plone.api Documentation

Release 1.1.0-rc.1

July 07, 2015

Contents

W ARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

CHAPTER 1

A Plone API

1.1 plone.api

- Documentation @ api.plone.org
- Source code @ GitHub
- Issues @ GitHub
- Continuous Integration @ Travis CI
- Code Coverage @ Coveralls.io

Overview

The `plone.api` is an elegant and simple API, built for humans wishing to develop with Plone.
It comes with *cookbook*-like documentation and step-by-step instructions for doing common development tasks in Plone. Recipes try to assume the user does not have extensive knowledge about Plone internals.

The intention of this package is to provide clear API methods for Plone functionality which may be confusing or difficult to access. As the underlying code improves some API methods may be deprecated and the documentation here will be updated to show how to use the improved code (even if it means not using `plone.api`)

Some parts of the documentation do not use *plone.api* methods directly, but simply provide guidance on achieving a task using Plone's internal API. For example, using the portal catalog (see 'Find content objects').

The intention is to cover 20% of the tasks any Plone developer does 80% of the time. By keeping everything in one place, the API stays introspectable and discoverable, important aspects of being Pythonic.

Note: This package is stable and used in production, but from time to time changes will be made to the API. Additional api methods may be introduced in minor versions (1.1 -> 1.2). Backward-incompatible changes to the API will be restricted to major versions (1.x -> 2.x).

1.2 Narrative documentation

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.1 About

Inspiration

We want *plone.api* to be developed with PEP 20 idioms in mind, in particular:

> Explicit is better than implicit.
> Readability counts.
> There should be one– and preferably only one –obvious way to do it.
> Now is better than never.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.

All contributions to *plone.api* should keep these rules in mind.

Two libraries are especially inspiring:

SQLAlchemy Arguably, the reason for SQLAlchemy's success in the developer community lies as much in its feature set as in the fact that its API is very well designed, is consistent, explicit, and easy to learn.

Requests If you look at the documentation for this library, or make a comparison between the urllib2 way and the requests way, you cannot but see a parallel between the way we *have been* and the way we *should be* writing code for Plone. At the least, we should have the option to write such clean code.

The API provides grouped functional access to otherwise distributed logic in Plone. This distribution is a result of two historical factors: re-use of CMF- and Zope-methods and reasonable but hard to remember splits like *acl_users* and *portal_memberdata*. Methods defined in *plone.api* implement best-practice access to the original distributed APIs. These methods also provide clear documentation of how best to access Plone APIs directly.

Note: If you doubt those last sentences: We had five different ways to get the portal root with different edge-cases. We had three different ways to move an object. With this in mind, it's obvious that even the most simple tasks can't be documented in Plone in a sane way.

We do not intend to cover all possible use-cases, only the most common. We will cover the 20% of possible tasks on which we spend 80% of our time. If you need to do something that *plone.api* does not support, use the underlying APIs directly. We try to document sensible use cases even when we don't provide APIs for them, though.

Design decisions

Import and usage style

API methods are grouped according to what they affect. For example: *Portal*, *Content*, *Users*, *Environment* and *Groups*. In general, importing and using an API looks something like this:

```
from plone import api

portal = api.portal.get()
catalog = api.portal.get_tool(name="portal_catalog")
user = api.user.create(email='alice@plone.org')
```

Always import the top-level package (`from plone import api`) and then use the group namespace to access the method you want (`portal = api.portal.get()`).

All example code should adhere to this style, to encourage one and only one preferred way of consuming API methods.

Prefer keyword arguments

We prefer using keyword arguments to positional arguments. Example code in *plone.api* will use this style, and we recommend users to follow this convention. For the curious, here are the reasons:

1. There will never be a doubt when writing a method on whether an argument should be positional or not. Decision already made.
2. There will never be a doubt when using the API on which argument comes first, or which ones are named/positional. All arguments are named.
3. When using positional arguments, the method signature is dictated by the underlying implementation (think required vs. optional arguments). Named arguments are always optional in Python. Using keywords allows implementation details to change while the signature is preserved. In other words, the underlying API code can change substantially but code using it will remain valid.
4. The arguments can all be passed as a dictionary.

```
# GOOD
from plone import api
alice = api.user.get(username='alice@plone.org')

# BAD
from plone.api import user
alice = user.get('alice@plone.org')
```

FAQ

Why aren’t we using wrappers?

We could wrap an object (like a user) with an API to make it more usable right now. That would be an alternative to the convenience methods.

Unfortunately a wrapper is not the same as the object it wraps, and answering the inevitable questions about this difference would be confusing. Moreover, functionality provided by `zope.interface` such as annotations would need to be proxied. This would be extremely difficult, if not impossible.

It is also important that developers be able to ensure that their tests continue to work even if wrappers were to be deprecated. Consider the failure lurking behind test code such as this:

```
if users['bob'].__class__.__name__ == 'WrappedMemberDataObject':
    # do something
```

Why `delete` instead of `remove`?

- The underlying code uses methods that are named more similarly to *delete* rather than to *remove*.
- The `CRUD` verb is *delete*, not *remove*.

GitHub-only

WARNING: If you are reading this on GitHub, DON’T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.2 Portal

Get portal object

Getting the Plone portal object is easy with `api.portal.get()`.

```
from plone import api
portal = api.portal.get()
```

Get navigation root

In multi-lingual Plone installations you probably want to get the language-specific navigation root object, not the top portal object. You do this with `api.portal.get_navigation_root()`.

Assuming there is a document `english_page` in a folder `en`, which is the navigation root:

```
from plone import api
nav_root = api.portal.get_navigation_root(english_page)
```

returns the folder `en`. If the folder `en` is not a navigation root it would return the portal.

Get portal url

Since we now have the portal object, it's easy to get the portal url.

```
from plone import api
url = api.portal.get().absolute_url()
```

Get tool

To get a portal tool in a simple way, just use `api.portal.get_tool()` and pass in the name of the tool you need.

```
from plone import api
catalog = api.portal.get_tool(name='portal_catalog')
```

Get localized time

To display the date/time in a user-friendly way, localized to the user's prefered language, use `api.portal.get_localized_time()`.

```
from plone import api
from DateTime import DateTime
today = DateTime()
localized = api.portal.get_localized_time(datetime=today)
```

Send E-Mail

To send an e-mail use `api.portal.send_email()`:

```
from plone import api
api.portal.send_email(
    recipient="bob@plone.org",
    sender="noreply@plone.org",
    subject="Trappist",
    body="One for you Bob!",
)
```

Show notification message

With `api.portal.show_message()` you can show a notification message to the user.

```
from plone import api
api.portal.show_message(message='Blueberries!', request=request)
```

Get plone.app.registry record

Plone comes with a package `plone.app.registry` that provides a common way to store various configuration and settings. `api.portal.get_registry_record()` provides an easy way to access these.

```
from plone import api
api.portal.get_registry_record('my.package.someoption')
```

Set plone.app.registry record

Plone comes with a package `plone.app.registry` that provides a common way to store various configuration and settings. `api.portal.set_registry_record()` provides an easy way to change these.

```
from plone import api
api.portal.set_registry_record('my.package.someoption', False)
```

Further reading

For more information on possible flags and usage options please see the full *plone.api.portal* specification.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.3 Content

Create content

To add an object, you must first have a container in which to put it. Get the portal object, it will serve nicely:

```
from plone import api
portal = api.portal.get()
```

Create your new content item using the `api.content.create()` method. The type argument will decide which content type will be created. Both Dexterity and Archetypes content types are supported.

```
from plone import api
obj = api.content.create(
    type='Document',
    title='My Content',
    container=portal)
```

The `id` of the new object is automatically and safely generated from its `title`.

```
assert obj.id == 'my-content'
```

Get content object

There are several approaches to getting your content object. Consider the following portal structure:

```
plone (portal root)
|-- blog
|-- about
|    |-- team
|    `-- contact
`-- events
     |-- training
     |-- conference
     `-- sprint
```

The following operations will get objects from the stucture above, including using `api.content.get()`.

```
# let's first get the portal object
from plone import api
portal = api.portal.get()
assert portal.id == 'plone'

# content can be accessed directly with dict-like access
blog = portal['blog']

# another way is to use ``get()`` method and pass it a path
about = api.content.get(path='/about')

# more examples
conference = portal['events']['conference']
sprint = api.content.get(path='/events/sprint')

# moreover, you can access content by its UID
uid = about['team'].UID()
team = api.content.get(UID=uid)
```

Find content objects

You can use the *catalog* to search for content. Here is a simple example:

```
from plone import api
catalog = api.portal.get_tool(name='portal_catalog')
documents = catalog(portal_type='Document')
```

More information about how to use the catalog may be found in the Collective Developer Documentation. Note that the catalog returns *brains* (metadata stored in indexes) and not objects. However, calling `getObject()` on brains does in fact give you the object.

```
document_brain = documents[0]
document_obj = document_brain.getObject()
assert document_obj.__class__.__name__ == 'ATDocument'
```

Get content object UUID

A Universally Unique IDentifier (UUID) is a unique, non-human-readable identifier for a content object which stays on the object even if the object is moved.

Plone uses UUIDs for storing references between content and for linking by UIDs, enabling persistent links.

To get the UUID of any content object use `api.content.get_uuid()`. The following code gets the UUID of the `contact` document.

```
from plone import api
portal = api.portal.get()
contact = portal['about']['contact']

uuid = api.content.get_uuid(obj=contact)
```

Move content

To move content around the portal structure defined above use the `api.content.move()` method. The code below moves the `contact` item (with all it contains) out of the folder `about` and into the Plone portal root.

```
from plone import api
portal = api.portal.get()
contact = portal['about']['contact']

api.content.move(source=contact, target=portal)
```

Actually, `move` behaves like a filesystem move. If you pass it an `id` argument the object will have that new ID in it's new home. By default it will retain its original ID.

Rename content

To rename a content object (change its ID), use the `api.content.rename()` method.

```
from plone import api
portal = api.portal.get()
api.content.rename(obj=portal['blog'], new_id='old-blog')
```

Copy content

To copy a content object, use the `api.content.copy()` method.

```
from plone import api
portal = api.portal.get()
training = portal['events']['training']

api.content.copy(source=training, target=portal)
```

Note that the new object will have the same ID as the old object (unless otherwise stated). This is not a problem, since the new object is in a different container.

You can also set `target` to source's container and set `safe_id=True` which will duplicate your content object in the same container and assign it a new, non-conflicting ID.

```
api.content.copy(source=portal['training'], target=portal, safe_id=True)
new_training = portal['copy_of_training']
```

Delete content

To delete a content object, pass the object to the `api.content.delete()` method:

```
from plone import api
portal = api.portal.get()
api.content.delete(obj=portal['copy_of_training'])
```

Content manipulation with the *safe_id* option

When manipulating content with `api.content.create()`, `api.content.move()` or `api.content.copy()` the *safe_id* flag is disabled by default. This means the uniqueness of IDs will be enforced. If another object with the same ID is already present in the target container these API methods will raise an error.

However, if the *safe_id* option is enabled, a non-conflicting id will be generated.

```
api.content.create(container=portal, type='Document', id='document', safe_id=True)
document = portal['document-1']
```

Get workflow state

To find out the current workflow state of your content, use the `api.content.get_state()` method.

```
from plone import api
portal = api.portal.get()
state = api.content.get_state(obj=portal['about'])
```

Transition

To transition your content to a new workflow state, use the `api.content.transition()` method.

```
from plone import api
portal = api.portal.get()
state = api.content.transition(obj=portal['about'], transition='publish')
```

Get view

To get a `BrowserView` for your content, use `api.content.get_view()`.

```
from plone import api
portal = api.portal.get()
view = api.content.get_view(
    name='plone',
    context=portal['about'],
    request=request,
)
```

Further reading

For more information on possible flags and usage options please see the full *plone.api.content* specification.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.4 Users

Create user

To create a new user, use `api.user.create()`. If your portal is configured to use emails as usernames, you just need to pass in the email of the new user.

```
from plone import api
user = api.user.create(email='alice@plone.org')
```

Otherwise, you also need to pass in the username of the new user.

```
user = api.user.create(email='jane@plone.org', username='jane')
```

To set user properties when creating a new user, pass in a properties dict.

```
properties = dict(
    fullname='Bob',
    location='Munich',
)
user = api.user.create(
    username='bob',
    email='bob@plone.org',
    properties=properties,
)
```

Besides user properties you can also specify a password for the new user. Otherwise a random 8-character alphanumeric password will be generated.

```
user = api.user.create(
    username='noob',
    email='noob@plone.org',
    password='secret',
)
```

Get user

You can get a user with `api.user.get()`.

```
from plone import api
user = api.user.get(username='bob')
```

User properties

Users have various properties set on them. This is how you get and set them, using the underlying APIs:

```
from plone import api
user = api.user.get(username='bob')
user.setMemberProperties(mapping={ 'location': 'Neverland', })
location = user.getProperty('location')
```

Get currently logged-in user

Getting the currently logged-in user is easy with `api.user.get_current()`.

```
from plone import api
current = api.user.get_current()
```

Check if current user is anonymous

Sometimes you need to trigger or display some piece of information only for logged-in users. It's easy to use `api.user.is_anonymous()` to do a basic check for it.

```
from plone import api
if not api.user.is_anonymous():
    trigger = False
trigger = True
```

Get all users

Get all users in your portal with `api.user.get_users()`.

```
from plone import api
users = api.user.get_users()
```

Get group's users

If you set the *groupname* parameter, then `api.user.get_users()` will return only users that are members of this group.

```
from plone import api
users = api.user.get_users(groupname='staff')
```

Delete user

To delete a user, use `api.user.delete()` and pass in either the username or the user object you want to delete.

```
from plone import api
api.user.create(username='unwanted', email='unwanted@example.org')
api.user.delete(username='unwanted')
```

```
unwanted = api.user.create(username='unwanted', email='unwanted@example.org')
api.user.delete(user=unwanted)
```

Get user roles

The `api.user.get_roles()` method is used for getting a user's roles. By default it returns site-wide roles.

```
from plone import api
roles = api.user.get_roles(username='jane')
```

If you pass in a content object, it will return local roles of the user in that particular context.

```
from plone import api
portal = api.portal.get()
blog = api.content.create(container=portal, type='Document', id='blog', title='My blog')
roles = api.user.get_roles(username='jane', obj=portal['blog'])
```

Get user permissions

The `api.user.get_permissions()` method is used for getting user's permissions. By default it returns site root permissions.

```
from plone import api
mike = api.user.create(email='mike@plone.org', username='mike')
permissions = api.user.get_permissions(username='mike')
```

If you pass in a content object, it will return local permissions of the user in that particular context.

```
from plone import api
portal = api.portal.get()
folder = api.content.create(container=portal, type='Folder', id='folder_two', title='Folder Two')
permissions = api.user.get_permissions(username='mike', obj=portal['folder_two'])
```

Grant roles to user

The `api.user.grant_roles()` allows us to grant a list of roles to the user.

```
from plone import api
api.user.grant_roles(username='jane',
    roles=['Reviewer', 'SiteAdministrator']
)
```

If you pass a content object or folder, the roles are granted only on that context and not site-wide. But all site-wide roles will also be returned by `api.user.get_roles()` for this user on the given context.

```
from plone import api
folder = api.content.create(container=portal, type='Folder', id='folder_one', title='Folder One')
api.user.grant_roles(username='jane',
    roles=['Editor', 'Contributor'],
    obj=portal['folder_one']
)
```

Revoke roles from user

The `api.user.revoke_roles()` allows us to revoke a list of roles from the user.

```
from plone import api
api.user.revoke_roles(username='jane', roles=['SiteAdministrator'])
```

If you pass a context object the local roles for that context will be removed.

```
from plone import api
folder = api.content.create(
    container=portal,
    type='Folder',
    id='folder_three',
    title='Folder Three'
)
api.user.grant_roles(
    username='jane',
    roles=['Editor', 'Contributor'],
    obj=portal['folder_three'],
)
api.user.revoke_roles(
    username='jane',
    roles=['Editor'],
    obj=portal['folder_three'],
)
```

Further reading

For more information on possible flags and usage options please see the full *plone.api.user* specification.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.5 Groups

Create group

To create a new portal group, use `api.group.create()`.

```
from plone import api
group = api.group.create(groupname='staff')
```

When creating groups `title`, `description`, `roles` and `groups` are optional.

```
from plone import api

group = api.group.create(
    groupname='board_members',
    title='Board members',
    description='Just a description',
    roles=['Readers', ],
    groups=['Site Administrators', ],
)
```

Get group

To get a group by its name, use `api.group.get()`.

```
from plone import api
group = api.group.get(groupname='staff')
```

Editing a group

Groups can be edited by using the `group_tool`. In this example the `title`, `description` and `roles` are updated for the group 'Staff'.

```
from plone import api
group_tool = api.portal.get_tool(name='portal_groups')
group_tool.editGroup(
    'staff',
    roles=['Editor', 'Reader'],
    title='Staff',
    description='Just a description',
)
```

Get all groups

You can also get all groups, by using `api.group.get_groups()`.

```
from plone import api
groups = api.group.get_groups()
```

Get user's groups

The groups returned may be filtered by member. By passing the `username` parameter, `api.group.get_groups()` will return only those groups to which the user belongs.

```
from plone import api
user = api.user.get(username='jane')
groups = api.group.get_groups(username='jane')
```

You may also pass the user directly to `api.group.get_groups()`:

from plone import api user = api.user.get(username='jane') groups = api.group.get_groups(user=user)

Get group members

Use the `api.user.get_users()` method to get all the users that are members of a group.

```
from plone import api
members = api.user.get_users(groupname='staff')
```

Delete group

To delete a group, use `api.group.delete()` and pass in either the groupname or the group object you want to delete.

```
from plone import api
api.group.create(groupname='unwanted')
api.group.delete(groupname='unwanted')
```

```
unwanted = api.group.create(groupname='unwanted')
api.group.delete(group=unwanted)
```

Adding user to group

To add a user to a group, use the `api.group.add_user()` method. This method accepts either the groupname or the group object for the target group and the username or the user object you want to add to the group.

```
from plone import api

api.user.create(email='bob@plone.org', username='bob')
api.group.add_user(groupname='staff', username='bob')
```

Removing user from group

To remove a user from a group, use the `api.group.remove_user()` method. This also accepts either the groupname or the group object for the target group and either the username or the user object you want to remove from the group.

```
from plone import api
api.group.remove_user(groupname='staff', username='bob')
```

Get group roles

To find the roles assigned to a group, use the `api.group.get_roles()` method. By default it returns site-wide roles.

```
from plone import api
roles = api.group.get_roles(groupname='staff')
```

If you pass in a content object, it will return the local roles of the group in that particular context.

```
from plone import api
portal = api.portal.get()
folder = api.content.create(
    container=portal,
    type='Folder',
    id='folder_four',
    title='Folder Four',
)
roles = api.group.get_roles(groupname='staff', obj=portal['folder_four'])
```

Grant roles to group

To grant roles to a group, use the `api.group.grant_roles()` method. By default, roles are granted site-wide.

```
from plone import api
api.group.grant_roles(
    groupname='staff',
    roles=['Reviewer, SiteAdministrator'],
)
```

If you pass in a content object, roles will be assigned in that particular context.

```
from plone import api
portal = api.portal.get()
folder = api.content.create(
    container=portal, type='Folder', id='folder_five', title='Folder Five')
api.group.grant_roles(
    groupname='staff', roles=['Contributor'], obj=portal['folder_five'])
```

Revoke roles from group

To revoke roles already granted to a group, use the `api.group.revoke_roles()` method.

```
from plone import api
api.group.revoke_roles(
    groupname='staff', roles=['Reviewer, SiteAdministrator'])
```

If you pass in a content object, it will revoke roles granted in that particular context.

```
from plone import api
api.group.revoke_roles(
    groupname='staff', roles=['Contributor'], obj=portal['folder_five'])
```

Further reading

For more information on possible flags and complete options please see the full *plone.api.group* specification.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.2.6 Environment

Switch roles inside a block

To temporarily override the list of roles that are available, use `api.env.adopt_roles()`. This is especially useful in unit tests.

```
from plone import api
from AccessControl import Unauthorized

portal = api.portal.get()
with api.env.adopt_roles(['Anonymous']):
    self.assertRaises(
       Unauthorized,
       lambda: portal.restrictedTraverse("manage_propertiesForm")
    )

with api.env.adopt_roles(['Manager', 'Member']):
    portal.restrictedTraverse("manage_propertiesForm")
```

Switch user inside a block

To temporarily override the user which is currently active, use `api.env.adopt_user()`.

```
from plone import api

portal = api.portal.get()

# Create a new user.
api.user.create(
    username="doc_owner",
    roles=('Member', 'Manager',),
    email="new_owner@example.com",
)

# Become that user and create a document.
with api.env.adopt_user(username="doc_owner"):
    api.content.create(
        container=portal,
        type='Document',
        id='new_owned_doc',
    )

self.assertEqual(
    portal.new_owned_doc.getOwner().getId(),
    "doc_owner",
)
```

Debug mode

To know if your zope instance is running in debug mode, use `api.env.debug_mode()`.

```
from plone import api

in_debug_mode = api.env.debug_mode()
if in_debug_mode:
    print 'Zope is in debug mode'
```

Test mode

To know if your plone instance is running in a test runner, use `api.env.test_mode()`.

```
from plone import api

in_test_mode = api.env.test_mode()
if in_test_mode:
    pass  # do something
```

Further reading

For more information on possible flags and usage options please see the full *plone.api.env* specification.

1.3 Complete API and advanced usage

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.1 List of all API methods with descriptions

api.portal

`api.portal.get`
`api.portal.get_navigation_root`
`api.portal.get_tool`
`api.portal.get_localized_time`
`api.portal.send_email`
`api.portal.show_message`
`api.portal.get_registry_record`

api.content

`api.content.get`
`api.content.create`
`api.content.delete`
`api.content.copy`
`api.content.move`
`api.content.rename`
`api.content.get_uuid`
`api.content.get_state`
`api.content.transition`
`api.content.get_view`

api.user

`api.user.get`
`api.user.create`
`api.user.delete`
`api.user.get_current`
`api.user.is_anonymous`
`api.user.get_users`
`api.user.get_roles`
`api.user.get_permissions`
`api.user.grant_roles`
`api.user.revoke_roles`

api.group

`api.group.get`
`api.group.create`
`api.group.delete`
`api.group.add_user`
`api.group.remove_user`
`api.group.get_groups`
`api.group.get_roles`
`api.group.grant_roles`
`api.group.revoke_roles`

api.env

`api.env.adopt_roles`
`api.env.adopt_user`
`api.env.debug_mode`
`api.env.test_mode`

Exceptions and errors

`api.exc.PloneApiError`
`api.exc.MissingParameterError`
`api.exc.InvalidParameterError`
`api.exc.CannotGetPortalError`

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.2 plone.api.portal

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.3 plone.api.content

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.4 plone.api.user

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.5 plone.api.group

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.6 plone.api.env

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.3.7 plone.api.exc

1.4 Contribute

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

1.4.1 How to contribute to this package?

Conventions

Rules and guidelines on syntax style, development process, repository workflow, etc.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

Conventions

Introduction We've modeled the following rules and recommendations based on the following documents:

- PEP8
- PEP257
- Rope project
- Google Style Guide
- Pylons Coding Style
- Tim Pope on Git commit messages

Line length All Python code in this package should be PEP8 valid. This includes adhering to the 80-char line length. If you absolutely need to break this rule, append `# noPEP8` to the offending line to skip it in syntax checks.

Note: Configuring your editor to display a line at 79th column helps a lot here and saves time.

Note: The line length rule also applies to non-python source files, such as documentation `.rst` files or `.zcml` files, but is a bit more relaxed there.

Breaking lines Based on code we love to look at (Pyramid, Requests, etc.), we allow the following two styles for breaking long lines into blocks:

1. Break into next line with one additional indent block.

```
foo = do_something(
    very_long_argument='foo', another_very_long_argument='bar')

# For functions the ): needs to be placed on the following line
```

```
    def some_func(
        very_long_argument='foo', another_very_long_argument='bar'
    ):
```

2. If this still doesn't fit the 80-char limit, break into multiple lines.

```
    foo = dict(
        very_long_argument='foo',
        another_very_long_argument='bar',
    )

    a_long_list = [
        "a_fairly_long_string",
        "quite_a_long_string_indeed",
        "an_exceptionally_long_string_of_characters",
    ]
```

- Arguments on first line, directly after the opening parenthesis are forbidden when breaking lines.
- The last argument line needs to have a trailing comma (to be nice to the next developer coming in to add something as an argument and minimize VCS diffs in these cases).
- The closing parenthesis or bracket needs to have the same indentation level as the first line.
- Each line can only contain a single argument.
- The same style applies to dicts, lists, return calls, etc.

This package follows all rules above, check out the source to see them in action.

Quoting For strings and such prefer using single quotes over double quotes. The reason is that sometimes you do need to write a bit of HTML in your python code, and HTML feels more natural with double quotes so you wrap HTML string into single quotes. And if you are using single quotes for this reason, then be consistent and use them everywhere.

There are two exceptions to this rule:

- docstrings should always use double quotes (as per PEP-257).
- if you want to use single quotes in your string, double quotes might make more sense so you don't have to escape those single quotes.

```
# GOOD
print 'short'
print 'A longer string, but still using single quotes.'

# BAD
print "short"
print "A long string."

# EXCEPTIONS
print "I want to use a 'single quote' in my string."
"""This is a docstring."""
```

Docstrings style Read and follow http://www.python.org/dev/peps/pep-0257/. There is one exception though: We reject BDFL's recommendation about inserting a blank line between the last paragraph in a multi-line docstring and its closing quotes as it's Emacs specific and two Emacs users here on the Beer & Wine Sprint both support our way.

The content of the docstring must be written in the active first-person form, e.g. "Calculate X from Y" or "Determine the exact foo of bar".

```
def foo():
    """Single line docstring."""

def bar():
    """Multi-line docstring.

    With the additional lines indented with the beginning quote and a
    newline preceding the ending quote.
    """
```

If you wanna be extra nice, you are encouraged to document your method's parameters and their return values in a reST field list syntax.

```
:param foo: blah blah
:type foo: string
:param bar: blah blah
:type bar: int
:returns: something
```

Check out the plone.api source for more usage examples. Also, see the following for examples on how to write good *Sphinxy* docstrings: http://stackoverflow.com/questions/4547849/good-examples-of-python-docstrings-for-sphinx.

Unit tests style Read http://www.voidspace.org.uk/python/articles/unittest2.shtml to learn what is new in `unittest2` and use it.

This is not true for in-line documentation tests. Those still use old unittest test-cases, so you cannot use `assertIn` and similar.

String formatting As per http://docs.python.org/2/library/stdtypes.html#str.format, we should prefer the new style string formatting (`.format()`) over the old one (`% ()`).

Also use numbering, like so:

```
# GOOD
print "{0} is not {1}".format(1, 2)
```

and *not* like this:

```
# BAD
print "{} is not {}".format(1, 2)
print "%s is not %s" % (1, 2)
```

because Python 2.6 supports only explicitly numbered placeholders.

About imports

1. Don't use `*` to import *everything* from a module, because if you do, pyflakes cannot detect undefined names (W404).
2. Don't use commas to import multiple things on a single line. Some developers use IDEs (like Eclipse) or tools (such as mr.igor) that expect one import per line. Let's be nice to them.
3. Don't use relative paths, again to be nice to people using certain IDEs and tools. Also *Google Python Style Guide* recommends against it.

```
    # GOOD
    from plone.app.testing import something
    from zope.component import getMultiAdapter
    from zope.component import getSiteManager
```

instead of

```
    # BAD
    from plone.app.testing import *
    from zope.component import getMultiAdapter, getSiteManager
```

4. Don't catch `ImportError` to detect whether a package is available or not, as it might hide circular import errors. Instead, use `pkg_resources.get_distribution` and catch `DistributionNotFound`. More background at http://do3.cc/blog/2010/08/20/do-not-catch-import-errors,-use-pkg_resources/.

```
    # GOOD
    import pkg_resources

    try:
        pkg_resources.get_distribution('plone.dexterity')
    except pkg_resources.DistributionNotFound:
        HAS_DEXTERITY = False
    else:
        HAS_DEXTERITY = True
```

instead of

```
    # BAD
    try:
        import plone.dexterity
        HAVE_DEXTERITY = True
    except ImportError:
        HAVE_DEXTERITY = False
```

Grouping and sorting Since Plone has such a huge code base, we don't want to lose developer time figuring out into which group some import goes (standard lib?, external package?, etc.). So we just sort everything alphabetically and insert one blank line between `from foo import bar` and `import baz` blocks. Conditional imports come last. Again, we *do not* distinguish between what is standard lib, external package or internal package in order to save time and avoid the hassle of explaining which is which.

As for sorting, it is recommended to use case-sensitive sorting. This means uppercase characters come first, so "Products.*" goes before "plone.*".

```
# GOOD
from __future__ import division
from Acquisition import aq_inner
from Products.CMFCore.interfaces import ISiteRoot
from Products.CMFCore.WorkflowCore import WorkflowException
from plone.api import portal
from plone.api.exc import MissingParameterError

import pkg_resources
import random

try:
    pkg_resources.get_distribution('plone.dexterity')
except pkg_resources.DistributionNotFound:
    HAS_DEXTERITY = False
else:
    HAS_DEXTERITY = True
```

Declaring dependencies All direct dependencies should be declared in `install_requires` or `extras_require` sections in `setup.py`. Dependencies, which are not needed for a production environment (like "develop" or "test" dependencies) or are optional (like "Archetypes" or "Dexterity" flavors of the same package) should go in `extras_require`. Remember to document how to enable specific features (and think of using `zcml:condition` statements, if you have such optional features).

Generally all direct dependencies (packages directly imported or used in ZCML) should be declared, even if they would already be pulled in by other dependencies. This explicitness reduces possible runtime errors and gives a good overview on the complexity of a package.

For example, if you depend on `Products.CMFPlone` and use `getToolByName` from `Products.CMFCore`, you should also declare the `CMFCore` dependency explicitly, even though it's pulled in by Plone itself. If you use namespace packages from the Zope distribution like `Products.Five` you should explicitly declare `Zope` as dependency.

Inside each group of dependencies, lines should be sorted alphabetically.

Versioning scheme For software versions, use a sequence-based versioning scheme:

```
MAJOR.MINOR[.MICRO][STATUS]
```

For more information, read http://semver.org/.

Restructured Text versus Plain Text Use the Restructured Text (`.rst` file extension) format instead of plain text files (`.txt` file extension) for all documentation, including doctest files. This way you get nice syntax highlighting and formating in recent text editors, on GitHub and with Sphinx.

Tracking changes Feature-level changes to code are tracked inside `CHANGES.rst`. The title of the `CHANGES.rst` file should be `Changelog`. Example:

```
Changelog
=========

1.0.0-dev (Unreleased)
----------------------

- Added feature Z.
  [github_userid1]

- Removed Y.
  [github_userid2]


1.0.0-alpha.1 (2012-12-12)
--------------------------

- Fixed Bug X.
  [github_userid1]
```

Add an entry every time you add/remove a feature, fix a bug, etc. on top of the current development changes block.

Sphinx Documentation Un-documented code is broken code.

For every feature you add to the codebase you should also add documentation for it to `docs/`.

After adding/modifying documentation, run `make` to re-generate your docs.

Publicly available documentation on http://api.plone.org is automatically generated from these source files, periodically. So when you push changes to master on GitHub you should soon be able to see them published on `api.plone.org`.

Read the reStructuredText Primer to brush up on your *reST* skills.

Example:

```
def add(a, b):
    """Calculate the sum of the two parameters.

    Also see the :func:`mod.path.my_func`, :meth:`mod.path.MyClass.method`
    and :attr:`mod.path.MY_CONSTANT` for more details.

    :param a: The first operand.
    :type a: :class:`mod.path.A`

    :param b: The second operand.
    :type b: :class:`mod.path.B`

    :rtype: int
    :return: The sum of the operands.
    :raise: `KeyError`, if the operands are not the correct type.
    """
```

Attributes are documented using the *#:* marker above the attribute. The documentation may span multiple lines.

```
#: Description of the constant value
MY_CONSTANT = 0xc0ffee

class Foobar(object):

    #: Description of the class variable which spans over
    #: multiple lines
    FOO = 1
```

Travis Continuous Integration On every push to GitHub, Travis runs all tests and syntax validation checks and reports build outcome to the `#sprint` IRC channel and the person who committed the last change.

Travis is configured with the `.travis.yml` file located in the root of this package.

Git workflow & branching model Our repository on GitHub has the following layout:

- **feature branches**: all development for new features must be done in dedicated branches, normally one branch per feature,
- **master branch**: when features get completed they are merged into the master branch; bugfixes are commited directly on the master branch,
- **tags**: whenever we create a new release we tag the repository so we can later re-trace our steps, re-release versions, etc.

Release process for Plone packages To keep the Plone software stack maintainable, the Python egg release process must be automated to high degree. This happens by enforcing Python packaging best practices and then making automated releases using the zest.releaser tool.

- Anyone with necessary PyPi permissions must be able to make a new release by running the `fullrelease` command

... which includes ...

- All releases must be hosted on PyPi
- All versions must be tagged at version control
- Each package must have README.rst with links to the version control repository and issue tracker
- CHANGES.txt (docs/HISTORY.txt in some packages) must be always up-to-date and must contain list of functional changes which may affect package users.
- CHANGES.txt must contain release dates
- README.rst and CHANGES.txt must be visible on PyPi
- Released eggs must contain generated gettext .mo files, but these files must not be committed to the repository (files can be created with *zest.pocompile* addon)
- `.gitignore` and `MANIFEST.in` must reflect the files going to egg (must include page template, po files)

More information

- High quality automated package releases for Python with zest.releaser.

Setting up Git Git is a very useful tool, especially when you configure it to your needs. Here are a couple of tips.

Enhanced git prompt Do one (or more) of the following:

- http://clalance.blogspot.com/2011/10/git-bash-prompts-and-tab-completion.html
- http://en.newinstance.it/2010/05/23/git-autocompletion-and-enhanced-bash-prompt/
- http://gitready.com/advanced/2009/02/05/bash-auto-completion.html

Git dotfiles Plone developers have dotfiles similar to these: https://github.com/plone/plone.dotfiles.

Git Commit Message Style Tim Pope's post on Git commit message style is widely considered the gold standard:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here
- Use a hanging indent
```

Github flavored markdown is also useful in commit messages.

Local development environment

Setting up and using the local development environment.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

Development environment

This section is meant for contributors to the *plone.api* project. Its purpose is to guide them through the steps needed to start contributing.

Prerequisites

System libraries First let's look at 'system' libraries and applications that are normally installed with your OS packet manager, such as apt, aptitude, yum, etc.:

- `libxml2` - An xml parser written in C.
- `libxslt` - XSLT library written in C.
- `git` - Version control system.
- `gcc` - The GNU Compiler Collection.
- `g++` - The C++ extensions for gcc.
- `GNU make` - The fundamental build-control tool.
- `GNU tar` - The (un)archiving tool for extracting downloaded archives.
- `bzip2` and `gzip` decompression packages - `gzip` is nearly standard, however some platforms will require that `bzip2` be installed.
- `Python 2.7` - Linux distributions normally already have it, OS X users should use https://github.com/collective/buildout.python to get a clean Python version (the one that comes with OS X is broken).

Python tools Then you'll also need to install some Python specific tools:

- easy_install - the Python packaging system (download http://peak.telecommunity.com/dist/ez_setup.py and run `sudo python2.7 ez_setup.py`.
- virtualenv - a tool that assists in creating isolated Python working environments. Run `sudo easy_install virtualenv` after your have installed *easy_install* above.

Note: Again, OS X users should use https://github.com/collective/buildout.python, it will make your life much easier to have a cleanly compiled Python instead of using the system one that is broken in many deeply confusing ways.

Further information If you experience problems read through the following links as almost all of the above steps are required for a default Plone development environment:

- http://plone.org/documentation/tutorial/buildout

- http://pypi.python.org/pypi/zc.buildout/
- http://pypi.python.org/pypi/setuptools
- http://plone.org/documentation/manual/installing-plone

If you are an OS X user, you first need a working Python implementation (the one that comes with the operating system is broken). Use https://github.com/collective/buildout.python and be happy. Also applicable to other OSes, if getting a working Python proves a challenge.

Creating and using the development environment Go to your projects folder and download the lastest *plone.api* code:

```
[you@local ~]$ cd <your_work_folder>
[you@local work]$ git clone https://github.com/plone/plone.api.git
```

Now *cd* into the newly created directory and build your environment:

```
[you@local work]$ cd plone.api
[you@local plone.api]$ make
```

Go make some tea while *make* creates an isolated Python environment in your `plone.api` folder, bootstraps *zc.buildout*, fetches all dependencies, builds Plone, runs all tests and generates documentation so you can open it locally later on.

Other commands that you may want to run:

```
[you@local plone.api]$ make tests  # run all tests and syntax validation
[you@local plone.api]$ make docs   # re-generate documentation
[you@local plone.api]$ make clean  # reset your env back to a fresh start
[you@local plone.api]$ make        # re-build env, generate docs, run tests
```

Open `Makefile` in your favorite code editor to see all possible commands and what they do. And read http://www.gnu.org/software/make/manual/make.html to learn more about *make*.

Working on an issue Our GitHub account contains a list of open issues. Click on one that catches your attention. If the issue description says `No one is assigned` it means no-one is already working on it and you can claim it as your own. Click on the button next to the text and make yourself the one assigned for this issue.

Based on our *Git workflow & branching model* all new features must be developed in separate git branches. So if you are not doing a very trivial fix, but rather adding new features/enhancements, you should create a *feature branch*. This way your work is kept in an isolated place where you can receive feedback on it, improve it, etc. Once we are happy with your implementation, your branch gets merged into *master* at which point everyone else starts using your code.

```
[you@local plone.api]$ git checkout master  # go to master branch
[you@local plone.api]$ git checkout -b issue_17  # create a feature branch
# replace 17 with the issue number you are working on

# change code here

[you@local plone.api]$ git add -p && git commit  # commit my changes
[you@local plone.api]$ git push origin issue_17  # push my branch to GitHub
# at this point others can see your changes but they don't get effected by
them; in other words, others can comment on your code without your code
changing their development environments
```

Read more about Git branching at http://learn.github.com/p/branching.html. Also, to make your git nicer, read the *Setting up Git* chapter.

Once you are done with your work and you would like us to merge your changes into master, go to GitHub to do a *pull request*. Open a browser and point it to `https://github.com/plone/plone.api/tree/issue_<ISSUE_NUMBER>`. There you should see a `Pull Request` button. Click on it, write some text about what you did and anything else you would like to tell the one who will review your work, and finally click `Send pull request`. Now wait that someone comes by and merges your branch (don't do it yourself, even if you have permissions to do so).

An example pull request text:

```
Please merge my branch that resolves issue #13, where I added the
get_navigation_root() method.
```

Commit checklist Before every commit you should:

- Run unit tests and syntax validation checks.
- Add an entry to *Tracking changes* (if applicable).
- Add/modify *Sphinx Documentation* (if applicable).

All syntax checks and all tests can be run with a single command. This command also re-generates your documentation.

```
$ make
```

Note: It pays off to invest a little time to make your editor run *pep8* and *pyflakes* (of *flake8* which combines both) on a file every time you save that file. This saves you lots of time in the long run.

Releasing a new version

Description of our release process and guidelines.

GitHub-only

WARNING: If you are reading this on GitHub, DON'T! Read the documentation at api.plone.org so you have working references and proper formatting.

Releasing a new version

Releasing a new version of *plone.api* involves the following steps:

1. Prepare source for a new release.
2. Create a git tag for the release.
3. Push the git tag upstream to GitHub.
4. Generate a distribution file for the package.
5. Upload the generated package to Python Package Index (PyPI).
6. Tell ReadTheDocs to display the latest version of docs by default.

To avoid human errors and to automate some of the tasks above we use `jarn.mkrelease`. It's listed as a dependency in `setup.py` and should already be installed in your local bin:

```
$ bin/mkrelease --help
```

Apart from that, in order to be able to upload a new version to PyPI you need to be listed under *Package Index Owner* list and you need to configure your PyPI credentials in the `~/.pypirc` file, e.g.:

```
[distutils]
index-servers =
  pypi

[pypi]
username = fred
password = secret
```

Checklist Folow these step to create a new release of *plone.api*.

1. Verify that we have documented all changes in the `CHANGES.rst` file. Go through the list of commits since last release on GitHub and check all changes are documented.
2. Modify the version identifier in the `setup.py` to reflect the version of the new release.
3. Confirm that the package description (generated from `README.rst` and others) renders correctly by running `bin/longtest` and open its ouput in your favorite browser.
4. Commit all changes to the git repository and push them upstream to GitHub.
5. Create a release, tag it in git and upload it to GitHub by running `bin/mkrelease -d pypi -pq .` (see example below).

Example In the following example we are releasing version 0.1 of *plone.api*. The package has been prepared so that `version.txt` contains the version `0.1`, this change has been committed to git and all changes have been pushed upstream to GitHub:

```
# Check that package description is rendered correctly
$ bin/longtest

# Make a release and upload it to PyPI
$ bin/mkrelease -d pypi -pq ./
Releasing plone.api 0.1
Tagging plone.api 0.1
To git@github.com:plone/plone.api.git
* [new tag]         0.1 -> 0.1
running egg_info
running sdist
warning: sdist: standard file not found: should have one of README, README.txt
running register
Server response (200): OK
running upload
warning: sdist: standard file not found: should have one of README, README.txt
Server response (200): OK
done
```

Note: Please ignore the sdist warning about README file above. PyPI does not depend on it and it's just a bug in setupools (reported and waiting to be fixed).

Changelog

1.1.0-rc.1 (2013-10-10)

- Fix README.rst so it renders correctly on PyPI. [zupo]
- Use api.plone.org/foo redirects. [zupo]
- Add MANIFEST.in file. [hvelarde]

1.0.0-rc.3 (2013-10-09)

- Packaging issues. [zupo]

1.0.0-rc.2 (2013-10-09)

- Proof-read the docs, improved grammar and wording. [cewing]
- Add plone.recipe.codeanalysis to our buildout. [flohcim]
- Make all assertRaise() calls use the *with* keyword. [winstonf88]
- Amend user.get method to accept a userid parameter, refs #112. [cewing, xiru, winstonf88]

  Note: This change fixes a bug in the earlier implementation that could cause errors in some situations. This situation will only arise if the userid and username for a user are not the same. If membrane is being used for content- based user objects, or if email-as-login is enabled *and* a user has changed their email address this will be the case. In the previous implementation the username parameter was implicitly being treated as userid. The new implementation does not do so. If consumer code is relying on this bug and passing userid, and if that code uses the username parameter as a keyword parameter, then lookup will fail. In all other cases, there should be no difference.

- Add api.env.debug_mode() and api.env.test_mode(), refs #125. [sdelcourt]
- Move most of text from docs/index.rst to README.rst so its also visible on PyPI and GitHub. [zupo]
- Deprecate plone.api on ReadTheDocs and redirect to api.plone.org, refs #130. [wormj, zupo]
- Add a new *make coverage* command and add support for posting coverage to Coveralls.io. [zupo]
- Make api.content.create() also print out the underlying error, refs #118. [winston88]
- Fix api.content copy/move/rename functions to return the object after they change content, refs #115. [rodfersou]
- Make Travis IRC notification message to be one-line instead of three-lines. [zupo]
- More examples of good and bad code blocks in documentation, more information on how to write good docstrings. [zupo]
- Prefer single quotes over double quotes in code style. [zupo]
- New bootstrap.py to stay in the land of zc.buildout 1.x. [zupo]
- Package now includes a copy of the GPLv2 license as stated in the GNU General Public License documentation. [hvelarde]
- Fixed copying folderish objects. [pingviini]
- Fixed moving folderish objects. [pingviini]

1.0.0-rc.1 (2013-01-27)

- Increase test coverage. [cillianderoiste, JessN, reinhardt, zupo]
- Implementation of `api.env.adopt_roles()` context manager for temporarily switching roles inside a block. [RichyB]
- Created `api.env` module for interacting with global environment. [RichyB]
- Decorators for defining constraints on api methods. Depend on *decorator* package. [JessN]
- Resolved #61: Improve api.portal.get(). [cillianderoiste]
- Use plone.api methods in plone.api codebase. [zupo]
- Switch to *flake8* instead of *pep8'+'pyflakes*. [zupo]
- Get the portal path with absolute_url_path. [cillianderoiste]
- Travis build speed-ups. [zupo]
- Support for Python 2.6. [RichyB, zupo]
- Support for Plone 4.0. [adamcheasley]
- Support for Plone 4.3. [cillianderoiste, zupo]
- Spelling fixes. [adamtheturtle]
- Make get_view and get_tool tests not have hardcoded list of *all* expected values. [RichyB, cillianderoiste]
- Code Style Guide. [iElectric, cillianderoiste, marciomazza, RichyB, thet, zupo]
- Depend on `manuel` in setup.py. [zupo]
- Documentation how to get/set member properties. [zupo]
- Improvements to `get_registry_record`. [zupo]

0.1b1 (2012-10-23)

- Contributors guide and style guide. [zupo]
- Enforce PEP257 for docstrings. [zupo]
- Fix `get_navigation_root()` to return object instead of path. [pbauer]
- Implementation of `get_permissions()`, `get_roles()`, `grant_roles()` and `revoke roles()` for users and groups. [rudaporto, xiru]
- Implementation of `get_registry_record` and `set_registry_record`. [pbauer]
- Use *Makefile* to build the project, run tests, generate documentation, etc. [witsch]
- Moving all ReadTheDocs dependencies into `rtd_requirements.txt`. [zupo]

0.1a2 (2012-09-03)

- Updated release, adding new features, test coverage, cleanup & refactor. [hvelarde, avelino, ericof, jpgimenez, xiru, macagua, zupo]

0.1a1 (2012-07-13)

- Initial release. [davisagli, fulv, iElectric, jcerjak, jonstahl, kcleong, mauritsvanrees, wamdam, witsch, zupo]

1.5 Indices and tables

- genindex
- modindex
- search

Python Module Index

Index