Misago Documentation

Release 0.6

Rafał Pitoń

July 06, 2016

Misago is featured internet forum solution developed in accordance with practices and trends currently used in web software development.

Like in any full stack solution, Misago provides different levels of interaction that allow for fullfilling different use cases. People who come to your site to talk with each other and exchange their opinions and knowledge will most likely asociate Misago with what they see in their browsers, while developers that are developing site around Misago will focus on its code, API's and implementations details.

The goal of Misago documentation is to provide refence and explanation of features available on each of the stack's layers.

Warning: Documentation is currently in early Work in Progress state. New documents are added as features they cover are implemented, with exception of "Users" part of documentation which will be worked on once Misago reaches beta testing stage.

CHAPTER 1

Table of Contents

1.1 Setup and Maintenance

Misago is Python and Django application which means system requirements as well as setup process and way maintenace taks are performed may appear confusing and suprising to administrators that have no experience outside of running PHP solutions.

1.1.1 Installing Misago

Misago installation is three step process. First you compare your server's specification to check if will be able to run Misago. Next you setup Misago and all extra services required for it to function like e-mails, database and automatic maintenance jobs. Finally you get your site running and accessible for your users.

Requirements

Before you start make sure your hosting provider grants you:

- SSH access to server
- Python **2.7** (this is important, Python version 3 and older than 2.7 are not supported) and PIP
- Node.js runtime with LESS
- PostgreSQL database
- 60 megabytes of RAM for Misago process
- A way to direct HTTP traffic from your domain to Misago
- CronTabs or other task scheduler

This is not a problem on VPS or dedicated servers, but availability of shared servers that meet those requirements differs from country to country.

Speaking of shared servers, ability to download, compile and run software from internet may be needed, but different ISP's have different approach to this. Some options come with all dependencies preinstalled, others let you install them yourself and others require you to mail them every time you need something installed. Generally you should avoid offers coming from last group because this turns running Python apps into a hore.

Setup

To install Misago setup and activate virtual environment for your site and then fire following command:

```
python setup.py install
```

This will install Misago in your virtual environment and make "misago-start.py" script available for you to use to create pre-configured Misago site.

Now decide on your site's "name". This name will be used for python module that will contain your configuration files. This means it should be only latin lowercase letters and (optionally) digits and underscore sign ("_"). Good idea is to use your domain name as source for project namespace, for example turning "misago-forum.org" into "misagoforumorg".

Once you've decided on your name, create your site configuration module. In example we assume your site will be named "misagoforumorg":

```
misago-start.py misagoforumorg
```

This will create directory "misagoforumorg" in your working directory. Inside you will find "manage.py" file that you can use to run administrative commands Misago provides as well as access its python shell which is usefull for quick and dirty administration work. In addition to this file you will find "cron.txt" that contains example crontab configuration for automating maintenance tasks on your site and "requirements.txt" that you can use as reference of versions of libraries Misago relies on to run. In addition to those, you will find one more "misagoforumorg" here, containing python module with configuration files for your site. We will get to it in a minute, but before that lets spend few more moments in our current location.

This directory has special purpose. It serves as "container" for your customizations for Misago. If you want to install extension or plugin that has no "setup.py" of its own or use custom styles or templates on your site, you will put them there, making them easily accessible for your Misago installation.

Let's go deeper. Change your current directory to "misagoforumorg". By default this directory will contain four files: "__init__.py", thats special file that tells python this directory is python package, "settings.py" that contains all low-level settings of your site, "urls.py" that tells your forum about links on your site and finally "wsgi.py", thats special file servers use to understand and talk with your application. Unless you are building entire site around your forum, you can ignore "urls.py".

Open "settings.py" in your code editor of choice and give a look in values listed here. Each value is accompanied by commentary explaining its purpose. See if any tuning is needed, then save your changes and leave editor.

Note: To simplify setup process, by default "settings.py" file contains only most basic settings that are needed for your site to run, with everything else being set for you automatically at the beginning of file.

Move back to directory with manage.py and use it to initialize Misago database by firing migrate:

```
python manage.py migrate
```

Next, call createsuperuser command to create super admin in database:

```
python manage.py createsuperuser
```

Finally, start development server using "runserver" command:

```
python manage.py runserver
```

If server starts, you should be able to visit 127.0.0.1:8000 in your browser and see forum index, however as work on project is underway revisions may frequently introduce changes that will break runserver.

Deployment

Deployment is a process in which you get your site running and reachable by your users.

Misago is de facto Django with extra features added. This means deployment of Misago should be largery same to deployment of other Django-based solutions. Django documentation already covers supported deployment methods, and while on dedicated and VPS options deployment method depends largery on your choice and employed software stack, shared servers may differ greatly by the way how Django should be deployed. If thats the case, make sure you consult your ISP documentation and/or ask its rep for details about supported deployment method.

Note: If you are deploying Misago on your own dedicated or virtual server, you may want to take interest in the Gunicorn that makes it easier to maintain Python deployments.

1.1.2 Updating to new version

Lorem ipsum dolor met.

1.1.3 Growing up with your community

Lorem ipsum dolor met.

1.2 Customizing and Extending Misago

Misago is being developed with customizability and extensibility on mind and offers many features for those looking into modifying its look and feel, extending its featurebase or changing behaviour of core features.

And if this isn't enough for you, Misago itself stands on shoulders of Django, powerful and battle-tested web framework with great documentation and rich ecosystem of additional modules (named "apps"). This means Misago is not "just forum solution". Its also complete framework you may use to build your entire site around, writing your own Django apps or installing and adapting one of thousands ready apps freely available in the internet.

No matter what you are trying to accomplish, Misago has you covered.

1.2.1 Customizing Appearance

Misago appearance is product of many different technologies acting together to produce final result that you and your users see in their browsers.

On server side simple and easy to learn but powerful Django template engine turns template files into final html that is displayed in browser while asset pipeline compiles, merges and optimizes less, css and javascript files making your browser load faster and relieving you from burden of compressing and optimizing them yourself.

On browser side Misago comes with it's own UI framework that uses Bootstrap, jQuery as well Glyphicons and Font Awesome as it's foundations that allows you to provide smooth, pleasant and modern experience to your site's users.

1.2.2 Writing Extensions

Misago extensions are called "apps". Each "app" can be either single feature like extra user profile tab or entire set of many features working together to add whole new part to your site like user blogs or galleries or file downloads.

To offer such level of extensibility many parts of Misago (and Django too!) were designed as simple and lightweight frameworks to which that you can add (or remove) features.

Following references cover everything you want to know about writing your own apps for Misago. From them you will learn what coding style, conventions and best practices you should follow when writing your code. Next you will be shown how to add your own pages and models to Misago and what features are available for you in doing so. Finally all available frameworks will be presented with explanation what they are used for and how you can add your features to them.

Permissions framework

Misago brings its own ACL (Access Control Lists) framework for implementing permissions and this document explains to how to use and extend it with your own permissions.

Checking permissions

Permissions are stored on special models named "roles" and assigned to users either directly or trough ranks. Guest users always have permissions from "Guest" role, and users always have permissions form "Member" role.

There are two kinds of objects in Misago: aware and unaware of their ACL's. Aware objects have "acl" annotation containing dict with their permissions for given ACL, while unaware objects don't. Instances of `User` and `AnonymousUser` classes are always ACL aware, while other objects need you to make them aware of their ACLs through use of `add_acl` functions ofered by `misago.acl` module. However this is not always needed (or possible), in which cases you have to introspect user's acl attribute directly.

ACL's are simple dictionaries, and their contents differ depending on objects they are belonging to. This means that to see if forum is visible to user, you have to perform following check:

```
if user.acl['forums'].get(forum.pk, {}).get('can_see'):
    # huzza, we can see forum!
```

Above snippet is edge example of checking forum permission, and luckily we have few alternatives:

```
if forum.pk in userl.acl['visible_forums']:
    # Not really shorter, but simpler to remember and works in django templates!

from misago.acl import add_acl

add_acl(user, forums)
for forum in forums
    if forum.acl['can_see']:
        # Now model instances in forums queryset are aware of their ACLs!
        # ACL's are easy to check in templates too now!
```

Because ACL framework is very flexible, different features can have different ways to check their permissions.

Misago comes with its own debug page titled "Misago User ACL" that is available from Django Debug Toolbar menu. This page display user roles permissions as well as final ACL assigned to current user.

Permissions cache Construction of User's ACLs can be costful process, especially once you start installing extensions adding new features to your site. Because of this, Misago is not assinging ACLs to Users, but to combinations of roles. This means that each individual uses has own "ACL key", that allows Misago to associate this user roles with valid ACL cache.

ACL's are cached in two places: in remote cache storage, for use between requests, and in thread memory, so you don't have to write your own caches and checks when you are checking multiple users ACL's during single request.

ACL cache is versioned and rebuilded when cache version is different than current ACL version, which happens when models being part of ACL framework are edited or deleted.

Extending permissions system

ACL framework extensions are modules registered in `MISAGO_ACL_EXTENSIONS` setting. By convention, those modules are either named "permissions", or they are located in "permissions" package.

Misago checks module for following functions:

change_permissions_form(*role*)

Required. This function is called when change permissions form for role is being build for view. It's expected to return Form type or none, if provider is not recognizing role type (eg. there is no sense in adding profiles visibility permissions to forums role form).

Note: Misago provides custom `YesNoSwitch` form field that renders nice "Yes/No" switch as input. This field is simple wrapper around `TypedChoiceField` that coerces to `int`. If you use use it for your permissions, make sure your ACL implementation handles their values as `1` or `0`, not as `True` or `False`, or your forms will break!

Warning: Make sure that all fields in your form have initial value, or your form will make tests suite fail because it will be unable to mock POST requests to admin forms correctly.

build_acl(*acl*, *roles*, *key_name*)

Required. Is used in process of building new ACL. Its supplied dict with incomplete ACL, list of user roles and name of key under which its permissions values are stored in roles `permissions` attributes. Its expected to access roles `permissions` attributes which are dicts of values coming from permission change forms and return updated `acl` dict.

register_with(*registry*)

Optional. Is called by providers registry after provider module was imported, to allow it to register annotators and serializers for ACL's. Receives only one argument:

- **registry** - istance of PermissionProviders that imported module.

Registering Annotators and Serializers

When module's `register_with` function is called, its passed `PermissionProviders` instance that exposes following methods:

acl_annotator(*hashable_type*, *func*)

Registers `func` as ACL annotator for `hashable_type`.

acl_serializer(*hashable_type*, *func*)

Registers `func` as ACL serializer for `hashable_type`.

get_type_annotators(*obj*)

Returns list of annotators registered for type of `obj` or empty list is none exist.

get_type_serializers(*obj*)

Returns list of serializers registered for type of `obj` or empty list is none exist.

Annotators Annotators are functions called when object is being made ACL aware. It always receives two arguments:

- **user** - user asking to make target aware of its ACL's
- **target** - target instance, guaranteed to be an single object, not list or other iterable (like queryset)

`target` has `acl` attribute which is dict with incomplete ACL that function should update with new keys.

Note: This will not work for instances of User model, that already reserve `acl` attribute for their own acls. Instead add_acl_to_target for User instances will add acl's to *acl_* attribute.

Serializers Serializers are functions called when ACL-aware object is being prepared for JSON serialization. Because python's `dict` type isnt 1:1 interchangeable with JSON, serializers allow ACL extensions to perform additional convertion or cleanup before model's ACL is serialized. They always receive single argument:

- **serialized_acl** - ACL that will be JSON serialized

Example serializer for extension setting dict using integers for keys could for example remove this dictionary from ACL to avoid problems during ACL serialization:

```
def serialize_forums_acl(user_acl):
    user_acl.pop('forums', None)
```

Algebra

Consider those three simple permission sets:

```
roles_permissions = (
    {'can_be_knight': False},
    {'can_be_knight': True},
    {'can_be_knight': False},
)
```

In order to obtain final ACL, one or more ACLs have to be sum together. Such operation requires loop over ACLs which compares values of dicts keys and picks preffered ones.

This problem can be solved using simple implementation:

```
final_acl = {'can_be_knight': False}

for acl in roles_permissions:
    if acl['can_be_knight']:
        final_acl['can_be_knight'] = True
```

But what if there are 20 permissions in ACL? Or if we are comparing numbers? What if complex rules are involved like popular "greater beats lower, zero beats all" in comparisions? This brings need for more suffisticated solution and Misago provides one in forum of `misago.acl.algebra` module.

This module provides utilities for summing two acls and supports three most common comparisions found in web apps:

- **greater**: True beats False, 42 beats 13
- **lower**: False beats True, 13 beats 42
- **greater or zero**: 42 beats 13, zero beats everything

- **lower non zero**, 13 beats 42, everything beats zero

sum_acls(*result_acl*, *acls=None*, *roles=None*, *key=None*, ***permissions*)

This function adds ACLs to result_acl using set or rules provided as additional kwargs. Alternatively, it access iterable of roles and extension key.

Example usage is following:

```
from misago.acl import algebra

user_acls = [
    {
        'can_see': 0,
        'can_hear': 0,
        'max_speed': 10,
        'min_age': 16,
        'speed_limit': 50,
    },
    {
        'can_see': 1,
        'can_hear': 0,
        'max_speed': 40,
        'min_age': 20,
        'speed_limit': 0,
    },
    {
        'can_see': 0,
        'can_hear': 1,
        'max_speed': 80,
        'min_age': 18,
        'speed_limit': 40,
    },
]

defaults = {
    'can_see': 0,
    'can_hear': 0,
    'max_speed': 30,
    'min_age': 18,
    'speed_limit': 60,
}

final_acl = algebra.sum_acls(
    defaults, acls=user_acls,
    can_see=algebra.greater,
    can_hear=algebra.greater,
    max_speed=algebra.greater,
    min_age=algebra.lower,
    speed_limit=algebra.greater_or_zero
    )
```

As you can see because tests are callables, its easy to extend `sum_acls` support for new tests specific for your ACLs.

Writing New Admin Actions

Misago Admin vs. Django Admin

Misago brings its own admin site just like Django does. This means you have to make a decision which one your app will use for administration.

If you intend to be sole user of your app, Django admin will propably be faster to get going. However if you plan for your app to be available to wider audience, its good for your admin interface to be part of Misago admin site. This will require you to write more code than in case you've went Django way, but will give your users more consistent experience and, in case for some languages, save them of quirkyness that comes with Django admin automatically created messages.

Creating Admin Views

Writing views Unlike Django, Misago admin is not "automagical". This means you will not get complete admin from nowhere by just creating one file and writing 3 lines of code in it. However Misago provides set of basic classes defined in in `misago.admin.views.generic` module that can offload most of burden of writing views handling items lists and forms from you.

Workflow with those classes is fast and easy to master. First, you define your own mixin (propably extending `AdminBaseMixin`). This mixin will define common properties and behaviour of all admin views, like which Model are admin views focused on, how to fetch its instances from database as well as where to seek templates and which message should be used when model could not be found.

Next you define your own views inheriting from your mixin and base views. Misago provides basic views for each of most common scenarios in admin:

- **ListView** - For items lists. Supports pagination, sorting, filtering and mass actions.
- **FormView** and **ModelFormView** - For displaying and handling forms submissions.
- **ButtonView** - For handling state-changing button presses like "delete item" actions.

AdminBaseMixin `misago.admin.views.generic.AdminBaseMixin`

Base class for admin mixins that contain properties and behaviours shared between admin views. While you are allowed to set any properties and function on your own mixins to dry your admin views more, bare minimum expected from you is following:

- **Model** property or **get_model(self)** function used to get model type.
- **root_link** property that is string with link name for "index" view for admin actions (usually link to items list).
- **templates_dir** property being string with name of directory with admin templates used by mixin views.

Optionally if you don't plan to set up action-specific item not found messages, you may set `message_404` property on mixin to make all your views use same message when requested model could not be found.

ListView `misago.admin.views.generic.ListView`

Base class for lists if items. Supports following properties:

- **template** - name of template file located in `templates_dir` used to render this view. Defaults to `list.html`
- **items_per_page** - integer controlling number of items displayed on single page. Defaults to 0 which means no pagination

- **SearchForm** - Form type used to construct form for filtering this list. Either this field or `get_search_form` method is required to make list searchable.
- **ordering** - list of supported sorting methods. List of tuples. Each tuple should countain two items: name of ordering method (eg. "Usernames, descending") and `order_by` argument ("-username"). Defaults to none which means queryset will not be ordered. If contains only one element, queryset is ordered, but option for changing ordering method is not displayed.
- **mass_actions** - list of dicts defining list's mass actions. Each dict should have `action` key that will be used to identify method to call, `name` for displayed name, `icon` for icon and optional `confirmation` message. Actions can define optional "is_atomic" key to control if they should be wrapped in transaction or not. This is default behaviour for mass actions.
- **selection_label** - Label displayed on mass action button if there are items selected. `0` will be replaced with number of selected items automatically.
- **empty_selection_label** - Label displayed on mass action button if there are no items selected.

In addition to this, ListView defines following methods that you may be interested in:

`get_queryset`(*self*)

This function is expected to return queryset of items that will be displayed. If filters, sorting or pagination is defined, this queryset will be further sliced and filtered.

`add_item_action`(*cls*, *name*, *icon*, *link*, *style=None*)

Class method that allows you to add custom links to item actions. Link should be a string with link name, not complete link. It should also accept same kwargs as other item actions links.

`add_item_action`(*cls*, *action*, *name*, *prompt=None*)

Class method that allows you to add custom mass action. Action should be name of list method that will be called for this action. Name will be used for button label and optional prompt will be used in JavaScript confirmation dialog that will appear when user clicks button.

`get_search_form(self, request):`

This function is used to get search form class that will be used to construct form for searching list items.

If you decide to make your list searchable, remember that your Form must meet following requirements:

- Must define `filter_queryset(self, search_criteria, queryset)` method that will be passed unfiltered queryset, which it should modify using filter/exclude clauses and data from search_criteria.
- Must return queryset.
- Must not define fields that use models for values.

If you add custom mass action to view, besides adding new entry to `mass_actions` list, you have to define custom method following this definition:

`action_ACTION`(*self*, *request*, *items*)

`ACTION` will be replaced with action dict `action` value. Request is `HttpRequest` instance used to call view and `items` is queryset with items selected for this action. This method should return nothing or `HttpResponse`. If you need to, you can raise `MassActionError` with error message as its first argument to interrupt mass action handler.

FormView `misago.admin.views.generic.FormView`

Base class for forms views.

- **template** - name of template file located in `templates_dir` used to render this view. Defaults to `form.html`

- **Form** property or **create_form_type** method - `create_form` method is called with `request` as its argument and is expected to return form type that will be used by view. If you need to build form type dynamically, instead of defining `Form` property, define your own `create_form`.

create_form_type(*self*, *request*)

Returns form type that will be used to create form instance. By default returns value of `Form` property.

initialize_form(*self*, *FormType*, *request*)

Initializes either bound or unbound form using request and `FormType` provided.

handle_form(*self*, *form*, *request*)

If form validated successfully, this method is called to perform action. Here you should place code that will read data from form, perform actions on models and set result message. Optionally you may return `HttpResponse` from this function. If nothing is returned, view returns redirect to `root_link`.

Optionally your form template may have button with `name="stay"` attribute defined, pressing which will cause view to redirect you to clean form instead.

ModelFormView `misago.admin.views.generic.ModelFormView`

Base class for targetted forms views. Its API is largery identic to `FormView`, except it's tailored at handling `ModelForm` and modifying model states. All methos documented for `FormView` are present in `ModelformView`, but they accept one more argument named "target", containing model instance to which model form will be tied.

In addition, this view comes with basic definition for form handler that calls `save()` on model instance and (if defined) sets success message using value of objects `message_submit` parameter.

ButtonView `misago.admin.views.generic.ButtonView`

Base class for handling non-form based POST requests.

Do control this view behaviour, define your own `button_action` method:

button_action(*self*, *request*, *target*)

This function is expected to perform requested action on target provided and set result message on `request`.

It may return nothing or `HttpResponse`. If nothing is returned, view returns redirect to `root_link` instead.

Targeted views Both `ModelFormView` and `ButtonView` are called "targeted views", because they are expected to manipulate model instances. They both inherit from `TargetedView` view, implements simple API that is used for associating request with corresponding model instance:

get_target_or_none(*self*, *request*, *kwargs*)

Function expected return valid model instance or None. If None is returned, this makes view set error message using `message_404` attribute and returns redirect to `root_link`.

get_target(*self*, *kwargs*)

Called by `get_target_or_none`.

If `kwargs` len is 1, its assumed to be value of seeked model pk value. This makes function call model manager `get()` method to fetch model instance from database. Otherwhise "empty" instance is created and returned instead. Eventual `DoesNotExist` errors are handled by `get_target_or_none`.

check_permissions(*self*, *request*, *target*)

Once model instance is obtained either from database or empty instance is created, this function is called to see intended action is allowed for this request and target. This function is expected to return `None` if no issues are found or string containing error message. If string is returned, its set as error messages, and view interrupts its execution by returning redirect to `root_link`.

Note: While target argument value is always present, you don't have to do anything with it if its not making any sense for your view.

In addition, views are wrapped in database transaction. To turn this behaviour off, define `is_atomic` attribute with value `False`.

Adding extra values to context Each view calls its `process_context` method before rendering template to response. This method accepts two arguments:

- **request** - HttpRequest instance received by view.
- **context** - Dict that is going to be used to render template.

It's required to return dict that will be then used as one of arguments to call `render()`.

Registering in Misago Admin

Misago Admin Site is just an hierarchy of pages, made of two parts: `site` that contains tree of links and `urlpatterns` that is included in `misago:admin` namespace.

When Misago is started, it scans registered apps for `admin` module, just like Django admin does. If module is found, Misago checks if it defines `MisagoAdminExtension` class. If such class is found, its instantiated with no arguments, and two of its methods are called:

`register_urlpatterns`(*self*, *urlpatterns*)

This function allows apps to register new urlpatterns under `misago:admin` namespace.

`register_navigation_nodes`(*self*, *site*)

This function allows apps to register new links in admin site navigation.

Registering urls under `misago:admin` namespace Admin links are stored within instance of special object `misago.admin.urlpatterns.URLPatterns` available as `urlpatterns` argument passed to `register_urlpatterns` method. This object exposes two methods as public api:

`namespace`(*path*, *namespace*, *parent=None*)

Registers new namespace in admin links hierarchy.

- **path** - Path prefix for links within this namespace. For example `r'^users/'`.
- **namespace** - Non-prefixed (eg. without `misago:admin` part) namespace name.
- **parent** - Optional. Name of parent namespace (eg. `users:accounts`).

`patterns`(*namespace*, **urlpatterns*)

Registers urlpatterns under defined namespace. Expects first argument to be name of namespace that defined links belong to (eg. `users:accounts`). Every next argument is expected to be valid Django link created with `url` function from `django.conf.urls` module.

Note: ''misago:admin" prefix of namespaces is implicit. Do not prefix namespaces passed as arguments to those functions with it.

Registering urls in navigation Your urls have to be discoverable by your users. Easiest way is to do this is to display primary link to your admin action in admin site navigation.

This navigation is controlled by instance of the `misago.admin.hierarchy.AdminHierarchyBuilder` class available as `site` argument passed to `register_navigation_nodes` method of your `MisagoAdminExtension` class. It has plenty of functions, but it's public api consists of one method:

add_node(*name=None*, *icon=None*, *parent='misago:admin'*, *after=None*, *before=None*, *namespace=None*, *link=None*)

This method accepts following named arguments:

- **parent** - Name of parent namespace under which this action link is displayed.
- **after** - Link before which one this one should be displayed.
- **before** - Link after which one this one should be displayed.
- **namespace** - This link namespace.
- **link** - Link name.
- **name** - Link title.
- **icon** - Link icon (both Glyphicons and Font Awesome are supported).

Only last three arguments are required. `after` and `before` arguments are exclusive. If you specify both, this will result in an error.

Misago Admin supports three levels of hierarchy. Each level should corelate to new namespace nested under `misago:admin`. Depending on complexity of your app's admin, it can define links that are one level deep, or three levels deep.

Adding actions to items lists Other way to make your views reachable is to include links to them on items lists. To do this, you may use `add_item_action` classmethod of ListView class that is documented above.

User Authentication

Admin Authentication

Misago adds additional layer of security around admin areas in your site. This means that unless you've signed to admin area directly, you have to authenticate yourself one more time to upgrade your session from "casual" one to "administrator".

This mechanism was put in place because it's common for forum administrators to browse and use forums while signed on their administrator account. By default, Django requires user to be signed in and have special `is_staff` set on his or her account and know the path to administration backend to administrate site, which is good approach for situations when staff accounts are used exclusively for administration and not day to day usage.

In addition for re-authentication requirement, Misago also monitors inactivity periods between requests to admin interfaces, and if one exceeds length specified in `MISAGO_ADMIN_SESSION_EXPIRATION` setting, it will assume that administrator has been inactive and request another reauthentication upon next request to admin backend.

Implementation in this mechanism is placed within `misago.admin.auth` module and `misago.admin.middleware.AdminAuthMiddleware` middleware. Middleware uses methods from `auth` to detect if request is pointed at protected namespace, and if it is, it uses facilities to handle and control state of administrators session.

`misago.admin.auth.`**`is_admin_session`**(*request*)

Returns true if current request has valid administrator session. Otherwhise returns false.

`misago.admin.auth.`**`start_admin_session`**(*request*, *user*)

Promotes current session to state of administrator session.

`misago.admin.auth.`**`update_admin_session`**(*request*)

Updates last activity timestamp on admin session.

`misago.admin.auth.`**`close_admin_session`**(*request*)

Closes current admin session, degrading it to "casual" session and keeps user signed in.

Testing Admin views using test client To test protected admin views from within your test cases, you have to open valid admin session for test client. Misago provides *`misago.admin.testutils.admin_login()`* function for this purpose.

`misago.admin.testutils.`**`admin_login`**(*client*, *username*, *password*)

This function will make provided test client instance use valid admin session during test requests. Note that internally this function makes POST request to `misago:admin:index` link that should result with admin login form for unauthenticated users.

Misago Caches

You can make Misago use its own cache instead of sharing cache with rest of your Django site. To do so, add new cache named `misago` to your `CACHES` setting:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    },
    'misago': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11212',
    }
}
```

Full-Page Cache

By default Full-Page Cache ("FPC") uses same cache other Misago features do, however on active site this may cause your cache backend to frequently delete other still valid caches if it runs out of space.

To avert this, you can define one more cache named `misago_fpc`.

Cache Buster

Cache buster is small feature that allows certain cache-based systems find out when data they were dependant on has been changed, making their cache no longer valid.

Using Cache Buster

Cache buster lives in `misago.core.cachebuster` and provides following API:

is_valid
`is_valid`(*cache*, *version*)
Checks if specific cache version is valid or raises `ValueError` if cache key is invalid.

get_version
`get_version`(*cache*)
Returns current valid cache version as an integer number or raises `ValueError` if cache key is invalid.

invalidate
`invalidate`(*cache*)
Makes specified cache invalid.

invalidate_all
`invalidate_all`()
Makes all versioned caches invalid.

Adding Custom Cache Buster

You may add and remove your own cache names to cache buster by using following commands:

Note: Don't forget to call *invalidate_all* function after adding or removing cache name from buster to force it to rebuild its own cache.

---

register
`register`(*cache*)
Registers new cache in cache buster for tracking.

unregister
`unregister`(*cache*)
Removes cache from cache buster and disables its tracking. This function will raise `ValueError` if cache you are trying to unregister is not registered.

Coding Style and Conventions

When writing Python code for Misago, please familiarize yourself with and follow those documents:

1. PEP 8
2. Django Coding Style and Conventions

Those documents should give you solid knowledge of coding style and conventions that are followed by Python and Django programmers when writing code.

In addition to those guidelines, Misago defines set of additional convetions and good practices that will help you write better and easier code.

Note: Originally Misago was writen with 79 character line limit in mind, but recently this convention was dropped for sake of Django's 119 characters line limit.

Note: Misago comes with ".pylintrc" file that contains configuration for pylint tool used to lint Misago's codebase.

Models

Fields Order When declaring model's database fields, start with taxonomical foreign keys followed by fields that make this model identifiable to humans. Order of remaining fileds is completely up to you.

Thread model is great example of this convention. Thread is part of taxonomy (Forum), so first field defined is foreign key to Forum model. This is followed by two fields that humans will use to recognise this model: "title" that will be displayed in UI and "slug" that will be included in links to this thread. After those Thread model defines additional fields.

When declaring model database fields, make sure they are grouped together based on their purpose. Most common example is storage of user name and slug in addition to foreign key to User model. In such case, make sure both "poster" field as well as "poster_name", "poster_slug" and "poster_ip" fields are grouped together.

Avoid Unmeaningful Names Whenever possible avoid naming fields representing relation to "User" model "user". Preffer more descriptive names like "poster", "last_editor", or "giver".

For same reason avoid using "date" or "ip" as field names. Use more descriptive "posted_on" or "poster_ip" instead.

True/False Fields For extra clarity prefix fields representing true/false states of model with "is". "is_deleted" is better than "deleted".

Serializers

Defining serializers Model's default serializer should be named after its model, but with "Serializer" suffix, ergo serializer for "Thread" model should be named "ThreadSerializer". Default serializer should be only serializer defining serialization methods.

In case that model has more than one serializer, all serializers should inherit from default one and be named in a way describing it usage. For example serializer for "User" model used to serialize post's author should be named "PostPosterSerializer". This serializer should only define "Meta" class with "model" and "fields" attributes, inheriting serialization behaviour form default serializer.

Fields Order Order of fields should correspond directly to order of fields on model that serializer handles.

If serializer defines non-model fields, those should be specified last and separated from model's fields with empty line, ergo:

```
fields = (
    'id',
    'user',
    'title',

    'acl', # annotation set on model by view
```

```
    'something_extra', # dynamic atribute coming from "SerializerMethodField"
)
```

URLs Serializers may define two special fields used for serialization of url's, "url" and "api". The first one should be string containing serialized model's "get_absolute_url" or list urls of interest for UI rendering serialized model, like:

```
'url': {
    'absolute': obj.get_absolute_url(),
    'first_unread': obj.get_first_unread_url(),
    'last_post': obj.get_last_post_url(),
}
```

Likewise the "api" key should contain the url to item api endpoint (eg. "/api/threads/132/") or list of avaiable endpoints:

```
'api': {
    'index': reverse('misago:api:threads', kwargs={'pk': obj.pk}),
    'read': reverse('misago:api:thread-read', kwargs={'pk': obj.pk}),
    'move': reverse('misago:api:thread-move', kwargs={'pk': obj.pk}),
}
```

Those keys should live at the end of the fields list and be separated from other fields with blank line:

```
fields = (
    'id',
    'user',
    'title',

    'acl', # annotation set on model by view
    'something_extra', # dynamic atribute coming from "SerializerMethodField"

    'api',
    'url',
)
```

Nested results Nested results should be included in view or viewset, as part of creding dict of serialized data for "Response" object:

```
data = UserSerializer(user).data
data['post_set'] = UserPostSerializer(posts, many=True).data
return Response(data)
```

The added key should be model's "related_name" in respect of model it annotates (defautly its "modelname_set").

Context Processors

Context Processors are simple python functions that receive HttpRequest object and extend template context with additional values. In addition to default context processors defined by Django, Misago defines its own context processors:

misago.core.context_processors.site_address

`misago.core.context_processors.site_address()`

This function adds `SITE_ADDRESS` value to template context that you can use to build absolue links in your templates:

```
# Will become "http://mysite.com/"
{{ SITE_ADDRESS }}{% url 'forum_index' %}
```

This is most useful for links in e-mail templates.

Views Decorators

Misago apps define plenty of decorators for you to wrap your views with.

misago.core.decorators

require_POST `misago.core.decorators.require_POST()`

Function that checks if request made to access view is POST. If it's not, decorator renders ''wrong_way.html" template and returns it in 405-coded response. This is its only difference to Django's counterpart.

misago.core.errorpages

shared_403_exception_handler `misago.core.errorpages.shared_403_exception_handler()`

If your project has different error handler for 403 errors defined, you can use this decorator to make your function shared handler between Misago and other views. This will make Misago handle 403 exceptions coming from under its path while leaving alone exceptions coming from your site.

shared_404_exception_handler `misago.core.errorpages.shared_404_exception_handler()`

Same as above but for custom 404 error handler.

misago.users.decorators

deny_authenticated `misago.users.decorators.deny_authenticated()`

This decorator will block requests made to view if user is authenticated, displaying page with error message or returning JSON/XML in its stead.

deny_guests `misago.users.decorators.deny_guests()`

This decorator will block requests made to view if user is not authenticated, displaying page with error message or returning JSON/XML in its stead.

deny_banned_ips `misago.users.decorators.deny_banned_ips()`

This decorator will block requests made to view if user IP is banned, displaying page with error message or returning JSON/XML in its stead.

deflect_authenticated `misago.users.decorators.deflect_authenticated()`

This decorator will return redirect to forum index if user is authenticated.

deflect_guests `misago.users.decorators.deflect_guests()`

This decorator will return redirect to forum index if user is not authenticated.

deflect_banned_ips `misago.users.decorators.deflect_banned_ips()`

This decorator will return redirect to forum index if user IP is banned.

Creating Forms in Misago

Note: Purpose of this document is to introduce you to differences and features of forms module available in misago.core.forms package.

For proper introduction to forums, see Django documentation.

Misago extends standard forms library that comes with Django with extra components. First one is Crispy Forms app that allows to use templates to display forms. Thanks to this feature, it's possible to render correct markup that Bootstrap requires with minimal amount of effort.

Misago Forms Module

Second extension lies in `misago.core.forms` module that imports Django forms. This layer introduces tiny convenience change to forms standard clean and validate behaviour that will save you from some "gotchas" associated with input length validation. Unlike other popular frameworks and solutions, Django does not automatically trim input values received from client before validation, which means user can enter few spaces into text fields instead of "readable" value and such input will pass automatical validation and will need additional cleanup and checks in custom `clean_` methods. Because there's good chance your app will have plenty of `CharField` fields, such boilerplate can quickly add up.

In `misago.core.forms` you will find three classes:

AutoStripWhitespacesMixin `misago.core.forms.AutoStripWhitespacesMixin`

Small mixin that strips whitespaces from all `CharField`'s input on form, before its passed further for validation.

If you wish to exclude one or more fields from this behaviour, you may add their names to "autostrip_exclude" attribute:

```
class MyLargeForm(Form):
    autostrip_exclude = ['i_wont_be_stripped']
    i_will_be_stripped = forms.CharField()
    i_wont_be_stripped = forms.CharField()
```

Form `misago.core.forms.Form`

Wrapper for `django.forms.Form` that uses AutoStripWhitespacesMixin.

ModelForm `misago.core.forms.ModelForm`

Wrapper for `django.forms.ModelForm` that uses AutoStripWhitespacesMixin.

YesNoSwitch `misago.core.forms.YesNoSwitch()`

Thin wrapper around Django's `TypedChoiceField`. This field renders nice yes/no switch as its input.

> **Warning:** `YesNoSwitch` coerces to `int`, not to `bool`! Remember about this when writing code dealing with forms containing this field!

Forums Forms Module

`misago.forums.forms` module defines two fields you may use for making forum selections in your forms:

Because those fields need to know ACL, you are required to call their `set_acl` method from your form's `__init__`:

```
class MoveThreadsForm(forms.Form):
    new_forum = ForumChoiceField(label=_("Move threads to forum"))

    def __init__(self, *args, **kwargs):
        self.forum = kwargs.pop('forum')
        acl = kwargs.pop('acl')

        super(MoveThreadsForm, self).__init__(*args, **kwargs)

        self.fields['new_forum'].set_acl(acl)
```

ForumChoiceField Extends `ModelChoiceField`.

ForumsMultipleChoiceField Extends `ModelMultipleChoiceField`.

Template Tags

Misago defines custom templates extension named `misago_forms`. This extension contains two template tags for rendering form fields:

form_row This tag takes form field as its first argument and renders field complete with label, help and errors. Accept two extra arguments: label class and field class, allowing you to control size of horizontal forms:

```
{% load misago_forms %}

{% form_row form.somefield %}
{% form_row form.otherfield 'col-md-3' 'col-md-9' %}
```

form_input This tag takes form field as its only argument and renders it's input.

Sending Mails

Misago provides its own API for sending e-mails to forum users that extends standard Django mailer.

This API lives in `misago.core.mail` and provides following functions:

build_mail

build_mail(*request*, *recipient*, *subject*, *template*, *context=None*)

Build e-mail message using supplied template name and (optionally) context. Template name shouldn't contain file extension, as Misago will automatically append `.html` for html content and `.txt` for plaintext content of the message. Message templates will have access to same request context as other templates, additional context you've provided and two extra context values: `recipient` and `sender`.

- `request`: HttpRequest object instance.
- `recipient`: User model instance.
- `subject`: A string.
- `template`: A string.
- `context`: The optional dictionary with extra context values that should be available for message templates.

mail_user

mail_user(*request*, *recipient*, *subject*, *template*, *context=None*)

Shortcut function that calls `build_mail` to build message, then sends it to user.

- `request`: HttpRequest object instance.
- `recipient`: User model instance.
- `subject`: A string.
- `template`: A string.
- `context`: The optional dictionary with extra context values that should be available for message templates.

mail_users

mail_users(*request*, *recipients*, *subject*, *template*, *context=None*)

Same as above, but instead of sending message to one recipient, it sends it to many recipients at same time. Keep on mind this may be memory intensitive as this function creates one Mail object instance for every recipient specified, so you may want to split recipients into smaller groups as you are sending them emails.

- `request`: HttpRequest object instance.
- `recipients`: Iterable of User models.
- `subject`: A string.
- `template`: A string.
- `context`: The optional dictionary with extra context values that should be available for message templates.

Misago Markup

Misago defines custom `misago.markup` module that provides facilities for parsing strings.

This module exposes following functions as its public API:

parse

parse(*text*, *author=None*, *allow_mentions=True*, *allow_links=True*, *allow_images=True*, *allow_blocks=True*)

Parses Misago-flavoured Markdown text according to settings provided. Returns dictionary with following keys:

- `original_text`: original text that was parsed
- `parsed_text`: parsed text
- `markdown`: markdown instance

common_flavour

common_flavour(*text*, *author=None*, *allow_mentions=True*)

Convenience function that wraps `parse()`. This function is used for parsing messages.

Extending Markup To extend Misago markup, create custom module defining one or both of following functions:

extend_markdown(*md*)

Defining this function will allow you to register new extensions in markdown used to parse text.

process_result(*result*, *soup*)

This function is called to allow additional changes in result dict as well as extra instrospection and cleanup of parsed text, which is provided as Beautiful Soup class instance.

Both functions should modify provided arguments in place.

Once your functions are done, add path to your module to `MISAGO_MARKUP_EXTENSIONS` setting which is tupe of modules.

Notifications

Modern site in which users interact with each other needs quick and efficient notifications system to let users know of each other actions as quickly as possible.

Misago implements such system and exposes simple as part of it, located in `misago.notifications`:

notify_user

notify_user(*user*, *message*, *url*, *type*, *formats=None*, *sender=None*, *update_user=True*)

- `user`: User to notify.
- `message`: Notification message.
- `url`: Link user should follow to read message.
- `type`: short text used to identify this message for `read_user_notifications` function. For example `see_thread_123` notification will be read when user sees thread with ID 123 for first time.
- `formats`: Optional. Dict of formats for `message` argument that should be boldened.
- `sender`: Optional. User that notification origins from.

- update_user: Optional. Boolean controlling if to call user.update after setting notification, or not. Defaults to True.

read_user_notifications

read_user_notifications(*user*, *types*, *atomic=True*)

Sets user notifications identified by types as read. This function checks internally if user has new notifications before it queries database.

- user: User to whom notification belongs to
- types: Short text or list of short texts used to identify notifications that will be set as read.
- atomic: Lets you control if you should wrap this in dedicated transaction.

Posting Process

Process of posting and editing messages in Misago is fully modular and extensible.

Whenever there is a need to allow user to post or edit existing message, special pipeline object is initialized and passed HttpRequest object istance as well as instances of current user, forum, thread, post and post you are replying to. This pipeline then instiates classes defined in MISAGO_POSTING_MIDDLEWARES setting.

This pipeline then asks its middlewares if they have any forms they want to display on posting page, or if they have any actions to perform before, on, or after save to database.

Pipeline itself performs no changes in models or database state, delegating all tasks to its middlewares.

Pipeline is also isolated in database transaction with locks on database rows taking part in posting process. This means that its protected from eventual race conditions and in cause of eventual errors or interruptions, can be rolled back without any downsides.

Note: Attachments and any other extra data stored outside of Database that was changed before rollback was called is excluded from this.

Writing custom middlewares

Middlewares are classes extending PostingMiddleware class defined in misago.threads.posting package.

During pipeline's initialization each class defined in MISAGO_POSTING_MIDDLEWARES setting is initialized and passed current context, which is then stored on its object attributes:

- mode: Integer used to identify type of action. You can compare it with START, REPLY and EDIT constants defined in misago.threads.posting package.
- request: HttpRequest using pipeline.
- user: Authenticated user writing message.
- forum: Forum in which message will be posted.
- thread: Depending on mode and phase in posting process, this may be thread loaded from database, or empty model that is yet to be saved in database.

- `post`: Depending on `mode` and phase in posting process, this may be post loaded from database, or empty model that is yet to be saved in database.
- `quote`: If `mode` is `REPLY`, this may be `None` or instance of message to which reply is written by `user`.
- `datetime`: Instance of `datetime` returned by `django.utils.timezone.now` call. You can use it to avoid subtle differences betwen dates stored on different models generated by new calls to `timezone.now`.
- `parsing_result`: dictionary containing result of parsing message entered by user.

After middleware object has been created, its `use_this_middleware` method is called with no arguments to let middleware make decision if it wants to participate in process and return `True` or `False` accordingly. If you don't define this method yourself, default definition always returns `True`.

In addition to `use_this_middleware` method mentioned ealier, each middleware may define following methods to participate in different phases of posting process:

make_form
`make_form`()
Allows middlewares to display and bind to data their own forms. This function is expected to return None (default behaviour) or Form class instance.

Forms returned by middlewares need to have two special attributes:

- `legend`: Name of form (eg. "Poll").
- `template`: String with path to template that will be used to render this form.

In addition to this form has to declare itself as either main or supporting form via defining `is_main` or `is_supporting` attribute with value being `True`.

If form requires inclusion of JavaScript to work, `js_template` may be defined with path to template to be rendered before document's `</body>`.

pre_save
`pre_save`(*form*)
This method is called with either middleware's form instance or `None` before any state was saved to database.

save
`save`(*form*)
This method is called with either middleware's form instance or `None` to save state to database.

post_save
`post_save`(*form*)
This method is called with either middleware's form instance or `None` to perform additional actions like sending notifications, etc. ect..

Reducing number of database saves

Misago provides custom `SaveChangesMiddleware` middleware that allows other middlewares to combine database saves. This middleware is called at the end of `save` and `post_save` phrases, meaning its possible for other middlewares to delegate saves to it during any time of posting process.

This middleware works by defining two extra attributes on `User`, `Forum`, `Thread` and `Post` models: `update_all` and `update_fields`.

If middleware made changes to many fields on model, changing it's `update_all` attribute to `True` will make `SaveChangesMiddleware` call model's `save` method with no arguments at the end of either `save` or `post_save` phase.

However if middleware changed only few fields, it may be better to append their names to model's `update_fields` attribute instead.

When different middlewares add custom fields to `update_fields` and set `update_all` flag at same time, Misago will perform one `save()` call updating whole model.

Note: Do not make assumptions or "piggyback" on other middlewares save orders. Introspecting either `update_all` or `update_fields` is bad practice. If your middlware wants its changes saved to database, it should add changed fields names to `update_fields` list even if it already contains them, or set `update_all` to `True`.

Interrupting posting process from middleware

Middlewares can always interrupt (and rollback) posting process during `interrupt_posting` phrase by raising `misago.threads.posting.PostingInterrupt` exception with error message as its only argument.

All `PostingInterrupt` raised outside that phrase will be escalated to `ValueError` that will result in 500 error response from Misago. However as this will happen inside database transaction, there is chance that this has caused no data loss in process.

Preloading Data for Mithril.js Components

When user visits Misago site for first time, his/hers HTTP request is handled by Django which outputs basic version of requested page. If user has JavaScript enabled in browser, blank spaces are then filled by the Mithril.js components.

To enable those components easy access to application's state, Misago provides simple "frontend context".

Exposing Data to Mithril.js

Misago creates empty dict and makes it available as `frontend_context` attribute on current `HttpRequest` object. This dict is converted into JSON when initial page is rendered by Django.

This means that as long as initial page wasn't rendered yet, you can preload data in any place that has access to request object, but for performance reasons views and context processors are prefereable place to do this.

Accessing Preloaded Data

The data exposed to Mithril.js by Misago lives in plain JavaScript object available as `context` attribute on services containter instance.

Settings

Accessing Settings

Misago splits its settings into two groups:

- **Low level settings** - those settings must be available when Misago starts or control resources usage and shouldn't be changed frequently from admin control panel. Those settings live in `settings.py`
- **High level settings** - those settings are stored in database and can be changed on runtime using interface provided by admin control panel.

Both types of settings can accessed as attributes of `misago.conf.settings` object and high level settings can be also accessed from your templates as attributes of `misago_settings` context value.

Note: Not all high level settings values are available at all times. Some settings ("lazy settings"), are evaluated to `True` or `None` immediately upon load. This means that they can be checked to see if they have value or not, but require you to use special `get_lazy_setting(setting)` getter to obtain their real value.

Defining Custom DB Settings

Note: Current Misago 0.6 migrations are south-based placeholders that will be replaced with new migrations introduced in Django 1.7 before release. For this reason this instruction focuses exclusively on usage of utility function provided by Misago.

In order to define or change high-level (stored in database) settings you have to add new rows to `conf_settingsgroup` and `conf_settings` database tables. This can be done by plenty of different ways, but preffered one is by creating new data migration and using functions from `misago.conf.migrationutils` module.

migrate_settings_group

`migrate_settings_group`(*orm*, *group_fixture*, *old_group_key=None*)

This function uses south supplied ORM instance to insert/update settings group in database according to provided dict contianing its name, description and contained settings. If new group should replace old one, you can provide its key in `old_group_key` argument.

The `group_fixture` dict should define following keys:

- **key** - string with settings group key.
- **name** - string with settings group name.
- **description** - optional string with short settings group description.
- **settings** - tuple containing dics describing settings belonging to this group.

Each dict in `settings` tuple should define following keys:

- **setting** - string with internal setting name.
- **name** - string with displayed setting name.
- **description** - optional string with small setting help message.
- **legend** - optional string indicating that before displaying this setting in form, new fieldset should be opened and this value should be used in its legend element.
- **python_type** - optional string defining type of setting's value. Can be either "string", "int", "bool" or "list". If omitted "string" is used by default.
- **value** - list, integer or string with default value for this setting.

- **default_value** - if your setting should always have value, specify there fallabck value used if no user-defined value is available.
- **is_public** - public settings are included JSON that is passed to Ember.js application. Defaults to `False`.
- **is_lazy** - if setting value may too large to be always loaded into memory, you may make setting lazily loaded by defining this key with `True` value assigned.
- **form_field** - What form field should be used to change this setting. Can be either "text", "textarea", "select", "radio", "yesno" or "checkbox". If not defined, "text" is used. "checkbox" should be used exclusively for multiple choices list.
- **field_extra** - dict that defines extra attributes of form field. For "select", "radio" and "checkbox" fields this dict should contain "choices" key with tuple of tuples that will be used for choices in input. For "string" settings you can define "min_length" and "max_length" extra specifying minmal and maximal lenght of entered text. For integer settings you can specify minimal and maximal range in which value should fall by "min_value" and "max_value". "textarea" field supports extra "rows" setting that controls generated textarea rows attribute. All text-based fields accept "required" setting. "checkbox" field supports "min" and "max" values that control minimum and maximum required choices.

Note: If you wish to make your names and messages translateable, you should use `ugettext_lazy` function provided by Misago instead of Django one. This function is defined in `misago.core.migrationutils` module and differs from Django one by the fact that it preserves untranslated message on its `message` attribute.

For your convience `migrate_settings_group` triess to switch translation messages with their "message" attribute when it writes to database and thus making their translation to new languages in future possible.

---

with_conf_models

`with_conf_models`(*migration*, *this_migration=None*)

South migrations define special `models` attribute that holds dict representing structure of database at time of migration execution. This dict will by default contain only your apps models. To add settings models that `migrate_settings_group` requires to work, you have to use `with_conf_models` function. This function accepts two arguments:

- **migration** - name of migration in `misago.conf` app containing models definitions current for the time of your data migration.
- **this_migration** - dict with model definitions for this migration.

In addition to this, make sure that your migration `depends_on` attribute defines dependency on migration from `misago.conf` app:

```
class Migration(DataMigration):

    # Migration code...

    models = with_conf_models('0001_initial', {
        # This migration models
    })

    depends_on = (
        ("conf", "0001_initial"),
    )
```

delete_settings_cache

`delete_settings_cache`()

If you have used `migrate_settings_group` function in your migration, make sure to call `delete_settings_cache` at its end to flush settings caches.

Misago Settings Reference

By convention, low level settings are written in UPPER_CASE and high level ones are written in lower_case.

account_activation Preffered way in which new user accounts are activated. Can be either of those:

- **none** - no activation required.
- **user** - new user has to click link in activation e-mail.
- **admin** - board administrator has to activate new accounts manually.
- **block** - turn new registrations off.

allow_custom_avatars Controls if users may set avatars from outside forums.

avatar_upload_limit Max allowed size of uploaded avatars in kilobytes.

default_avatar Default avatar assigned to new accounts. Can be either `initials` for randomly generated pic with initials, `gravatar` or `gallery` which will make Misago pick random avatar from gallery instead.

default_timezone Default timezone used by guests and newly registered users that haven't changed their timezone prefferences.

forum_branding_display Controls branding's visibility in forum navbar.

forum_branding_text Allows you to include text besides brand logo on your forum.

forum_name Forum name, displayed in titles of pages.

forum_index_meta_description Forum index Meta Description used as value meta description attribute on forum index.

forum_index_title Forum index title. Can be empty string if not set, in which case `forum_name` should be used instead.

MISAGO_ACL_EXTENSIONS List of Misago ACL framework extensions.

MISAGO_ADMIN_NAMESPACES Link namespaces that are administrator-only areas that require additional security from Misago. Users will have to re-authenticate themselves to access those namespaces, even if they are already signed in your frontend. In addition they will be requested to reauthenticated if they were inactive in those namespaces for certain time.

Defautly `misago:admin` and `admin` namespaces are specified, putting both Misago and Django default admin interfaces under extended security mechanics.

MISAGO_ADMIN_PATH Path prefix for Misago administration backend. Defautly "admincp", but you may set it to empty string if you with to disable your forum administration backend.

MISAGO_ADMIN_SESSION_EXPIRATION Maximum allowed lenght of inactivity period between two requests to admin namespaces. If its exceeded, user will be asked to sign in again to admin backed before being allowed to continue activities.

MISAGO_ATTACHMENTS_ROOT Path to directory that Misago should use to store post attachments. This directory shouldn't be accessible from outside world.

MISAGO_AVATAR_SERVER_PATH Url path that that all avatar server urls starts with. If you are running Misago subdirectory, make sure to update it (i.e. valid path for "http://somesite.com/forums/" is `/forums/user-avatar`).

MISAGO_AVATAR_STORE Path to directory that Misago should use to store user avatars. This directory shouldn't be accessible from outside world.

MISAGO_AVATARS_SIZES Misago uses avatar cache that prescales avatars to requested sizes. Enter here sizes to which those should be optimized.

MISAGO_COMPACT_DATE_FORMAT_DAY_MONTH Date format used by Misago `compact_date` filter for dates in this year.

Expects standard Django date format, documented here

MISAGO_COMPACT_DATE_FORMAT_DAY_MONTH_YEAR Date format used by Misago `compact_date` filter for dates in past years.

Expects standard Django date format, documented here

MISAGO_DIALY_POST_LIMIT Dialy limit of posts that may be posted from single account. Fail-safe for situations when forum is flooded by spam bot. Change to 0 to lift this restriction.

MISAGO_DYNAMIC_AVATAR_DRAWER Function used to create unique avatar for this user. Allows for customization of algorithm used to generate those.

MISAGO_HOURLY_POST_LIMIT Hourly limit of posts that may be posted from single account. Fail-safe for situations when forum is flooded by spam bot. Change to 0 to lift this restriction.

MISAGO_LOGIN_API_URL URL to API endpoint used to authenticate sign-in credentials. Musn't contain api prefix or wrapping slashes. Defaults to 'auth/login'.

MISAGO_MAILER_BATCH_SIZE Default maximum size of single mails package that Misago will build before sending mails and creating next package.

MISAGO_MARKUP_EXTENSIONS List of python modules extending Misago markup.

MISAGO_NEW_REGISTRATIONS_VALIDATORS List of functions to be called when somebody attempts to register on forums using registration form.

MISAGO_NOTIFICATIONS_MAX_AGE Max age, in days, of notifications stored in database. Notifications older than this will be delted.

MISAGO_POSTING_MIDDLEWARES List of middleware classes participating in posting process.

MISAGO_POSTS_PER_PAGE Controls number of posts displayed on thread page. Greater numbers can increase number of objects loaded into memory and thus depending on features enabled greatly increase memory usage.

MISAGO_POSTS_TAIL Defines minimal number of posts for thread's last page. If number of posts on last page is smaller or equal to one specified in this setting, last page will be appended to previous page instead.

MISAGO_RANKING_LENGTH Some lists act as rankings, displaying users in order of certain scoring criteria, like number of posts or likes received. This setting controls maximum age in days of items that should count to ranking.

MISAGO_RANKING_SIZE Maximum number of items on ranking page.

MISAGO_READTRACKER_CUTOFF Controls amount of data used by readtracking system. All content older than number of days specified in this setting is considered old and read, even if opposite is true. Active forums can try lowering this value while less active ones may wish to increase it instead.

MISAGO_SENDFILE_HEADER If your server provides proxy for serving files from application, like "X-Sendfile", set its header name in this setting.

Leave this setting empty to use Django fallback.

MISAGO_SENDFILE_LOCATIONS_PATH Some Http servers (like Nginx) allow you to restrict X-Sendfile to certain locations.

Misago supports this feature with this setting, however with limitation to one "root" path. This setting is used for paths defined in ATTACHMENTS_ROOT and AVATAR_CACHE settings.

Rewrite algorithm used by Misago replaces path until last part with value of this setting.

For example, defining `MISAGO_SENDFILE_LOCATIONS_PATH = 'misago_served_internals'` will result in following rewrite:

`/home/mysite/www/attachments/13_05/142123.rar` => `/misago_served_internals/attachments/13_05/`

MISAGO_STOP_FORUM_SPAM_USE This settings allows you to decide wheter of not Stop Forum Spam database should be used to validate IPs and emails during new users registrations.

MISAGO_STOP_FORUM_SPAM_MIN_CONFIDENCE Minimum confidence returned by Stop Forum Spam for Misago to reject new registration and block IP address for 1 day.

MISAGO_THREADS_ON_INDEX Change this setting to `False` to display categories list instead of threads list on board index.

MISAGO_THREADS_PER_PAGE Controls number of threads displayed on page. Greater numbers can increase number of objects loaded into memory and thus depending on features enabled greatly increase memory usage.

MISAGO_THREADS_TAIL Defines minimal number of threads for lists last page. If number of threads on last page is smaller or equal to one specified in this setting, last page will be appended to previous page instead.

MISAGO_THREAD_TYPES List of clasess defining thread types.

MISAGO_USERS_PER_PAGE Controls pagination of users lists.

password_complexity Complexity requirements for new user passwords. It's value is list of strings representing following requirements:

- **case** - mixed case.
- **alphanumerics** - both digits and letters.
- **special** - special characters.

password_length_min Minimal required length of new user passwords.

post_length_max Maximal allowed post content length.

post_length_min Minimal allowed post content length.

signature_length_max Maximal allowed length of users signatures.

subscribe_reply Default value for automatic subscription to replied threads prefference for new user accounts. Its value represents one of those settings:

- **no** - don't watch.
- **watch** - put on watched threads list.
- **watch_email** - put on watched threads list and send e-mail when somebody replies.

subscribe_start Default value for automatic subscription to started threads prefference for new user accounts. Allows for same values as `subscribe_reply`.

thread_title_length_max Maximal allowed thread title length.

thread_title_length_min Minimal allowed thread title length.

username_length_max Maximal allowed username length.

username_length_min Minimal allowed username length.

Django Settings Reference

Django defines plenty of configuration options that control behaviour of different features that Misago relies on.

Those are documented and available in Django documentation: Settings

Shortcut Functions

Just like Django, Misago defines shortcuts module that reduce some procedures to single functions.

This module lives in `misago.views.shortcuts` and in addition to Misago-native shortcut functions, it imports whole of `django.shortcuts`, so you don't have to import it separately in your views.

paginate

paginate(*object_list*, *page*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)

This function is a factory that validates received data and returns Django's Page object. In adition it also translates `EmptyPage` errors into `Http404` errors and validates if first page number was explictly defined in url parameter or not.

`paginate` function has certain requirements on handling views that use it. Firstly, views with pagination should have two links instead of one:

```
# inside blog.urls.py
urlpatterns += patterns('blog.views',
    url(r'^/$', 'index', name='index'),
    url(r'^/(?P<page>[1-9][0-9]*)/$', 'index', name='index'),
)

# inside blog.views.py
def index(request, page=None):
    # your view that calls paginate()
```

Warning: Giving `page` argument default value of 1 will make `paginate` function assume that first page was reached via link with explicit first page number and cause redirect loop.
Error handler expects link parameter that contains current page number to be named "page". Otherwise it will fail to create new link and raise `KeyError`.

validate_slug

validate_slug(*model*, *slug*)

This function compares model instance's "slug" attribute against user-friendly slug that was passed as link parameter. If model's slug attribute is different this function, `misago.views.OutdatedSlug` is raised. This exception is then captured by Misago's exception handler which makes Misago return permanent (http 301) redirect to client with valid link.

Example of view that first fetches object form database and then makes sure user or spider that reaches page has been let known of up-to-date link:

```
from misago.views.shortcuts import validate_slug, get_object_or_404
from myapp.models import Cake

def cake_fans(request, cake_id, cake_slug):
    # first get cake model from DB
    cake = get_object_or_404(Cake, pk=cake_id)
    # issue redirect if cake slug is invalid
    validate_slug(cake, cacke_slug)
```

Note: You may have noticed that there's no exception handling for either Http404 exception raised by `get_object_or_404`, nor `OutdatedSlug` exception raised by `validate_slug`. This is by design. Both exceptions are handled by Misago for you so you don't have to spend time writing exception handling boiler plate on every view that fetches objects from database and validates their links.

Naturally if you need to, you can still handle them yourself.

Note: Your links should use "slug" parameters only when they are supporting GET requests. For same reason you should call `validate_slug` only when request method is GET or HEAD.

Template Tags Reference

Misago defines plenty of custom tags and filters for use by template authors.

misago_avatars

`avatar` filter Accepts user model instance or integer representing user's PK, returns link to avatar server that can be used as `src` attribute value for `img`.

Takes one optional argument, image size.

`blankavatar` tag Returns link to avatar blank avatar that can be used as `src` attribute value for `img`. Should be used when no user pk is avaialable to render avatar, eg. displaying items belonging to deleted users.

Takes one optional argument, image size.

misago_batch

There are situations when you want to slice list of items in template into sublists, e.g. when displaying grid of items in HTML it makes more sense to split iteration into two steps: iteration over rows and items in each row.

`misago_batch` provides two simple and lazy filters that enable you to do this:

`batch` filter Takes one argument, integer individual batch length, then turns big list into list of lists.

`batchnonefilled` filter Works same as `batch` filter, but with one difference:

If last batch length is shorter than requested, it fills it with `None` to make it requested length.

misago_capture

capture as tag Captures part of template to variable that may then be displayed many more times.

There is also trimmed flavour `capture trimmed as` that trims captured template part before assinging it to variable.

misago_dates

compact_date filter Filter that formats date according to format defines in `MISAGO_COMPACT_DATE_FORMAT_DAY_MONTH` setting if date is in current year, or `MISAGO_COMPACT_DATE_FORMAT_DAY_MONTH_YEAR` if not. Defaults to "7 may" for same year dates and "may '13" for past years dates.

misago_editor

editor_body tag Renders Misago Markup editor body in template. Requires one argument: variable containing editor instance.

editor_js tag Renders Misago Markup editor's javascript in template. Requires one argument: variable containing editor instance.

misago_forms

form_row tag Takes form field as its first argument and renders field complete with label, help and errors. Accept two extra arguments: label class and field class, allowing you to control size of horizontal forms:

```
{% load misago_forms %}

{% form_row form.somefield %}
{% form_row form.otherfield 'col-md-3' 'col-md-9' %}
```

form_input tag Takes form field as its only argument and renders it's input.

misago_json

as_json filter Turns value into json string.

misago_shorthands

iftrue filter Shorthand for simple if clauses: `{{ "fade in"|iftrue:thread.is_closed }}` will render `"fade in"` in template if `thread.is_closed` is true.

iffalse filter Opposite filter for `iftrue`.

misago_pagination

Shortcut for rendering paginators using template. Accepts following arguments:

- **page** - paginator's page object
- **template** - template to use to render paginator
- **link_name** - link name to use for pages

Also accepts kwargs which will be passed to template context as they were given to tag.

Paginator template gets `paginator`, `page` and `link_name` values to use in rendering.

Thread Store

Thread store is simple memory-based cache some Misago features use to maintain state for request duration.

Thread store lives in `misago.core.threadstore` and offers subset of standard cache API known from Django

> **Warning:** Never use thread store for messaging between parts of your code. Usage of this feature for this is considered bad practice that leads to code with flow that is hard to understand.

get

get(*key*, *default=None*)

Get value for key from thread store or default value if key is undefined:

```
>>> from misago.core import threadstore
>>> threadstore.get('peach')
None

>>> threadstore.get('peach', 'no peach!')
'no peach!'
```

get() never raises an exception for non-existant value which is why you should avoid storing "None" values and use custom default values to spot non-existant keys.

set

set(*key*, *value*)

Set value for a key on thread store. This value will then be stored until you overwrite it with new value, thread is killed, `misago.core.middleware.ThreadStoreMiddleware` process_response method is called, or you explictly call clear() funciton, clearing thread store.

clear

clear()

Delete all values from thread store. This function is automatically called by ThreadStoreMiddleware to make sure contents of thread store won't have effect on next request.

Thread Types

All user-published content in Misago follows basic hierarchy: Forums have threads which have (optionally) polls and posts that have attachments. In addition forums and threads have labels.

Whenever user creates new discussion on forums, sends private message to other user, or reports offensive post, mechanics behind UI are largery the same and result in thread of certain kind being posted in destined section of site.

This system was designed to be extensible to enable developers using Misago as foundation for their community sites to add new content types such as blogs, articles or galleries.

Writing custom thread type

Thread type is basically UI for users to interact with and Python code implementing features behind it, plus helper object for enabling your models to point to custom views.

Thread type is decided by value of *special_role* attribute of forum model instance that content (thread, post, attachment, etc. ect.) belongs to. Using this value model is able to call *misago.threads.threadtypes.get(thread_type)* in order to obtain its helper object.

Helper classes

Paths to helper classess definitions are specified in *MISAGO_THREAD_TYPES* settings. Each helper class is expected to define *type_name* attribute corresponding to forum its forum *special_role* attribute.

Note: If *special_role* is not defined, Misago falls back to *role* attribute.

Once helper class is defined, it's available as "thread_type" attribute on forum, thread, post, event, poll and attachment models.

Depending on features used by thread type, its helper is expected to define different of the following methods:

get_forum_name
`get_forum_name`(*forum*)
Used to obtain forum name. Useful when thread type uses single forum with predefined name. Should return string.

get_forum_absolute_url
`get_forum_absolute_url`(*forum*)
Used to obtain forum absolute url.

get_new_thread_url
`get_new_thread_url`(*forum*)
Used to obtain "post new thread" url for forum.

get_reply_url
`get_reply_url`(*thread*)
Used to obtain "post reply" url for thread.

get_edit_post_url
get_edit_post_url(*post*)
Used to obtain edit url for post.

get_thread_absolute_url
get_thread_absolute_url(*thread*)
Used to obtain thread absolute url.

get_thread_post_url
get_thread_post_url(*thread*, *post_id*, *page*)
Used by "go to post" views to build links pointing user to posts. Should return URL to specified thread page with fragment containing specified post.

get_thread_last_reply_url
get_thread_last_reply_url(*thread*)
Should return url to view redirecting to last post in thread.

get_thread_new_reply_url
get_thread_new_reply_url(*thread*)
Should return url to view redirecting to first unread post in thread.

get_thread_moderated_url
get_thread_moderated_url(*thread*)
Should return url to view returning list of posts in thread that are pending moderator review.

get_thread_reported_url
get_thread_reported_url(*thread*)
Should return url to view returning list of reported posts in thread.

get_post_absolute_url
get_post_absolute_url(*post*)
Used to obtain post absolute url.

get_post_approve_url
get_post_approve_url(*post*)
Used to obtain url that moderator should follow to approve post.

get_post_unhide_url
get_post_unhide_url(*post*)
Used to obtain url that will make hidden post visible.

get_post_hide_url
get_post_hide_url(*post*)
Used to obtain url that will make visible post hidden.

get_post_delete_url
get_post_delete_url(*post*)
Used to obtain url that will delete post.

get_post_report_url
get_post_report_url(*post*)
Used to obtain url for reporting post.

get_event_edit_url
get_event_edit_url(*event*)
Used to obtain url that will handle API calls for hiding/unhiding and deleting thread events.

Validators

Misago apps implement plenty of validators, some of which are considered public API. Those validators are per convention contained within `validators` module of their respective apps.

misago.core.validators

validate_sluggable `misago.core.validators.validate_sluggable`

Callable class that validates if string can be converted to non-empty slug thats no longer than 255 characters.

To you use it, first instantiate it. If you want to define custom error messages, you can pass them using `error_short` and `error_long` arguments on initializer. After that you can simply call the class like other validator functions to see if it raises `ValidationError`:

```
from misago.core.validators import validate_sluggable
validator = validate_sluggable()
validator(some_value)
```

misago.users.validators

validate_email `misago.users.validators.validate_email`

Function that takes email address and runs content, availability and ban check validation in this order via calling dedicated validators.

validate_email_banned `misago.users.validators.validate_email_banned()`

Function that accepts email string as its only argument and raises Validation error if it's banned.

validate_email_content `misago.users.validators.validate_email_content`

Callable instance of `django.core.validators.EmailValidator` that checks if email address has valid structure and contents.

validate_password `misago.users.validators.validate_password()`

Function that takes plaintext password and runs length and complexity validation in this order via calling dedicated validators.

validate_password_complexity `misago.users.validators.validate_password_complexity()`

Validates password complexity against tests specified in `password_complexity` setting.

validate_password_length `misago.users.validators.validate_password_length()`

Validates password length and raises ValidationError if specified plaintext password is shorter than `password_length_min`.

validate_username `misago.users.validators.validate_username()`

Function that takes username and runs content, length, availability and ban check validation in this order via calling dedicated validators.

validate_username_available `misago.users.validators.validate_username_available()`

Function that accepts username string as its only argument and raises ValidationError if it's already taken.

validate_username_banned `misago.users.validators.validate_username_banned()`

Function that accepts username string as its only argument and raises Validation error if it's banned.

validate_username_content `misago.users.validators.validate_username_content()`

Function that accepts username string as its only argument and raises Validation error if username contains disallowed characters (eg. those that are not matched by `[0-9a-z]+` regex).

validate_username_length `misago.users.validators.validate_username_length()`

Function that accepts username string as its only argument and raises Validation error if it's shorter than `username_length_min` setting or longer than `username_length_max` setting.

Validating Registrations

Misago implements simple framework for extending process of new user registration with additional checks.

When user submits registration form with valid data, this data is then passed to functions defined in `MISAGO_NEW_REGISTRATIONS_VALIDATORS` setting.

Each function is called with following arguments:

- `ip`: IP address using form.
- `username`: username for which new account will be created.
- `email`: e-mail address for which new account will be created.

If function decides to interrup registration process and thus stop user from registering account, it can raise `django.core.exceptions.PermissionDenied` exception, which will result in user receiving "403 Permission Denied" response from site, as well as having his IP address automatically banned for one day.

If none of defined tests raised `PermissionDenied`, user account will be registered normally.

Errors Boilerplate

Modern forum software is busy place where access content is decided by many factors. This means that your users may frequently be trying to follow links that has been outdated, deleted or simply restricted and each of those scenarios must be handled by your views.

While Django provides plenty of approaches to handling those situations, those can hardly be named fit for internet forum usecases. For example, you may want to communicate to your user a reason why he is not able to reply in selected thread at the moment. This brings need for custom error handling in your code.

And what to do if user reached page using outdated link? You will have to compare link to model's slug field and return 301 redirect to valid address on every view that has friendly link.

To solve this problem you would have to write custom error views and handlers that then you would have to add in every view that needs it. Depending on number of views you are writing, number of lines would quickly add up becoming annoying boilerplate.

Misago views too have to solve this problem and this reason is why error handling boilerplate is part of framework.

Views Exceptions

While Misago raises plenty of exceptions, only four are allowed to leave views. Two of those are django's `Http404` and `PermissionDenied` exceptions. Misago defines its own two exceptions that act as "messages" for it's own error handler that link user followed to reach view is not up-to-date and could use 301 redirect to make sure bookmarks and crawlers get current link.

Note: You should never raise those exceptions yourself. If you want to redirect user to certain page, return proper redirect response instead.

Banned `misago.core.exceptions.Banned`

Raising this exception with `Ban` or `BanCache` instance as its only argument will cause Misago to display "You are banned" error page to the user.

ExplicitFirstPage `misago.core.exceptions.ExplicitFirstPage`

This exception is raised by `misago.core.shortcuts.paginate()` helper function that creates pagination for given data, page number and configuration. If first page is explicit ("user-blog/1/") instead implicit ("user-blog/"), this exception is raised for error handler to return redirect to link with implicit first page.

> **Warning:** This is reason why Misago views pass this function `None` as page number when no page was passed through link.

OutdatedSlug `misago.core.exceptions.OutdatedSlug`

This exception is raised by `misago.core.shortcuts.validate_slug()` helper function that compares link's "slug" part against one from database. If check fails OutdatedSlug exception is raised with parameter name and valid slug as message that Misago's exception handler then uses to construct redirection response to valid link.

Exception Handler

`misago.core.exceptionhandler`

Exception handler is lightweight system that pairs exceptions with special "handler" functions that turn those exceptions into valid HTTP responses that are then served back to client.

This system has been designed exlusively for handling exceptions listed in this document and is was not intended to be universal and extensible solution. If you need special handling for your own exception, depending on how wide is its usage, consider writing custom exception handler decorator or middleware for it.

P

R

S

U

V

W