.Entry attribute), 13
PLONE_AUTH (built-in variable), 19
PLONE_EXCLUDE (built-in variable), 19
PLONE_URL (built-in variable), 19
PloneObject (built-in class), 19
portlet, 7
possible_target_states (zadig.core.models.Entry attribute), 13
process_edit_subform() (zadig.core.models.Entry method), 13

R

remove entry, 6
REMOVE_TARGET (built-in variable), 19
rename() (zadig.core.models.Entry method), 13
reorder() (zadig.core.models.Entry method), 13
request (zadig.core.models.Entry attribute), 13
request (zadig.core.models.VObject attribute), 15
require_POST() (built-in function), 18

S

seq (zadig.core.models.Entry attribute), 11
set_altlang() (zadig.core.models.Entry method), 13
short_title (PloneObject attribute), 20
show, 6
soup (PloneObject attribute), 20
spath (zadig.core.models.Entry attribute), 14
state, 6
state (PloneObject attribute), 20
state (zadig.core.models.Entry attribute), 11
state-transition rule, 6
subentries (zadig.core.models.Entry attribute), 14

T

TARGET_PATH (built-in variable), 19
template_name (zadig.core.models.Entry attribute), 14
title, 5
  long, 5
  short, 5
title (PloneObject attribute), 20
title pair, 5
touchable (zadig.core.models.Entry attribute), 14
type (zadig.core.models.Entry attribute), 14
typename (zadig.core.models.Entry attribute), 14

U

undelete() (zadig.core.models.Entry method), 14
user, 6
  anonymous, 6

V

version_number (zadig.core.models.VObject attribute), 15
view_deleted() (zadig.core.models.VObject method), 15
vobject, 5
vobject (zadig.core.models.Entry attribute), 11

W

workflow, 6

Z

zadig.core.models.ContentFormat (built-in class), 17
zadig.core.models.Entry (built-in class), 10
zadig.core.models.EntryManager (built-in class), 14
zadig.core.models.EntryPermission (built-in class), 16
zadig.core.models.File (built-in class), 17
zadig.core.models.Image (built-in class), 17
zadig.core.models.Language (built-in class), 16
zadig.core.models.Lentity (built-in class), 16
zadig.core.models.Lentity.zadig.core.models.EVERYONE (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.LOGGED_ON_USER (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.OWNER (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.PERM_ADMIN (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.PERM_DELETE (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.PERM_EDIT (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.PERM_SEARCH (built-in variable), 16
zadig.core.models.Lentity.zadig.core.models.PERM_VIEW (built-in variable), 16
zadig.core.models.MultilingualGroup (built-in class), 17
zadig.core.models.Page (built-in class), 17
zadig.core.models.Permission (built-in class), 16
zadig.core.models.State (built-in class), 16

zadig.core.models.StatePermission (built-in class), 16
zadig.core.models.StateTransition (built-in class), 16
zadig.core.models.VObject (built-in class), 15
zadig.core.models.VObjectMetatags (built-in class), 17
zadig.core.models.Workflow (built-in class), 17
zadig.core.utils.get_current_path() (built-in function), 17
zadig.core.utils.get_request() (built-in function), 18
zadig.core.utils.including_lentities() (built-in function), 18
zadig.core.utils.join_path() (built-in function), 17
zadig.core.utils.sanitize_html() (built-in function), 18
zadig.core.utils.set_request() (built-in function), 18
zadig.core.utils.split_path() (built-in function), 17
ZADIG_LANGUAGES (built-in variable), 4
ZADIG_WORKFLOW_ID (built-in variable), 4