gs.group.member.canpost Documentation

Release 3.1.3

GroupServer.org

December 05, 2015

Contents

CHAPTER 1

Determining if a group member can post

Author Michael JasonSmith

Contact Michael JasonSmith <mpj17@onlinegroups.net>

Date 2015-12-02

Organization GroupServer.org

Contents:

1.1 Posting rules

In this section I present the *Rule abstract base-class*, and how the *Can Post adaptor* is used to collect the rules for each group. I then provide an example of *chaining rules*.

1.1.1 Rule abstract base-class

A rule is an adaptor. It takes a user [1] and a group [2]. The `BaseRule` abstract base-class provides most of what is required to create a rule.

class **`BaseRule`**(*userInfo*, *group*)

Parameters

- **`userInfo`** (*IGSUserInfo*) – The user that is being tested by the rule.
- **`group`** – The group the user is being tested for.

`__doc__`
(Abstract property: sub-classes must provide a weight.) The documentation on the rule, which is shown on the page `rules.html` in each group.

`weight`
(Abstract property: sub-classes must provide a weight.) An integer that has two related functions. First, it is used as the sort-key to determine the order that the rules are checked (see *can post adaptor* below). Second, the `statusNum` is set to this value to uniquely identifies the rule. (No two rules should have the same weight as this can lead to ambiguity.)

[1] The user is almost always a `Products.CustomUserFolder.interfaces.IGSUserInfo` instance.
[2] The group will be a group-folder that has been marked with an interface that is *generally* specific to the type of group.

check()
(Abstract method: sub-classes must supply *check()*.) Perform the actual check to see if a user can post to a group. Based on the result it sets the values in the dictionary `self.s`:

checked: `True` after the check, `False` initially.

canPost: `True` if the user can post (see *canPost*).

status: A string that summarises the status (see *status*).

statusNum: A number representing the status. Normally this is set to *weight* (see *statusNum*).

canPost
(Read only.) A Boolean value that is `True` if this rule thinks that the user can post the the group.

status
(Read only.) A Unicode value that summaries why the user should be prevented from posting to the group.

statusNum
(Read only.) An integer that is one of three values:

- `-1` if it is unknown whether the user can post to the group (`canPost` will be `False` in this case),
- `0` if the user can post to the group (`canPost` is `True`), and
- Set to the `weight` value if the user cannot post to the group (`canPost` is `False`).

Example

Most rules only provide a doc-string, the *BaseRule.weight* attribute, and *BaseRule.check()* method. For example, the `BlockedFromPosting` rule checks to see if the identifier of the user is in the `blocked_members` property of the mailing list. It then sets the `canPost`, `status` and `statusNum` values of the `self.s` dictionary accordingly. Finally, it sets `self.s['checked']` to `True` to prevent the system from performing the check more than once.

```
class BlockedFromPosting(BaseRule):
  '''A person will be prevented from posting if he or she is
  explicitly blocked by an administrator of the group.'''
  weight = 10

  def check(self):
      if not self.s['checked']:
          ml = self.mailingList
          blockedMemberIds = ml.getProperty('blocked_members', [])
          if (self.userInfo.id in blockedMemberIds):
              self.s['canPost'] = False
              self.s['status'] = 'blocked from posting'
              self.s['statusNum'] = self.weight
          else:
              self.s['canPost'] = True
              self.s['status'] = 'not blocked from posting'
              self.s['statusNum'] = 0
          self.s['checked'] = True
```

The **ZCML** sets up each rule as an adaptor [3]. It adapts a `userInfo` and the *specific* group type and provides an `IGSCanPostRule`. The adaptor must be a **named adaptor**, as multiple rules are used for each group. The names are also shown on the `rules.html` page in each group.

[3] It easier to use ZCML to set up the adaptor for each rule because rules can be mixed and matched by different group-types. By using ZCML the mixing-and-matching can be done with very little Python code.

```
<adapter
  name="Blocked from Posting"
  for="Products.CustomUserFolder.interfaces.IGSUserInfo
       gs.group.base.interfaces.IGSGroupMarker"
  provides=".interfaces.IGSCanPostRule"
  factory=".rules.BlockedFromPosting" />
```

1.1.2 Can post adaptor

The `CanPost` adaptor looks very very very much like the adaptor for the *rule abstract base-class*. However, rather than providing a single rule it *aggregates* all the rules for a group, giving the final answer as to weather the user can post. It provides the answer using the same three properties as the rules: *`CanPost.canPost`*, *`CanPost.status`* and *`CanPost.statusNum`*.

class CanPost (*userInfo*, *group*)

Parameters

- **userInfo** (*IGSUserInfo*) – The user that is being tested.
- **group** – The group the user is being tested for.

canPost
`True` if the user can post to the group.

status
A description of the reason the user cannot post, for the most important reason (the rule with lowest weight; see *`BaseRule.weight`*). Undefined if *`canPost`* is `True`.

statusNum
A numeric description of the reason the user cannot post, for the most important reason (the rule with lowest weight; see *`BaseRule.weight`*). Undefined if *`canPost`* is `True`.

Only one `CanPost` adaptor is needed for *all* group-types. That is because the it implements the **strategy** pattern to determine the applicable rules.

1.1.3 Chaining Rules

The core GroupServer group types use the following inheritance hierarchy for their interfaces:

```
gs.group.base.interfaces.IGSGroupMarker

   |         |
   |       gs.group.type.discussion.interfaces.IGSDiscussionGroup
   |
   |         |
   |       gs.group.type.announcement.interfaces.IGSAnnouncementGroup
   |
  gs.group.type.support.interfaces.IGSSupportGroup
```

This product (`gs.group.member.canpost`) provides one rule for the `IGSGroupMarker` — which prevents people who have been explicitly blocked from posting (see the *example* above). All other group types inherit this rule because their marker-interfaces inherit from the `IGSGroupMarker`.

The discussion group (`IGSDiscussionGroup`) provides the most rules: six in all. All these rules are inherited by the announcement group because its marker-interface (`IGSAnnouncementGroup`) inherits from the discussion group. The announcement group also provides its own rule, to ensure that only posting members can post.

The support group (`IGSSupportGroup`) provides no extra rules, so it just has the rule that is provided by this package for all the `IGSGroupMarker` groups.

1.2 Viewlets

Each rule will need a viewlet that provides feedback about why a person cannot post. The code for each viewlet is relatively simple:

- Each viewlet inherits from `RuleViewlet`, and
- The `weight` for each viewlet is taken from the weight for the respective rule.

For example, the viewlet for the *Blocked from posting* rule is as follows:

```
class BlockedRuleViewlet(RuleViewlet):
    weight = BlockedFromPosting.weight
```

The viewlets appear in two places. First, they are shown at the bottom of the Topic page if the person viewing the page cannot post. Second, they are shown in the Notifications.

1.2.1 Abstract blase-class

The rule-viewlets typically inherit from the `RuleViewlet` abstract base-class.

RuleViewlet(group):

> **param group** The group that the viewlet is for
>
> `RuleViewlet` is an *abstract base-class* for viewlets that display rules.

weight
: The *weight* (sort-order) for the viewlet. Sub-classes must implement this property.

show
: `True` if the viewlet should be shown. Read only.

canPost
: The *CanPost* instance for the current user and the group. Read only.

userInfo
: The user that the rule is for. Read only.

1.3 Notifications

There are two notifications: the *cannot post* notification is sent to people with a profile who cannot post, while *unknown email address* is sent when the email address is not recognised.

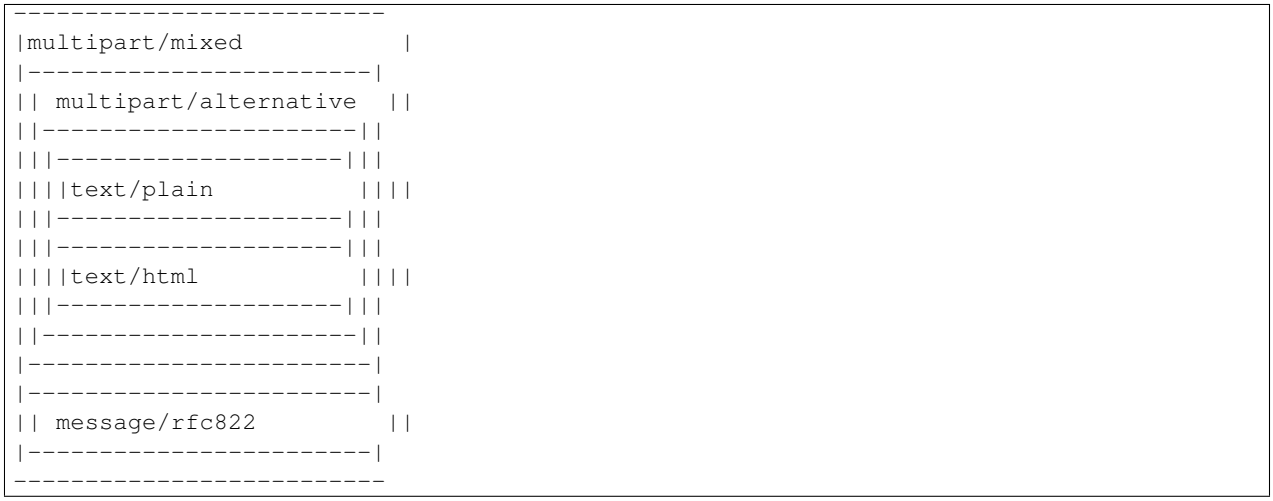
1.3.1 Cannot post

The Cannot Post notification is sent out to people who post to the group, but the rules block the post. The notification contains the viewlets [4]. As such care should be taken to ensure that each viewlet makes sense outside the context of the group, and all links in each viewlet are **absolute** links that include the site name.

[4] The Cannot Post notification contains each viewlet in two forms: the normal HTML version, and a plain-text version, which the notification generates from the HTML.

The Cannot Post notification can be previewed by viewing the pages `cannot-post.html` and `cannot-post.txt` within each group.

The notification email is sent using a variant of the class `gs.profile.notify.sender.MessageSender`. The main difference is the notification is constructed differently, so it can include the original email message that was blocked. The notification email is made up of five parts:



- The text of the Cannot Post notification is contained within two components:
  - `text/plain` contains the `cannot-post.txt` message, and
  - `text/html` components contains the `cannot-post.html`.
- The two text block are wrapped in a `multipart/alternative` block.
- The message that could not be posted is placed in a `message/rfc822` block at the end of the email.
- Finally, everything is wrapped in a `multipart/mixed` block, which carries the subject line, addresses, and the rest of the headers.

1.3.2 Unknown Email Address

The unknown email address notification can be thought of as a highly specialised form of Cannot Post. It is sent when the mailing list (`Products.XWFMailingListManager.XWFMailingList`) fails to recognise the email address of the sender of a message.

The notification is constructed the same way as the *cannot post* notification, with the same five parts. The text encourages the recipient to add the email address to his or her profile: we speculate that existing members posting from an unknown email address is the most common reason for receiving the notification. The rest of the message is similar to the "Not a Member" message that is sent by the standard Cannot Post notification. The text can be previewed by looking at the `unknown-email.html` and `unknown-email.txt` within each group.

The unknown-email notifier (`unknownemail.Notifier` within this egg) avoids all use of the `gs.profile.notify` system — because there is not profile to sent the notification to! To send the notification the code assembles the email message, and sends the post using `gs.email.send_email`.

TODO

The unknown email address notification should *probably* appear in the code that handles the mailing list. However, that product [5] is due for a **huge** refactor, so the unknown email address notification was placed here for safe-keeping. In the future this notification should be moved closer to the mailing list.

1.4 Changelog

1.4.1 3.1.3 (2015-12-02)

- Using *Hello* rather than *Dear* in the opening salutation of the *Cannot post* message
- Switching from using `functools.reduce` to `all` to determine if someone can post
- Adding unit tests for the `CanPost` class, and the `BlockedFromPosting` rule
- Moving the documentation to Sphinx

1.4.2 3.1.2 (2015-05-26)

- Pointing the *Request* link at the group page

1.4.3 3.1.1 (2015-02-11)

- Handle poorly formatted `Subject` lines better
- Added some basic unit tests

1.4.4 3.1.0 (2015-01-14)

- Following the `convert_to_txt` function to `gs.group.list.base`.
- Naming the ReStructuredText files as such
- Moving the repository to GitHub

1.4.5 3.0.2 (2014-03-31)

- Adding logging to the *Unknown email address* notification

1.4.6 3.0.1 (2014-02-26)

- Using the `gs.content.email.base.TextMixin` class
- Adding `gs.core` to the product dependencies

[5] See <https://github.com/groupserver/Products.XWFMailingListManager>

1.4.7 3.0.0 (2013-10-09)

- Switching the notifications to use the standard code provided by `gs.content.email.base`
- Switching to absolute-imports
- Further PEP-8 compliance cleanups

1.4.8 2.4.0 (2013-05-28)

- Removing the jQuery UI code from the `canpost` info

1.4.9 2.3.1 (2013-01-22)

- Code cleanup, thanks to Ninja IDE

1.4.10 2.3.0 (2012-08-06)

- Add a delivery-address check

1.4.11 2.2.0 (2012-07-18)

- Use the new `gs.email` product to send the notifications

1.4.12 2.1.0 (2012-06-22)

- Updating SQLAlchemy

1.4.13 2.0.0 (2012-03-28)

- New *Cannot post* notification and a *Unknown email address* notification that has both HTML and plain-text components
- Allow the *Cannot post* viewlet manager to work from anywhere
-

1.4.14 1.0.1 (2011-11-22)

- Updated documentation

1.4.15 1.0.0 (2011-17-11)

- Initial release, inspired (loosely) on code provided by `Products.XWFMailingListManager`.

This is the core code for determining if a group member can post. The mailing list code, the *Topic* page and *Start a topic* page rely on this code for determining if a member can post.

In this document I present how the rules for posting are created for each different type of group. I then discuss the viewlets and the notifications that are sent to those that cannot post.

1.5 Indices and tables

- genindex
- modindex
- search

1.6 Resources

- Code repository: https://github.com/groupserver/gs.group.member.canpost
- Questions and comments to http://groupserver.org/groups/development
- Report bugs at https://redmine.iopen.net/projects/groupserver

Symbols

B

C

S

U

W