plone.api Documentation

Release 1.0.0-rc.1

October 05, 2013

Contents

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/index.html so you have working
references and proper formatting.

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 be transitional. It points out the parts of Plone which are particularly nasty – we hope they will get fixed so that we can deprecate the *plone.api* methods that cover them up, but the documentation can still be useful.

Some parts of the documentation already are this way: they don't use *plone.api* methods directly, but simply provide guidance on achiving a task using Plone's internals. Example: usage of the catalog in *Find content object*.

The intention is to cover 20% of the tasks we do 80% of the time. Keeping everything in one place helps keep the API introspectable and discoverable, which are important aspects of being Pythonic.

Note: This package is still under development, but should be fairly stable and is already being used in production. It's currently a release candidate, meaning that we don't intend to change method signatures, but it may still happen.

CHAPTER 1

Narrative documentation

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/about.html so you have working
references and proper formatting.

1.1 About

1.1.1 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 important 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 As of this writing, this is still a very new library, but just looking at a comparison between the urllib2 way and the requests way, as well as the rest of its documentation, one cannot but see a parallel between the way we *have been* and the way we *should be* writing code for Plone (or at least have that option).

The API provides grouped functional access to otherwise distributed logic in Plone. Plone's original distribution of logic is a result of two things: The historic re-use of CMF- and Zope-methods and reasonable, but at first hard to understand splits like acl_users.* and portal_memberdata.

That's why we've created a set of useful methods that implement best-practice access to the original distributed APIs. In this way we also document in code how to use Plone 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.

Also, we don't intend to cover all possible use-cases. Only the most common ones. If you need to do something that *plone.api* does not support, just use the underlying APIs directly. We will cover 20% of tasks that are being done 80% of the time, and not one more. We try to document sensible use cases even when we don't provide them, though.

1.1.2 Design decisions

Import and usage style

API methods are grouped by their field of usage. For example: *Portal*, *Content*, *Users* and *Groups*. Hence the importing and usage of API methods look 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')
```

In other words, 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, so we encourage one and only one preferred way of consuming API methods.

Prefer keyword arguments

For the following reasons the example code in the API (and hence the recommendation to people on how to use it) shall always prefer using keyword instead of positional arguments:

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. This allows us to change implementation details and leave the signature unchanged. In other words, the underlying API code can change substantially and the code using it will remain valid.
4. The arguments can all be passed as a dictionary.

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

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

1.1.3 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.

But telling developers that they will get yet another object from the API which isn't the requested object, but an API-wrapped one instead, would be very hard. Also, making this wrap transparent in order to make the returned object directly usable would be nearly impossible, because we'd have to proxy all the `zope.interface` stuff, annotations and more.

Furthermore, we want to avoid people writing code like this in tests or their internal utility code and failing miserably in the future if wrappers would no longer be needed and would therefore be removed:

```
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*
- `CRUD` has *delete*, not *remove*.

1.1.4 Roadmap

Medium- to long-term:

Below is a collection of ideas we have for the long run, in no particular order:

- api.env.adopt_user (to use with `with`, especially in tests):

  ```
  with api.env.adopt_user('admin'):
      api.context.create(
          type='Document',
          title='Exhaustive list of kittens',
          container=portal
      )
      # Should leave behind a Document object owned by the user 'admin'.
  ```

- api.env TEST_MODE and DEBUG_MODE

  ```
  if api.env.TEST_MODE:
      # you are now in test environment

  if api.env.DEBUG_MODE:
      # you are now in development environment
  ```

- api.env.versions: don't do a wrapper, just explain how to use pkg_resources to query for installed versions
- unify permissions
  - have all different types of permission in one place and one way to use them
- rewrite sub-optimal underlying APIs and deprecate plone.api methods, but leave the (updated) documentation:
  - getting/setting member properties
  - tools:

    * portal_groupdata, portal_groups, portal_memberdata, portal_membership
    * portal_quickinstaller, portal_undo

- JSON webservices
  - probably in a separate package plone.jsonapi
  - one view (@@jsonapi for example) that you can call in your JS and be sure it won't change
  - easier to AJAXify stuff
- Flask-type url_for_view() and view_for_url()

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs: http://ploneapi.readthedocs.org/en/latest/portal.html so you have working references and proper formatting.

1.2 Portal

1.2.1 Get portal object

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

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

1.2.2 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.

1.2.3 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()
```

1.2.4 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')
```

1.2.5 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()
api.portal.get_localized_time(datetime=today)
```

1.2.6 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!",
)
```

1.2.7 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)
```

1.2.8 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')
```

1.2.9 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)
```

1.2.10 Further reading

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

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/content.html so you have working
references and proper formatting.

1.3 Content

1.3.1 Create content

First get the portal object that we will use as a container for new content:

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

If you want to create a new content item, use the `api.content.create()` method. The type attribute will automatically decide which content type (dexterity, archetype, ...) should be created.

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

The `id` of the object gets generated (in a safe way) from its `title`.

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

1.3.2 Get content object

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

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

You can do the following operations to get to various content objects in 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()
conference = api.content.get(UID=uid)
```

1.3.3 Find content object

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 about how to use the catalog and what parameters it supports is written 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'
```

1.3.4 Get content object UUID

An 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 content-to-content references and for linking by UIDs, enabling persistent links.

To get a content object UUID 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)
```

1.3.5 Move content

To move content around the portal structure defined above use `api.content.move()` The code below moves the `contact` item (with all objects that it contains) out of folder `about` 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, you can define to what target ID the object will be moved to. Otherwise it will be moved with the same ID that it had.

1.3.6 Rename content

To rename, use the `api.content.rename()` method.

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

1.3.7 Copy content

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

```
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 (if not stated otherwise). 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 non-conflicting id.

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

1.3.8 Delete content

Deleting content works by passing the object you want to delete to the `api.content.delete()` method:

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

1.3.9 Content manipulation with the *safe_id* option

When manipulating content with `api.content.create()`, `api.content.move()` and `api.content.copy()` the *safe_id* flag is disabled by default. This means the id will be enforced, if the id is taken on the target container the API method will raise an error.

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

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

1.3.10 Get workflow state

To find out in which workflow state your content is, use `api.content.get_state()`.

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

1.3.11 Transition

To transition your content into a new state, use `api.content.transition()`.

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

1.3.12 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,
)
```

1.3.13 Further reading

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

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/user.html so you have working
references and proper formatting.

1.4 Users

1.4.1 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-char alphanumeric password will be generated.

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

1.4.2 Get user

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

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

1.4.3 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')
```

1.4.4 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()
```

1.4.5 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
```

1.4.6 Get all users

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

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

1.4.7 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')
```

1.4.8 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)
```

1.4.9 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'])
```

1.4.10 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'])
```

1.4.11 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 conext and not site wide. But all site wide roles will be also returned by `api.user.get_roles()` to 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']
)
```

1.4.12 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 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'],
)
```

1.4.13 Further reading

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

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs: http://ploneapi.readthedocs.org/en/latest/group.html so you have working references and proper formatting.

1.5 Groups

1.5.1 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', ],
)
```

1.5.2 Get group

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

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

1.5.3 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',
)
```

1.5.4 Get all groups

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

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

1.5.5 Get user's groups

If you set the *user* parameter, then `api.group.get_groups()` will return groups that the user is member of.

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

1.5.6 Get group members

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

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

1.5.7 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)
```

1.5.8 Adding user to group

The `api.group.add_user()` method accepts either the groupname or the group object of 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')
```

1.5.9 Removing user from group

The `api.group.remove_user()` method accepts either the groupname or the group object of 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')
```

1.5.10 Get group roles

The `api.group.get_roles()` method is used for getting a group's roles. 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 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'])
```

1.5.11 Grant roles to group

The `api.group.grant_roles()` allows us to grant a list of roles site-wide to the group.

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

If you pass in a content object, it will grant these roles 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'])
```

1.5.12 Revoke roles from group

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

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

If you pass in a content object, it will grant these roles in that particular context.

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

1.5.13 Further reading

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

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs: http://ploneapi.readthedocs.org/en/latest/env.html so you have working references and proper formatting.

1.6 Environment

1.6.1 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")
```

1.6.2 Further reading

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

Complete API and advanced usage

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/api.html so you have working
references and proper formatting.

2.1 List of all API methods with descriptions

2.1.1 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`

2.1.2 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`

2.1.3 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`

2.1.4 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`

2.1.5 api.env

`api.env.adopt_roles`

2.1.6 Exceptions and errors

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

2.2 plone.api.portal

2.3 plone.api.content

2.4 plone.api.user

2.5 plone.api.group

2.6 plone.api.env

2.7 plone.api.exc

Contributing

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/about.html so you have working
references and proper formatting.

3.1 How to contribute to this package?

3.1.1 Conventions

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

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/contribute/conventions.html so you
have working references and proper formatting.

Conventions

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.

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.

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.

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 (`% ()`).

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 stuff on a single line. Some developers use IDEs (like Eclipse <http://pydev.org/>_) 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.

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

instead of

```
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. Instead, use pkg_resources.get_distribution and catch DistributionNotFound.

```
import pkg_resources

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

instead of

```
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 loose 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.

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

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 `docs/CHANGES.rst`. Example:

```
CHANGES
=======

1.0dev (unreleased)
-------------------

- Added feature Z.
  [github_userid1]

- Removed Y.
  [github_userid2]


1.0a1 (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.

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.

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.

3.1.2 Local development environment

Setting up and using the local development environment.

WARNING: If you are reading this on GitHub, DON'T! Read it on ReadTheDocs:
http://ploneapi.readthedocs.org/en/latest/contribute/develop.html 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.

3.1.3 Releasing a new version

Description of our release process and guidelines.

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).
6. Go to `https://readthedocs.org/dashboard/ploneapi/versions/` and choose the latest version to be displayed as the default version.

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).

3.1.4 Changes

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]

CHAPTER 4

Indices and tables

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

Python Module Index