senaite.queue Documentation

Release 1.0.3

Riding Bytes
Naralabs

Jul 01, 2022

Contents

This add-on enables asynchronous tasks for SENAITE LIMS, that allows to better handle concurrent actions and processes when the workload is high. Is specially indicated for high-demand instances and for when there are custom processes that take long to complete. Essentially, senaite.queue reduces the chance of transaction commits by handling tasks asynchronously, in an unattended and sequential manner.

Once installed, this add-on enables asynchronous processing of those tasks that usually have a heavier footprint regarding performance, and with highest chance of transaction conflicts:

- Assignment of analyses to worksheets
- Assignment of worksheet template to a worksheet
- Creation of a worksheet by using a worksheet template
- Workflow actions (submit, verify, etc.) for analyses assigned to worksheets
- Recursive permissions assignment on client contacts creation

This add-on neither provides support for workflow transitions/actions at Sample level nor for Sample creation. However, this add-on can be extended easily to match additional requirements.

This documentation is divided in different parts. We recommend that you get started with *Installation* and then head over to the *Quickstart*. Please check out *Tasks handling* and *Doctests* for internals about *senaite.queue*.

Table of Contents:

CHAPTER 1

Installation

Is strongly recommended to have a SENAITE instance setup in ZEO mode, because this add-on is especially useful when a reserved zeo client is used to act as a queue server and at least one additional zeo client for tasks consumption.

In standalone installation, only one CPU / CPU core can be used for processing requests, with a limited number of threads (usually 2). With a ZEO mode setup, the database can be used by multiple zeo clients at the same time, each one using it's own CPU. See Scalability and ZEO for further information.

Create a new reserved user in SENAITE instance (under */senaite/acl_users*). The recommended username is *queue_consumer*.

This user will be used by the consumer to pop tasks from the queue server in a sequential manner. The consumer will eventually process the task, but acting as the user who initially triggered the process. However, the reserved user responsible of dispatching must have enough privileges. Assign this user to the group "Site Administrator" and/or "Manager".

First, add this add-on in the *eggs* section of your buildout configuration file:

```
[buildout]

...

[instance]
...
eggs =
    ...
    senaite.queue
```

Then, add a two clients (a consumer and the server) in your buildout configuration:

```
# Reserved user for dispatching queued tasks
# See https://pypi.org/project/senaite.queue
queue-user-name=queue_consumer
queue-user-password=queue_consumer_password

parts =
```

(continues on next page)

(continued from previous page)

```
    ....
    queue_consumer
    queue_server
```

and configure two reserved clients:

```
[queue_consumer]
# ZEO Client reserved for the consumption of queued tasks
<= client_base
recipe = plone.recipe.zope2instance
http-address = 127.0.0.1:8089
zope-conf-additional =
    <clock-server>
        method /senaite/queue_consume
        period 5
        user ${buildout:queue-user-name}
        password ${buildout:queue-user-password}
        host localhost:8089
    </clock-server>

[queue_server]
# ZEO Client reserved to act as the server of the queue
<= client_base
recipe = plone.recipe.zope2instance
http-address = 127.0.0.1:8090
```

Note: These clients will listen to ports 8089 and 8090. They should not be accessible to regular users. Thus, if you use a load-balancer (e.g HAProxy), is strongly recommended to not add these clients in the backend pool.

In most scenarios, this configuration is enough. However, senaite.queue supports multi consumers, that can be quite useful for those SENAITE installations that have a very high overload. To add more consumers, add as many zeo client sections as you need with the additional *clock-server* zope configuration. Do not forget to set the value *host* correctly to all them, because this value is used by the queue server to identify the consumers when tasks are requested.

The maximum number of concurrent consumers supported by the queue server is 4.

Run *bin/buildout* afterwards. With this configuration, buildout will download and install the latest published release of senaite.queue from Pypi.

Note: If the buildout fails with a `ImportError: cannot import name aead`, please update OpenSSL to v1.1.1 or above. OpenSSL v1.0.2 is no longer supported by `cryptography` starting from v3.2. Please, read the changelog from cryptography for further information. Although not recommended, you can alternatively stick to version 3.1.1 by adding `cryptography=3.1.1` in `[versions]` section from your buildout.

Once buildout finishes, start the clients:

```
$ sudo -u plone_daemon bin/client1 start
$ sudo -u plone_daemon bin/queue_server start
$ sudo -u plone_daemon bin/queue_client start
```

Note: `plone_daemon` is the default user created by the quick-installer when installing Plone in ZEO cluster mode. Please check Installation of Plone for further information. You might need to change this user name depending on

how you installed SENAITE.

Then visit your SENAITE site and login with a user with "Site Administrator" privileges to activate the add-on:

http://localhost:8080/senaite/prefs_install_products_form

Note: It assumes you have a SENAITE zeo client listening to port 8080

Once activated, go to *Site Setup* > *Queue Settings* and, in field "Queue Server", type the url of the zeo client that will act as the server of the queue.

http://localhost:8090/senaite

Note: Do not forget to specify the site id in the url (usually "senaite")

CHAPTER 2

Quickstart

This section gives an introduction about senaite.queue. It assumes you have SENAITE LIMS and *senaite.queue* already installed, with a consumer listening at port 8089, the queue server listening at port 8090 and a regular zeo client listening at port 8080. Please read the *Installation* for further details.

2.1 Queue control panel

Visit the control panel view for *senaite.queue* to configure the settings. This control panel is accessible to users with *Site Administrator* role, through "Site Setup" view, "Add-on Configuration" section:

http://localhost:8080/senaite/@@queue-controlpanel

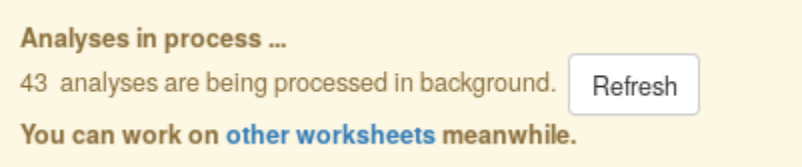
In most cases, the settings that come by default will fit well. Modifying some of them might speed-up the processing of queued tasks, but might also increase the chance of conflicts. Therefore, is strongly recommended to monitor the instance while modifying this settings:

- **Queue server**: URL of the zeo client that will act as the queue server. This is, the zeo client others will rely on regarding tasks addition, retrieval and removal. An empty value or a non-reachable queue server disables the asynchronous processing of tasks. In such case, system will behave as if senaite.queue was not installed.
- **Number of objects to process per task**: This is the default number of objects to process in a single request when the task contains multiple items. The items from a task are processed in chunks, and remaining are re-queued for later. For instance, when a user selects multiple analyses for their assignment to a worksheet, only one task is generated. If the value defined is 5, the analyses will be assigned in chunks of this size, and the system will keep generating tasks for the remaining analyses until all them are finally assigned. Higher values increment the chance of transaction commit conflicts, while lower values tend to slow down the completion of the whole task. A value of 0 disables queueing if tasks functionality at all.
- **Maximum retries**: Number of times a task will be re-queued before being considered as failed. A value of 0 disables the re-queue of failing tasks.
- **Minimum seconds**: Minimum number of seconds to book per task. If the task is performed very rapidly, it will have priority over a transaction done from userland. In case of conflict, the transaction from userland will fail and will be retried up to 3 times. This setting makes the thread that handles the task to take some time to complete, thus preventing threads from userland to be delayed or fail.

- **Maximum seconds**: Number of seconds to wait for a task to finish before being re-queued or considered as failed. System will keep retrying the task until the value set in 'Maximum retries' is reached, at which point the task will be eventually considered as failed and no further actions will take place.
- **Auth secret key**: This secret key is used by senaite.queue to generate an encrypted token (symmetric encryption) for the authentication of requests sent by queue clients and consumers to the Queue's server API. Must be 32 url-safe base64-encoded bytes.

2.2 Queueing a task

Login as a SENAITE regular user with "Lab Manager" privileges. Be sure there are some analyses awaiting for assignment and create a worksheet, either by manually assigning some analyses or by using a Worksheet Template. As soon as the worksheet is created, the system displays a viewlet stating that some analyses have been queued for the current worksheet:



Keep pressing the "Refresh" button and the message will eventually disappear, as soon as the reserved client finishes processing the task.

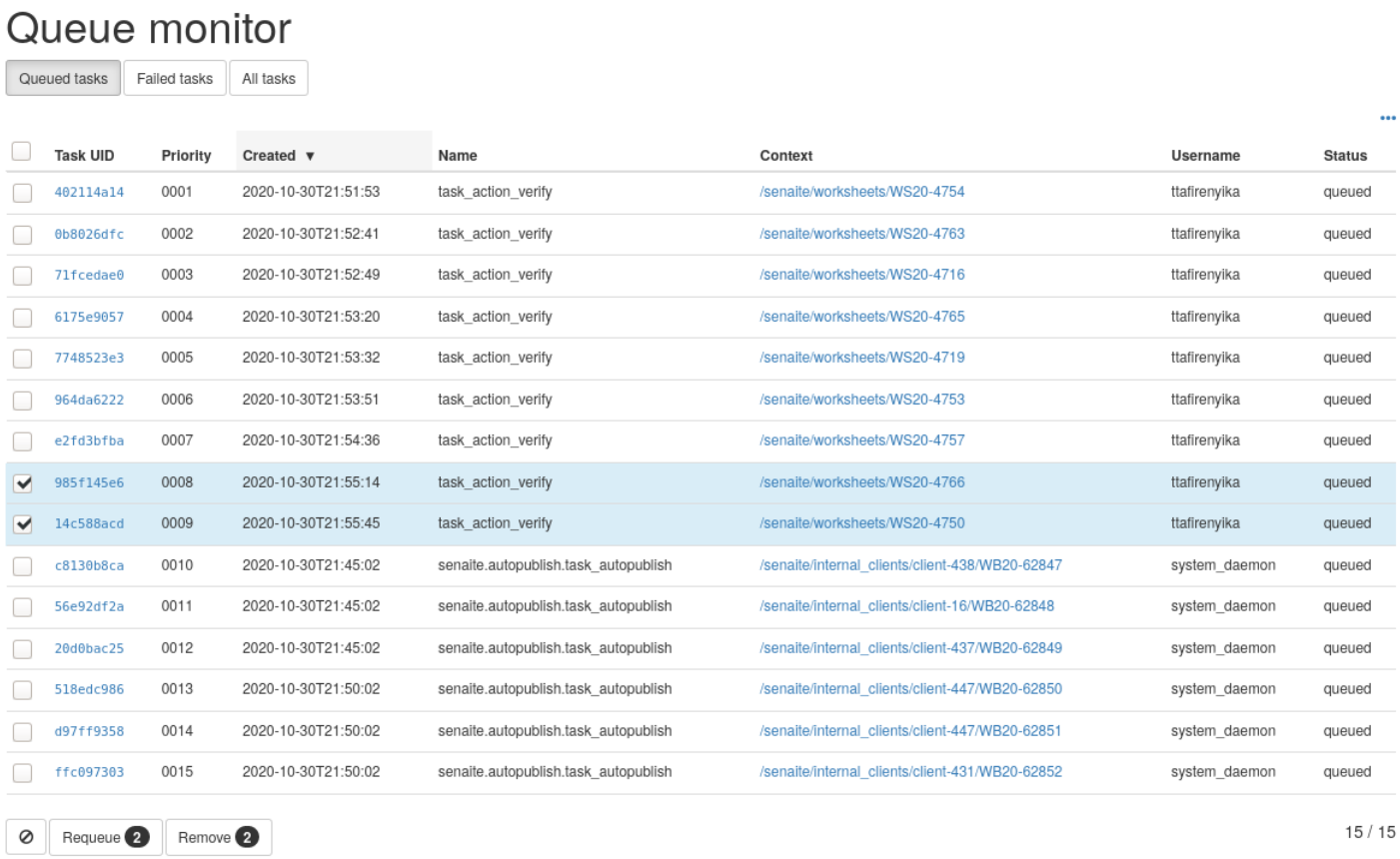
Note: If you don't see any change after refreshing the page several times, check that you have the consumer client running in background and the reserved user is properly configured.

2.3 Queue monitoring

The queue monitoring view is accessible from the top-right "hamburger" menu, link "Queue Monitor":

http://localhost:8080/senaite/queue_tasks

The failed, running and queued tasks are displayed in this view, along with their Task Unique Identifiers (TUIDs). From this view, the user can manually re-queue or remove tasks at a glance:



Failed tasks shouldn't be the norm, but there is always the chance that a task cannot complete. In order to provide insights about the reason/s behind a failure, the monitor listing displays also the error trace raised by the system when trying to process the task.

2.4 Queued task details

Given a TUID, the user can see the whole information of a given task in JSON format. The TUID of each task displayed in the Queue Monitoring view explained above is a link to the full detail of the task:


```
{
    "status": "queued",
    "context_uid": "67127b454506455f81d69921beec4e93",
    "context_path": "/senaite/worksheets/WS-018",
    "name": "task_action_submit",
    "retries": 5,
    "uids": [
        "bc0c7489fa974e74b68a680568608277",
        "7e6cc0c0de9449ca953dd8b7dfaffb96",
        "2f8f2a05faa14af19545e9f08b4b282c",
        "b2bd04cb1755493186bea52a50f37326",
        "5531c1adc95e47c38ff11c49ff8ff50b",
        "ef19831a8ef9467db401008c1269b937"
    ],
    "created": 1598626797.74663,
    "error_message": null,
    "username": "analyst1",
```


(continues on next page)

(continued from previous page)


```
    "priority": 10,
    "max_seconds": 60,
    "task_uid": "2bb771e4bb7cbcf9625bf761377292d8",
    "action": "submit",
    "min_seconds": 2
}
```


The fields displayed might vary depending on the type of task (the "name" field defines the type of the task). In the example above, the task refers to the submission (field *action*) of results for 6 analyses from worksheet with id "WS-018" (field *context_path*). This action has been triggered by the user with id "analyst1" (field *username*). The field *uids* contains the unique identifiers of the analyses to be submitted, and the *context_uid* indicates the unique identifier of the object from which the action/task was triggered.

Note: There are plenty of add-ons for browsers that beautify the generated JSON, making it's interpretation more comfortable for humans. These are some of the plugins you might consider to install in your browser: JSONView for Firefox, JSON Lite for Firefox, JSONView for Google Chrome

---

CHAPTER 3

Tasks handling

SENAITE QUEUE keeps a prioritized queue that contains the tasks to be processed. Each time the clock wakes-up (*clock-server* directive in *buildout* configuration, see *Installation*), the system checks if the queue is currently locked by the tasks consumer. If locked, the system does nothing and returns to a neutral state, awaiting for the undergoing task to finish. If the queue is not locked, the consumer pops the next task from the queue. The consumer then starts a new thread for processing the task.

As soon as the processing of the task finishes, the consumer notifies the Queue so it can return to a neutral state and dispatch next task. This task is removed from the queue.

If an error arises while processing the task, the consumer notifies the Queue about the incident as well. This time, the queue resumes to neutral state, but labels the task as "failed" and is not removed.

3.1 Prioritization

Two factors are taken into account for tasks prioritization: creation date time and task custom priority value.

By default, system applies a priority value of 10 for all type of tasks. This value can be changed for specific tasks though. The lesser the priority value, the higher will be the priority of that task over others.

However, the creation date time is also used for tasks prioritization. So, even if a task has a higher priority based on the priority value explained before, tasks that were created long before this task will be prioritized. For this to happen, the system calculates the task priority with this formula:

$$P = t + 300 * p$$

where:

- P: Priority of the task
- t: Number of seconds passed since epoch when the task was queued
- p: Priority value

For instance, given two tasks added to the queue with a difference of 5 minutes (300 seconds), the first one with a *p* of 100 and the second with a *p* of 10:

$$P_0 = 1600003935 + 300 \times 100 = 1600033935$$
$$P_1 = (1600003935 + 300) + 300 \times 10 = 1600037235$$
$$P_0 < P_1$$

In this example, the task that was added first will be processed first, although its priority value was greater than second's (remember the lesser the priority value, the more priority).

This mechanism prevents the Queue to be jeopardized by high-priority tasks when there is a lot of overload. Also, each time a new attempt for a failed task takes place, the *created* value is updated accordingly. Thus, the mechanism also acts as a safeguard for when a task takes long to complete and requires several attempts to finish: it makes room for other tasks to be processed instead of retrying the same task time again and again.

3.2 Failed tasks

The Queue discards a task as "failed" because of any of the following reasons:

- The process did not complete because of transaction commit conflicts
- The process did not complete because of other errors
- The process reached the timeout defined in settings

By default, the Queue will try to re-process the failed tasks up to 3 times. This value can be changed in *Queue control panel*. view: *Maximum retries*. When a task is considered as failed, the Queue transitions from status "locked" to "unlocked" and therefore, next task becomes available for consumption. If the process does not succeed after maximum retries is reached, the task is discarded as failed again, but no further retries will take place.

On each re-attempt, the queue sets a delay of 5 seconds, giving some time before the task is re-processed. This mechanism reduces the chance of failures and also makes room for other tasks to be processed before retries.

Also, the number of items to process for that precise task is reduced in a half. This reduces the chance of both conflicts and timeouts.

When a process does not complete successfully, the thread in charge of handling the task ends gracefully and the queue is immediately notified. This is the safest case, cause there is no risk that more threads the CPU can handle are started accidentally.

However, a process might take long to complete or maybe the zeo client was stopped while a task was being processed. These are the two scenarios the last reason refers to. In such cases, the Queue does not know if the task is actually running or is not. Still, the Queue needs to resume because otherwise, no further tasks will ever be processed: the queue would enter into a dead-lock status. The Timeout mechanism (see next section) prevents this to happen.

3.3 Timeout

When system retries a task, it will increase the timeout for that specific task. Timeout is the time in seconds the Queue will wait for the task to complete before being discarded as failed. By default, this value is set to 120 seconds, but can be changed in *Queue control panel*: *Maximum seconds*.

Given a value of timeout of 120 seconds, if a task fails the first time, the system will increase the timeout for that task to 180 seconds. If it fails a second time, it will increase its timeout to 270 seconds: the system multiplies the seconds by a factor of 1.5 each time.

Note: Note that if *Maximum retries* is set to 5 and the timeout is 120 seconds, the time in seconds the Queue will wait for the task to complete in the last attempt will be 608 seconds (10 minutes). Take this into account when configuring default values for *Maximum seconds* and *Maximum retries*.

3.4 Transaction commit conflicts

When a database transaction commit conflict takes place, the system retries the same transaction up to 3 times as per Zope's default. However, if the last transaction attempt cannot be completed, the Queue re-queues the task for further attempts, up to the value defined in *Queue control panel*: *Maximum retries*.

CHAPTER 4

Extend and Customize

This package is built to be extended. You can use the *Zope Component Architecture* to provide specific Adapters to both control how a task is processed and to indicate which processes/logic needs to be executed asynchronously by senaite.queue. The process or logic to be handled by *senaite.queue* can be from either SENAITE LIMS or from any other *SENAITE*-specific add-on.

4.1 Queued task for a workflow action

Let's imagine you have your own add-on with a custom transition/action (e.g. *dispatch*) in sample's workflow, that transitions the sample to a *dispatched* status. The user can choose multiple samples at once from the listing and transition all them at once. This functionality might entail an undesired impact on performance, specially if hundreds of samples are selected at once.

To address this functionality, we can extend *senaite.queue* in our own add-on. We are not interested in replacing the logic behind such transition, but feed the queue for this action. Therefore, we can make use of the generic adapter *WorkflowGenericQueueAdapter* that comes by default with *senaite.queue* and only do the registration in *configure.zcml*:

```
<adapter
  name="workflow_action_dispatch"
  for="*
       zope.publisher.interfaces.browser.IBrowserRequest"
  factory="senaite.queue.actions.WorkflowActionGenericQueueAdapter"
  provides="bika.lims.interfaces.IWorkflowActionAdapter"
  permission="zope.Public" />
```

This is a named adapter, and the name must be the action id with *workflow_action* prepended. When the workflow action *dispatch* is triggered, the system looks for registered adapters and if a match is found, the adapter is called. Note that *for* field is neither context-specific nor layer specific, so this adapter will always be called when the action *dispatch* is triggered, regardless of context and layer.

Alternatively, you can directly feed the queue programmatically:

```
from senaite.queue import api
api.add_action_task(objects, action)
```

Parameter objects can be either a brain, an object, a uid or a list/tuple of any of them.

4.2 Queued task for custom logic

Imagine that instead of having a workflow action "dispatch" in place, you rather have a simple view from which the user can choose samples and generate a dispatch pdf from all them at once. Basically you want to feed the queue directly by your own:

```
class DispatchSamplesView(BrowserView):

    def __call__(self):
        ...

        # Get the selected samples from the form
        uids = self.request.form.get("selected_uids", [])

        # Queue the task
        params = {"uids": uids}
        api.add_task("my.addon.task_dispatch", self.context, **params)
```

Note the following:

- We use a "uids" field to store the list of objects to be processed
- We've set a custom task id *my.addon.task_dispatch*. This task id will be used by *senaite.queue* to look for a suitable adapter able to handle tasks with this id.

Create an adapter in charge of handling the task:

```
from bika.lims import api as _api
from Products.Archetypes.interfaces.base import IBaseObject
from senaite.queue import api
from senaite.queue.queue import get_chunks_for
from senaite.queue.interfaces import IQueuedTaskAdapter

DISPATCH_TASK_ID = "my.addon.task_dispatch"

class DispatchQueuedTaskAdapter(object):
    """Adapter for dispatch transition
    """
    implements(IQueuedTaskAdapter)
    adapts(IBaseObject)

    def __init__(self, context):
        self.context = context

    def process(self, task):
        """Process the objects from the task
        """
        # If there are too many objects to process, split them in chunks to
        # prevent the task to take too much time to complete
        chunks = get_chunks_for(task)
```

(continues on next page)

(continued from previous page)

```
        # Process the first chunk
        objects = map(_api.get_object_by_uid, chunks[0])
        map(dispatch_sample, objects)

        # Add remaining objects to the queue
        params = {"uids": chunks[1]}
        api.add_task(DISPATCH_TASK_ID, self.context, **params)

    def dispatch_sample(self, sample):
        """Generates a dispatch report for this sample
        """
        # Generate the pdf here
        pdf = generate_dispatch_pdf(sample)

        # Store the pdf as an attachment to the sample
        att = _api.create(sample.aq_parent, "Attachment")
        att.setAttachmentFile(open(pdf))
        sample.setAttachment(att)
```

Register this adapter in *configure.zcml*:

```
<adapter
  name="my.addon.task_dispatch"
  factory="my.addon.adapters.DispatchQueuedTaskAdapter"
  provides="senaite.queue.interfaces.IQueuedTaskAdapter"
  for="*" />
```

Note that this adapter is not only in charge of generating the dispatch pdfs, but also splits the tasks into separate chunks preventing overload.

CHAPTER 5

Doctests

5.1 Queue API

senaite.queue comes with an api to facilitate the interaction with queue.

Running this test from the buildout directory:

bin/test test_textual_doctests -t API

5.1.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from senaite.queue import api
>>> from senaite.queue.queue import QueueTask
>>> from senaite.queue.tests import utils as test_utils
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample():
...     return test_utils.create_sample([Cu], client, contact,
...                                     sampletype, receive=False)
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
```

(continues on next page)

(continued from previous page)

```
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

5.1.2 Retrieve the Queue Utility

The queue utility is the engine from *senaite.queue* that is responsible of providing access to the queue storage. Unless the current zeo client is configured to act as the queue's server, `api.get_queue()` always returns the client queue utility:

```
>>> api.get_queue()
<senaite.queue.client.utility.ClientQueueUtility object at...
```

If we configure the current zeo client as the server, we get the server queue utility instead:

```
>>> api.is_queue_server()
False
```

```
>>> api.get_server_url()
'http://localhost:8080/senaite'
```

```
>>> key = "senaite.queue.server"
>>> plone_api.portal.set_registry_record(key, u'http://nohost/plone')
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

```
>>> api.is_queue_server()
True
```

```
>>> api.get_server_url()
'http://nohost/plone'
```

Both utility queues provide same interface and same behavior is expected, regardless of the type of `QueueUtility`. See `ClientQueueUtility.rst` and `ServerQueueUtility.rst` doctests for additional information.

5.1.3 Queue status

We can check the queue status:

```
>>> api.get_queue_status()
'ready'
```

We can even use the helper `is_queue_ready`:

```
>>> api.is_queue_ready()
True
```

Queue might be enabled, but not ready:

```
>>> api.is_queue_enabled()
True
```

5.1.4 Enable/Disable queue

The queue can be disabled and enabled from Site Setup > Queue Settings:

```
>>> key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(key, 0)
>>> api.is_queue_enabled()
False
```

```
>>> api.is_queue_ready()
False
```

```
>>> api.get_queue_status()
'disabled'
```

We can re-enable the queue by defining the default's chunk size:

```
>>> plone_api.portal.set_registry_record(key, 10)
>>> api.is_queue_enabled()
True
```

```
>>> api.is_queue_ready()
True
```

```
>>> api.get_queue_status()
'ready'
```

5.1.5 Add a task

We can add a task without the need of retrieving the queue utility or without the need of creating a `QueueTask` object:

```
>>> sample = new_sample()
>>> kwargs = {"action": "receive"}
>>> task = api.add_task("task_action_receive", sample)
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(task, QueueTask)
True
```

```
>>> api.get_queue().get_tasks()
[{...}]
```

```
>>> len(api.get_queue())
1
```

5.1.6 Add an action task

Tasks for workflow actions are quite common. Therefore, a specific function for actions is also available:

```
>>> task = api.add_action_task(sample, "submit")
>>> isinstance(task, QueueTask)
True
```

```
>>> len(api.get_queue())
2
```

5.1.7 Add assign action task

The action "assign" (for analyses) requires not only the worksheet, but also the list of analyses to be assigned and the slot positions as well. Therefore, a helper function to make it easier is also available:

```
>>> worksheet = _api.create(portal.worksheets, "Worksheet")
>>> analyses = sample.getAnalyses(full_objects=True)
>>> task = api.add_assign_task(worksheet, analyses)
>>> isinstance(task, QueueTask)
True
```

```
>>> len(api.get_queue())
3
```

5.1.8 Check if an object is queued

```
>>> new_sample = new_sample()
>>> api.is_queued(new_sample)
False
```

```
>>> api.is_queued(sample)
True
```

```
>>> api.is_queued(worksheet)
True
```

5.1.9 Flush the queue

Flush the queue to make room for other tests:

```
>>> test_utils.flush_queue(browser, self.request)
```

5.2 Server's Queue utility

The `IServerQueueUtility` is an utility that acts as a singleton and is used to store and keep track of tasks added by queue clients and the delivery of tasks to consumers.

This utility is only used by the zeo instance that acts as the Queue Server. The rest (consumers and queue clients), use `IClientQueueUtility` instead.

Running this test from the buildout directory:

bin/test test_textual_doctests -t ServerQueueUtility

5.2.1 Test Setup

Needed imports:

```
>>> import binascii
>>> import os
>>> import time
>>> from bika.lims import api as _api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from senaite.queue.interfaces import IQueueUtility
>>> from senaite.queue.interfaces import IServerQueueUtility
>>> from senaite.queue.queue import new_task
>>> from senaite.queue.tests import utils as test_utils
>>> from zope.component import getUtility
```

Functional Helpers:

```
>>> def new_sample():
...     return test_utils.create_sample([Cu], client, contact,
...                                     sampletype, receive=False)
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
```

(continues on next page)

(continued from previous page)

```
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

5.2.2 Retrieve the server's queue utility

The server queue utility provides all the functionalities required to manage the queue and the tasks from the server side. Is a global utility, acting like a singleton, and it guarantees no conflicts when multiple threads from same instance (zeo client) operate against. To retrieve the server utility:

```
>>> getUtility(IServerQueueUtility)
<senaite.queue.server.utility.ServerQueueUtility object at...
```

Server utility implements `IQueueUtility` interface too:

```
>>> utility = getUtility(IServerQueueUtility)
>>> IQueueUtility.providedBy(utility)
True
```

5.2.3 Add a task

The queue server does not have any task awaiting yet:

```
>>> utility.is_empty()
True
```

Create a new object and the task first:

```
>>> sample = new_sample()
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
```

Add the new task to the utility

```
>>> added_task = utility.add(task)
>>> added_task == task
True
```

```
>>> utility.is_empty()
False
```

```
>>> len(utility)
1
```

Only tasks from `QueueTask` type are supported:

```
>>> utility.add("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

Adding an existing task has no effect:

```
>>> dummy = utility.add(task)
>>> dummy is None
True
```

```
>>> len(utility)
1
```

However, we can add another task for same context and with same name:

```
>>> kwargs = {"action": "receive"}
>>> copy_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(copy_task) == copy_task
True
```

```
>>> len(utility)
2
```

But is not possible to add a new task for same context and task name when the `unique` wildcard is used:

```
>>> kwargs = {"action": "receive", "unique": True}
>>> unique_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(unique_task) is None
True
```

```
>>> len(utility)
2
```

5.2.4 Delete a task

We can delete a task directly:

```
>>> utility.delete(copy_task)
>>> len(utility)
1
```

Or by using its task uid:

```
>>> added = utility.add(copy_task)
>>> len(utility)
2
```

```
>>> utility.delete(copy_task.task_uid)
>>> len(utility)
1
```

5.2.5 Get a task

We can retrieve the task we added before by it's uid:

```
>>> retrieved_task = utility.get_task(task.task_uid)
>>> retrieved_task == task
True
```

If we ask for a task that does not exist, returns None:

```
>>> dummy_uid = binascii.hexlify(os.urandom(16))
>>> utility.get_task(dummy_uid) is None
True
```

If we ask for something that is not a valid uid, we get an exception:

```
>>> utility.get_task("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.2.6 Get tasks

Or we can get all the tasks the utility contains:

```
>>> tasks = utility.get_tasks()
>>> tasks
[{...}]
```

```
>>> task in tasks
True
```

```
>>> len(tasks)
1
```

5.2.7 Get tasks by status

We can even get the tasks filtered by their status:

```
>>> utility.get_tasks(status=["queued", "running"])
[{...}]
```

```
>>> utility.get_tasks(status="queued")
[{...}]
```

```
>>> utility.get_tasks(status="running")
[]
```

5.2.8 Get tasks by context

Or we can get the task by context:

```
>>> utility.get_tasks_for(sample)
[{...}]
```

```
>>> utility.get_tasks_for(_api.get_uid(sample))
[{...}]
```

```
>>> utility.get_tasks_for(task.task_uid)
[]
```

```
>>> utility.get_tasks_for("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.2.9 Get tasks by context and task name

```
>>> utility.get_tasks_for(sample, name="task_action_receive")
[{...}]
```

```
>>> utility.get_tasks_for(sample, name="dummy")
[]
```

5.2.10 Get objects uids from tasks

We can also ask for all the uids from objects the utility contains:

```
>>> uids = utility.get_uids()
>>> len(uids)
1
```

```
>>> _api.get_uid(sample) in uids
True
```

```
>>> task.task_uid in uids
False
```

5.2.11 Ask if a task exists

```
>>> utility.has_task(task)
True
```

```
>>> utility.has_task(task.task_uid)
True
```

```
>>> utility.has_task(_api.get_uid(sample))
False
```

```
>>> utility.has_task("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.2.12 Ask if a task for a context exists

```
>>> utility.has_tasks_for(sample)
True
```

```
>>> utility.has_tasks_for(_api.get_uid(sample))
True
```

```
>>> utility.has_tasks_for(task.task_uid)
False
```

```
>>> utility.has_tasks_for("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.2.13 Ask if a task for a context and name exists

```
>>> utility.has_tasks_for(sample, name="task_action_receive")
True
```

```
>>> utility.has_tasks_for(sample, name="dummy")
False
```

5.2.14 Pop a task

When a task is popped, the utility changes the status of the task to "running", cause expects the task has been popped for consumption:

```
>>> consumer_id = u'http://nohost'
>>> popped = utility.pop(consumer_id)
>>> popped.status
'running'
```

We can still add new tasks at the same time, even if they are for same context and with same name:

```
>>> kwargs = {"action": "receive"}
>>> copy_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(copy_task) == copy_task
True
```

However, the server does not allow the consumer to pop more tasks until receives an acknowledgment that the previously popped task is done:

```
>>> utility.pop(consumer_id) is None
True
```

Even if we ask again:

```
>>> utility.pop(consumer_id) is None
True
```

Unless we wait for 10 seconds, when the server assumes the consumer failed while processing the task. Consumers always check that there is no thread running from their side before doing a `pop()`. Also, a consumer (that in fact, is a zeo client) might be stopped at some point. Therefore, this timeout mechanism is used as a safety fallback to prevent the queue to enter in a dead-lock:

```
>>> time.sleep(11)
>>> utility.pop(consumer_id) is None
True
```

The previous task is now re-queued by the server:

```
>>> popped = utility.get_task(popped.task_uid)
>>> popped.status
'queued'
```

```
>>> popped.get("error_message")
'Purged on pop (http://nohost)'
```

And a `pop` returns now the next task:

```
>>> next_task = utility.pop(consumer_id)
>>> next_task.status
'running'
```

```
>>> next_task.task_uid != popped.task_uid
True
```

Delete the newly added task, so we keep only one task in the queue for testing:

```
>>> utility.delete(next_task)
>>> len(utility)
1
```

If we try now to `pop` again, the task the queue server considered as timeout won't be popped because the server adds a delay of 5 seconds before the task can be popped again. This mechanism prevents the queue to be jeopardized by recurrent failing tasks and makes room for other tasks to be processed:

```
>>> popped.get("delay")
5
```

```
>>> utility.pop(consumer_id) is None
True
```

```
>>> time.sleep(5)
>>> delayed = utility.pop(consumer_id)
>>> delayed.task_uid == popped.task_uid
True
```

Flush the queue:

```
>>> utility.delete(delayed)
>>> len(utility)
0
```

5.2.15 Task timeout

Create a new task:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
```

When a consumer thread in charge of processing a given task times out, it notifies the queue accordingly so the task is re-queued:

```
>>> running = utility.pop(consumer_id)
>>> running.status
'running'
```

```
>>> utility.timeout(running)
>>> queued = utility.get_task(running.task_uid)
>>> queued.task_uid == running.task_uid
True
```

```
>>> queued.status
'queued'
```

```
>>> queued.get("error_message")
'Timeout'
```

Each time a task is timed out, the number of seconds the system will wait for the thread in charge of processing the task to complete increases. This mechanism is used as a fall-back for when the processing of task takes longer than initially expected:

```
>>> queued.get("max_seconds") > running.get("max_seconds")
True
```

Flush the queue:

```
>>> utility.delete(queued)
>>> len(utility)
0
```

5.2.16 Task failure

Create a new task:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
```

If an error arises when processing a task, the consumer tells the server to mark the task as failed. By default, the queue server re-queues the task up to a pre-defined number of times before considering the task as failed. The most common reason why a task fails is because of a transaction commit conflict with a transaction taken place from userland. This mechanism is used as a safeguard for when the workload is high and tasks keep failing because of this.

Pop a task first:

```
>>> running = utility.pop(consumer_id)
>>> task_uid = running.task_uid
>>> running.status
'running'
```

```
>>> running.retries
3
```

Flag as failed and the number of retries decreases in one unit:

```
>>> utility.fail(running)
>>> failed = utility.get_task(running.task_uid)
>>> failed.task_uid == running.task_uid
True
```

```
>>> failed.retries
2
>>> failed.status
'queued'
```

When the number of retries reach 0, the server eventually considers the task as failed, its status becomes *failed* and cannot be popped anymore:

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'queued'
>>> failed.retries
1
```

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'queued'
>>> failed.retries
0
```

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'failed'
>>> failed.retries
0
```

```
>>> time.sleep(5)
>>> utility.pop(consumer_id) is None
True
```

Flush the queue:

```
>>> utility.delete(failed)
>>> len(utility)
0
```

5.2.17 Task done

When the consumer notifies a task has been done to the server queue, the task is removed from the queue:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
>>> utility.has_task(task)
True
```

```
>>> running = utility.pop(consumer_id)
>>> utility.has_task(running)
True
```

```
>>> utility.done(running)
>>> utility.has_task(running)
False
```

5.2.18 Flush the queue

Flush the queue to make room for other tests:

```
>>> deleted = map(utility.delete, utility.get_tasks())
```

5.3 Client's Queue utility

The `IClientQueueUtility` is an utility that acts as a singleton and is used as an interface to interact with the Server's queue. It provides functions to add tasks to the queue and retrieve them.

This utility is used by the instances that either act as queue clients or consumers. The zeo instance that acts as the queue server uses `IServerQueueUtility` instead. `IClientQueueUtility` has a cache of tasks that keeps up-to-date with those from server's queue through POST calls.

However, both utilities provide same interface, so developer does not need to worry about which utility is actually using: except for some particular cases involving *failed* and *ghost* tasks, their expected behaviour is exactly the same.

Running this test from the buildout directory:

bin/test test_textual_doctests -t ClientQueueUtility

5.3.1 Test Setup

Needed imports:

```
>>> import binascii
>>> import os
>>> import time
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from senaite.queue.interfaces import IQueueUtility
>>> from senaite.queue.interfaces import IClientQueueUtility
>>> from senaite.queue.interfaces import IServerQueueUtility
>>> from senaite.queue.queue import new_task
>>> from senaite.queue.tests import utils as test_utils
>>> from senaite.queue.tests.utils import RequestTestHandler
>>> from zope import globalrequest
>>> from zope.component import getUtility
```

Functional Helpers:

```
>>> def new_sample():
...     return test_utils.create_sample([Cu], client, contact,
...                                     sampletype, receive=False)
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
>>> sample = new_sample()
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
```

5.3.2 Retrieve the client's queue utility

The client queue utility provides all the functionalities required to manage the the queue from the client side. This utility interacts internally with the queue server via JSON API calls, but provides same interface. Therefore, the user should expect the same behavior no matter if is using the client's queue or the server's queue.

```
>>> getUtility(IClientQueueUtility)
<senaite.queue.client.utility.ClientQueueUtility object at...
```

Client utility implements `IQueueUtility` interface too:

```
>>> utility = getUtility(IClientQueueUtility)
>>> IQueueUtility.providedBy(utility)
True
```

The utility makes use of `requests` module to ask the queue server. We override the requests handler here for the doctests to mimic its behavior, but using `plone.testing.z2.Browser` (instead:

```
>>> utility._req = RequestTestHandler(browser, self.request)
```

We we will also use the server's queue utility to validate integrity:

```
>>> s_utility = getUtility(IServerQueueUtility)
```

5.3.3 Add a task

The queue client does not have any task awaiting yet:

```
>>> utility.is_empty()
True
```

Add a task for a sample:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
```

Add the new task:

```
>>> utility.add(task) == task
True
```

```
>>> utility.is_empty()
False
```

```
>>> len(utility)
1
```

The server queue contains the task as well:

```
>>> len(s_utility)
1
```

```
>>> s_utility.has_task(task)
True
```

Only tasks from `QueueTask` type are supported:

```
>>> utility.add("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

Adding an existing task has no effect:

```
>>> utility.add(task) is None
True
```

```
>>> len(utility)
1
```

```
>>> len(s_utility)
1
```

However, we can add another task for same context and with same name:

```
>>> kwargs = {"action": "receive", "test": "test"}
>>> copy_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(copy_task) == copy_task
True
```

```
>>> len(utility)
2
```

```
>>> len(s_utility)
2
```

But is not possible to add a new task for same context and task name when the `unique` wildcard is used:

```
>>> kwargs = {"action": "receive", "unique": True}
>>> unique_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(unique_task) is None
True
```

```
>>> len(utility)
2
```

The server queue contains the two tasks as well:

```
>>> len(s_utility)
2
```

```
>>> all(map(s_utility.has_task, utility.get_tasks()))
True
```

5.3.4 Delete a task

We can delete a task directly:

```
>>> utility.delete(copy_task)
>>> len(utility)
1
```

And the task gets removed from the server's queue as well:

```
>>> len(s_utility)
1
```

We can also delete a task by using the task uid:

```
>>> added = utility.add(copy_task)
>>> len(utility)
2
>>> len(s_utility)
2
```

```
>>> utility.delete(copy_task.task_uid)
>>> len(utility)
1
>>> len(s_utility)
1
```

5.3.5 Get a task

We can retrieve the task we added before by it's uid:

```
>>> retrieved_task = utility.get_task(task.task_uid)
>>> retrieved_task == task
True
```

If we ask for a task that does not exist, returns None:

```
>>> dummy_uid = binascii.hexlify(os.urandom(16))
>>> utility.get_task(dummy_uid) is None
True
```

If we ask for something that is not a valid uid, we get an exception:

```
>>> utility.get_task("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.3.6 Get tasks

Or we can get all the tasks the utility contains:

```
>>> tasks = utility.get_tasks()
>>> tasks
[{...}]
```

```
>>> task in tasks
True
```

```
>>> len(tasks)
1
```

5.3.7 Get tasks by status

We can even get the tasks filtered by their status:

```
>>> utility.get_tasks(status=["queued", "running"])
[{...}]
```

```
>>> utility.get_tasks(status="queued")
[{...}]
```

```
>>> utility.get_tasks(status="running")
[]
```

5.3.8 Get tasks by context

Or we can get the task by context:

```
>>> utility.get_tasks_for(sample)
[{...}]
```

```
>>> utility.get_tasks_for(_api.get_uid(sample))
[{...}]
```

```
>>> utility.get_tasks_for(task.task_uid)
[]
```

```
>>> utility.get_tasks_for("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.3.9 Get tasks by context and task name

```
>>> utility.get_tasks_for(sample, name="task_action_receive")
[{...}]
```

```
>>> utility.get_tasks_for(sample, name="dummy")
[]
```

5.3.10 Get objects uids from tasks

We can also ask for all the uids from objects the utility contains:

```
>>> uids = utility.get_uids()
>>> len(uids)
1
```

```
>>> _api.get_uid(sample) in uids
True
```

```
>>> task.task_uid in uids
False
```

5.3.11 Ask if a task exists

```
>>> utility.has_task(task)
True
```

```
>>> utility.has_task(task.task_uid)
True
```

```
>>> utility.has_task(_api.get_uid(sample))
False
```

```
>>> utility.has_task("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.3.12 Ask if a task for a context exists

```
>>> utility.has_tasks_for(sample)
True
```

```
>>> utility.has_tasks_for(_api.get_uid(sample))
True
```

```
>>> utility.has_tasks_for(task.task_uid)
False
```

```
>>> utility.has_tasks_for("dummy")
Traceback (most recent call last):
[...]
ValueError: 'dummy' is not supported
```

5.3.13 Ask if a task for a context and name exists

```
>>> utility.has_tasks_for(sample, name="task_action_receive")
True
```

```
>>> utility.has_tasks_for(sample, name="dummy")
False
```

5.3.14 Synchronize with queue server

If we add a task directly to the server's queue:

```
>>> kwargs = {"action": "receive"}
>>> server_task = new_task("task_action_receive", sample, **kwargs)
>>> s_utility.add(server_task) == server_task
True
>>> s_utility.has_task(server_task)
True
>>> len(s_utility)
2
```

The task is not in client's queue local pool:

```
>>> server_task in utility.get_tasks()
False
```

However, the client queue falls back to a search against server's queue when asked for an specific task that does not exist in the local pool:

```
>>> utility.has_task(server_task)
True
```

```
>>> utility.get_task(server_task.task_uid)
{...}
```

Client queue's local pool of tasks can be easily synchronized with the tasks from the server's queue:

```
>>> len(utility)
1
```

```
>>> utility.sync()
>>> len(utility)
2
```

```
>>> server_task in utility.get_tasks()
True
```

```
>>> all(map(s_utility.has_task, utility.get_tasks()))
True
```

When the task status in the server is "running", the corresponding task of the local pool is always updated on synchronization:

```
>>> consumer_id = u'http://nohost'
>>> running = s_utility.pop(consumer_id)
>>> running.status
'running'
```

```
>>> local_task = utility.get_task(running.task_uid)
>>> local_task.status
'queued'
```

```
>>> utility.sync()
>>> local_task = utility.get_task(running.task_uid)
>>> local_task.status
'running'
```

Flush the queue:

```
>>> deleted = map(utility.delete, utility.get_tasks())
>>> len(utility)
0
>>> len(s_utility)
0
```

5.3.15 Pop a task

Add a new task to the queue:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(task) == task
True
```

When a task is popped, the utility changes the status of the task to "running", cause expects that the task has been popped for consumption:

```
>>> consumer_id = u'http://nohost'
>>> popped = utility.pop(consumer_id)
>>> popped.status
'running'
```

We can still add new tasks at the same time, even if they are for same context and with same name:

```
>>> kwargs = {"action": "receive"}
>>> copy_task = new_task("task_action_receive", sample, **kwargs)
>>> utility.add(copy_task) == copy_task
True
```

However, is not allowed to consume more more tasks unless the queue server receives an acknowledgment that the previously popped task is done:

```
>>> utility.pop(consumer_id) is None
True
```

Even if we ask again:

```
>>> utility.pop(consumer_id) is None
True
```

Unless we wait for 10 seconds, when the server assumes the consumer failed while processing the task. Consumers always check that there is no thread running from their side before doing a `pop()`. Also, a consumer (that in fact, is a zeo client) might be stopped at some point. Therefore, this timeout mechanism is used as a safety fallback to prevent the queue to enter in a dead-lock:

```
>>> time.sleep(11)
>>> utility.pop(consumer_id) is None
True
```

The previous task is now re-queued:

```
>>> popped = utility.get_task(popped.task_uid)
>>> popped.status
'queued'
```

```
>>> popped.get("error_message")
'Purged on pop (http://nohost)'
```

And a `pop` returns now the next task:

```
>>> next_task = utility.pop(consumer_id)
>>> next_task.status
'running'
```

```
>>> next_task.task_uid != popped.task_uid
True
```

Delete the newly added task, so we keep only one task in the queue for testing:

```
>>> utility.delete(next_task)
>>> len(utility)
1
```

If we try now to `pop` again, the task the queue server considered as timeout won't be popped because the server adds a delay of 5 seconds before the task can be popped again. This mechanism prevents the queue to be jeopardized by recurrent failing tasks and makes room for other tasks to be processed:

```
>>> popped.get("delay")
5
```

```
>>> utility.pop(consumer_id) is None
True
```

```
>>> time.sleep(5)
>>> delayed = utility.pop(consumer_id)
>>> delayed.task_uid == popped.task_uid
True
```

Flush the queue:

```
>>> utility.delete(delayed)
>>> len(utility)
0
```

5.3.16 Task timeout

Create a new task:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
```

When a consumer thread in charge of processing a given task times out, it notifies the queue accordingly so the task is re-queued:

```
>>> running = utility.pop(consumer_id)
>>> running.status
'running'
```

```
>>> utility.timeout(running)
>>> queued = utility.get_task(running.task_uid)
>>> queued.task_uid == running.task_uid
True
```

```
>>> queued.status
'queued'
```

```
>>> queued.get("error_message")
'Timeout'
```

Each time a task is timed out, the number of seconds the system will wait for the thread in charge of processing the task to complete increases. This mechanism is used as a fall-back for when the processing of task takes longer than initially expected:

```
>>> queued.get("max_seconds") > running.get("max_seconds")
True
```

Flush the queue:

```
>>> utility.delete(queued)
>>> len(utility)
0
```

5.3.17 Task failure

Create a new task:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
```

If an error arises when processing a task, the client queue tells the server to mark the task as failed. By default, the queue server re-queues the task up to a pre-defined number of times before considering the task as failed. The most common reason why a task fails is because of a transaction commit conflict with a transaction taken place from userland. This mechanism is used as a safeguard for when the workload is high and tasks keep failing because of this.

Pop a task first:

```
>>> running = utility.pop(consumer_id)
>>> task_uid = running.task_uid
>>> running.status
'running'
```

```
>>> running.retries
3
```

Flag as failed and the number of retries decreases in one unit:

```
>>> utility.fail(running)
>>> failed = utility.get_task(running.task_uid)
>>> failed.task_uid == running.task_uid
True
```

```
>>> failed.retries
2
>>> failed.status
'queued'
```

When the number of retries reach 0, the server eventually considers the task as failed, its status becomes *failed* and cannot be popped anymore:

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'queued'
>>> failed.retries
1
```

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'queued'
>>> failed.retries
0
```

```
>>> time.sleep(5)
>>> failed = utility.pop(consumer_id)
>>> utility.fail(failed)
>>> failed = utility.get_task(failed.task_uid)
>>> failed.status
'failed'
>>> failed.retries
0
```

```
>>> time.sleep(5)
>>> utility.pop(consumer_id) is None
True
```

Flush the queue:

```
>>> utility.delete(failed)
>>> len(utility)
0
```

5.3.18 Task done

When the client notifies a task has been done to the server queue, the task is removed from the queue:

```
>>> kwargs = {"action": "receive"}
>>> task = new_task("task_action_receive", sample, **kwargs)
>>> task = utility.add(task)
>>> utility.has_task(task)
True
```

```
>>> running = utility.pop(consumer_id)
>>> utility.has_task(running)
True
```

```
>>> utility.done(running)
>>> utility.has_task(running)
False
```

5.3.19 Flush the queue

Flush the queue to make room for other tests:

```
>>> test_utils.flush_queue(browser, self.request)
```

5.4 Assignment of analyses

SENAITE Queue supports the *assign* transition for analyses, either for when the analyses are assigned manually (via *Add analyses* view from Worksheet) or when using a Worksheet Template.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesAssign

5.4.1 Test Setup

Needed imports:

```
>>> import time
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_samples(num_analyses):
...     samples = []
...     for num in range(num_analyses):
...         sample = test_utils.create_sample([Cu], client, contact,
...                                           sampletype, receive=True)
...         samples.append(sample)
...     transaction.commit()
...     return samples
```

```
>>> def get_analyses_from(samples):
...     analyses = []
...     for sample in samples:
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     return analyses
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
```

5.4.2 Manual assignment of analyses to a Worksheet

Set the default number of objects to process per task to 5:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Create 15 Samples with 1 analysis each:

```
>>> samples = new_samples(15)
>>> analyses = get_analyses_from(samples)
```

Create an empty worksheet and add all analyses:

```
>>> worksheet = _api.create(portal.worksheets, "Worksheet")
>>> worksheet.addAnalyses(analyses)
>>> transaction.commit()
```

The worksheet is queued now:

```
>>> api.is_queued(worksheet)
True
```

And the analyses as well:

```
>>> queued = map(api.is_queued, analyses)
>>> all(queued)
True
```

None of the analyses have been transitioned:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
0
```

The queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Change the number of items to process per task to 2:

```
>>> plone_api.portal.set_registry_record(chunk_key, 2)
>>> transaction.commit()
```

Pop a task and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Only 2 analyses are transitioned now:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
7
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
8
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

```
>>> api.is_queued(worksheet)
True
```

We can disable the queue. Set the number of items to process per task to 0:

```
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Because the queue contains tasks not yet processed, the queue remains enabled, although is not ready:

```
>>> api.is_queue_enabled()
True
```

```
>>> api.is_queue_ready()
False
```

```
>>> api.get_queue_status()
'resuming'
```

Queue does not allow the addition of new tasks, but remaining tasks are processed as usual but will transition all remaining analyses at once:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

```
>>> queue.is_empty()
True
```

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> api.is_queued(worksheet)
False
```

Since all analyses have been processed, the worksheet is no longer queued and the queue is now disabled:

```
>>> api.is_queued(worksheet)
False
```

```
>>> api.is_queue_enabled()
False
```

```
>>> api.is_queue_ready()
False
```

```
>>> api.get_queue_status()
'disabled'
```

5.4.3 Assignment of analyses through Worksheet Template

Analyses can be assigned to a worksheet by making use of a Worksheet Template. In such case, the system must behave exactly the same way as before.

Set the number of analyses to be transitioned in a single process:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Create 15 Samples with 1 analysis each:

```
>>> samples = new_samples(15)
>>> analyses = get_analyses_from(samples)
```

Create a Worksheet Template with 15 slots reserved for *Cu* analysis:

```
>>> template = _api.create(setup.bika_worksheettemplates, "WorksheetTemplate")
>>> template.setService([Cu])
>>> layout = map(lambda idx: {"pos": idx + 1, "type": "a"}, range(15))
>>> template.setLayout(layout)
>>> transaction.commit()
```

Use the template for Worksheet creation:

```
>>> worksheet = _api.create(portal.worksheets, "Worksheet")
>>> worksheet.applyWorksheetTemplate(template)
>>> transaction.commit()
```

The worksheet is now queued:

```
>>> api.is_queued(worksheet)
True
```

And the analyses as well:

```
>>> queued = map(api.is_queued, analyses)
>>> all(queued)
True
```

None of the analyses have been transitioned:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
0
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Wait for the task delay. This is a mechanism to prevent consumers to start processing while the life-cycle of current request has not been finished yet:

```
>>> task = queue.get_tasks_for(worksheet)[0]
>>> time.sleep(task.get("delay", 0))
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.5 Unassign transition

SENAITE Queue comes with an adapter for generic actions (e.g. submit, unassign). Generic actions don't require additional logic other than transitioning and this is handled by DC workflow. Thus, the adapter for generic actions provided by *senaite.queue* only deal with the number of chunks to process per task, with no additional logic. Most transitions from *senaite.core* match with these requirements.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesUnassign

5.5.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

(continues on next page)

(continued from previous page)

```
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

5.5.2 Unassign transition

Disable the queue first, so *assign* transitions is performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
```

Enable the queue so we can trap the *unassign* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Unassign analyses:

```
>>> test_utils.handle_action(worksheet, analyses, "unassign")
```

The worksheet is queued and the analyses as well:

```
>>> api.is_queued(worksheet)
True
```

```
>>> len(test_utils.filter_by_state(analyses, "unassigned"))
0
```

```
>>> all(map(api.is_queued, analyses))
True
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.5.3 Unassign transition (with ClientQueue)

Perform same test as before, but now using the *ClientQueueUtility*:

```
>>> queue = test_utils.get_client_queue(browser, self.request)
```

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
```

Enable the queue so we can trap the *unassign* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Unassign the analyses:

```
>>> test_utils.handle_action(worksheet, analyses, "unassign")
```

The queue contains one task:

```
>>> queue.sync()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

```
>>> all(filter(queue.get_tasks_for, analyses))
True
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
10
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
5
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "unassigned")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
0
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> queue.is_empty()
True
```

```
>>> queue.has_tasks_for(worksheet)
False
```

5.6 Submit transition

SENAITE Queue comes with an adapter for generic actions (e.g. submit, unassign). Generic actions don't require additional logic other than transitioning and this is handled by DC workflow. Thus, the adapter for generic actions provided by *senaite.queue* only deal with the number of chunks to process per task, with no additional logic. Most transitions from *senaite.core* match with these requirements.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesSubmit

5.6.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

```
>>> def set_analyses_results(worksheet):
...     for analysis in worksheet.getAnalyses():
...         analysis.setResult(13)
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

5.6.2 Submit transition

Disable the queue first, so *assign* transition is performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses and set results:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
```

Enable the queue so we can trap the *submit* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Submit the analyses

```
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

The worksheet is now queued:

```
>>> api.is_queued(worksheet)
True
```

The worksheet is queued and the analyses as well:

```
>>> api.is_queued(worksheet)
True
```

```
>>> len(test_utils.filter_by_state(analyses, "to_be_verified"))
0
```

```
>>> all(map(api.is_queued, analyses))
True
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.6.3 Submit transition (with ClientQueue)

Perform same test as before, but now using the *ClientQueueUtility*:

```
>>> queue = test_utils.get_client_queue(browser, self.request)
```

Disable the queue first, so *assign* transition is performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
```

Enable the queue so we can trap the *submit* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Submit the analyses

```
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

The queue contains one task:

```
>>> queue.sync()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

```
>>> all(filter(queue.get_tasks_for, analyses))
True
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
10
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
5
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> len(non_transitioned)
0
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> queue.is_empty()
True
```

```
>>> queue.has_tasks_for(worksheet)
False
```

5.7 Reject transition

SENAITE Queue comes with an adapter for generic actions (e.g. submit, unassign). Generic actions don't require additional logic other than transitioning and this is handled by DC workflow. Thus, the adapter for generic actions provided by `senaite.queue` only deal with the number of chunks to process per task, with no additional logic. Most transitions from `senaite.core` match with these requirements.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesReject

5.7.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

```
>>> def set_analyses_results(worksheet):
...     for analysis in worksheet.getAnalyses():
...         analysis.setResult(13)
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
```

5.7.2 Reject transition

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *reject* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Reject the results:

```
>>> test_utils.handle_action(worksheet, analyses, "reject")
```

The worksheet is queued and the analyses as well:

```
>>> api.is_queued(worksheet)
True
```

```
>>> len(test_utils.filter_by_state(analyses, "rejected"))
0
```

```
>>> all(map(api.is_queued, analyses))
True
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.7.3 Reject transition (with ClientQueue)

Perform same test as before, but now using the *ClientQueueUtility*:

```
>>> queue = test_utils.get_client_queue(browser, self.request)
```

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *reject* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Retract the results:

```
>>> test_utils.handle_action(worksheet, analyses, "reject")
```

The queue contains one task:

```
>>> queue.sync()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

```
>>> all(filter(queue.get_tasks_for, analyses))
True
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "rejected")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> queue.is_empty()
True
```

```
>>> queue.has_tasks_for(worksheet)
False
```

5.8 Retract transition

SENAITE Queue comes with an adapter for generic actions (e.g. submit, unassign). Generic actions don't require additional logic other than transitioning and this is handled by DC workflow. Thus, the adapter for generic actions provided by *senaite.queue* only deal with the number of chunks to process per task, with no additional logic. Most transitions from *senaite.core* match with these requirements.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesRetract

5.8.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

```
>>> def set_analyses_results(worksheet):
...     for analysis in worksheet.getAnalyses():
...         analysis.setResult(13)
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

5.8.2 Retract transition

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *retract* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Retract the results:

```
>>> test_utils.handle_action(worksheet, analyses, "retract")
```

The worksheet is queued and the analyses as well:

```
>>> api.is_queued(worksheet)
True
```

```
>>> len(test_utils.filter_by_state(analyses, "retracted"))
0
```

```
>>> all(map(api.is_queued, analyses))
True
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.8.3 Retract transition (with ClientQueue)

Perform same test as before, but now using the *ClientQueueUtility*:

```
>>> queue = test_utils.get_client_queue(browser, self.request)
```

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *retract* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Retract the results:

```
>>> test_utils.handle_action(worksheet, analyses, "retract")
```

The queue contains one task:

```
>>> queue.sync()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

```
>>> all(filter(queue.get_tasks_for, analyses))
True
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "retracted")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> queue.is_empty()
True
```

```
>>> queue.has_tasks_for(worksheet)
False
```

5.9 Verify transition

SENAITE Queue comes with an adapter for generic actions (e.g. submit, unassign). Generic actions don't require additional logic other than transitioning and this is handled by DC workflow. Thus, the adapter for generic actions provided by *senaite.queue* only deal with the number of chunks to process per task, with no additional logic. Most transitions from *senaite.core* match with these requirements.

Running this test from the buildout directory:

bin/test test_textual_doctests -t WorksheetAnalysesVerify

5.9.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

```
>>> def set_analyses_results(worksheet):
...     for analysis in worksheet.getAnalyses():
...         analysis.setResult(13)
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Enable the self-verification:

```
>>> setup.setSelfVerificationEnabled(True)
>>> setup.getSelfVerificationEnabled()
True
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

5.9.2 Verify transition

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *verify* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Verify the results:

```
>>> test_utils.handle_action(worksheet, analyses, "verify")
```

The worksheet is queued and the analyses as well:

```
>>> api.is_queued(worksheet)
True
```

```
>>> len(test_utils.filter_by_state(analyses, "verified"))
0
```

```
>>> all(map(api.is_queued, analyses))
True
```

And the queue contains one task:

```
>>> queue = api.get_queue()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

And the worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

As the queue confirms:

```
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> any(map(api.is_queued, transitioned))
False
```

```
>>> all(map(api.is_queued, non_transitioned))
True
```

Since there are still 5 analyses remaining, the Worksheet is still queued:

```
>>> api.is_queued(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> any(map(api.is_queued, transitioned))
False
```

The queue is now empty:

```
>>> queue.is_empty()
True
```

And the worksheet is no longer queued:

```
>>> api.is_queued(worksheet)
False
```

5.9.3 Verify transition (with ClientQueue)

Perform same test as before, but now using the *ClientQueueUtility*:

```
>>> queue = test_utils.get_client_queue(browser, self.request)
```

Disable the queue first, so *submit* and *assign* transitions are performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses, set a result and submit all them:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

Enable the queue so we can trap the *verify* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Verify the results:

```
>>> test_utils.handle_action(worksheet, analyses, "verify")
```

The queue contains one task:

```
>>> queue.sync()
>>> queue.is_empty()
False
```

```
>>> len(queue)
1
```

```
>>> len(queue.get_tasks_for(worksheet))
1
```

```
>>> all(filter(queue.get_tasks_for, analyses))
True
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The first chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
5
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
10
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Next chunk of analyses has been processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
10
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
5
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> all(map(queue.has_tasks_for, non_transitioned))
True
```

```
>>> queue.has_tasks_for(worksheet)
True
```

Pop and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Last chunk of analyses is processed:

```
>>> transitioned = test_utils.filter_by_state(analyses, "verified")
>>> len(transitioned)
15
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> len(non_transitioned)
0
```

```
>>> queue.sync()
>>> any(map(queue.has_tasks_for, transitioned))
False
```

```
>>> queue.is_empty()
True
```

```
>>> queue.has_tasks_for(worksheet)
False
```

5.10 Sample with queued analyses

Samples that contain queued analyses cannot be transitioned until all analyses it contains are successfully processed.

Running this test from buildout directory:

bin/test test_textual_doctests -t SampleWithQueuedAnalyses

5.10.1 Test Setup

Needed imports:

```
>>> import transaction
>>> from bika.lims import api as _api
>>> from bika.lims.workflow import getAllowedTransitions
>>> from plone import api as plone_api
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
>>> from senaite.queue import api
>>> from senaite.queue.tests import utils as test_utils
>>> from zope import globalrequest
```

Functional Helpers:

```
>>> def new_sample(services):
...     return test_utils.create_sample(services, client, contact,
...                                     sampletype, receive=True)
```

```
>>> def new_worksheet(num_analyses):
...     analyses = []
...     for num in range(num_analyses):
...         sample = new_sample([Cu])
...         analyses.extend(sample.getAnalyses(full_objects=True))
...     worksheet = _api.create(portal.worksheets, "Worksheet")
...     worksheet.addAnalyses(analyses)
...     transaction.commit()
...     return worksheet
```

```
>>> def set_analyses_results(worksheet):
...     for analysis in worksheet.getAnalyses():
...         analysis.setResult(13)
...     transaction.commit()
```

```
>>> def samples_transitions_allowed(analyses):
...     samples = map(lambda an: an.getRequest(), analyses)
...     transitions = map(lambda samp: getAllowedTransitions(samp), samples)
...     transitions = map(lambda trans: any(trans), transitions)
...     return all(transitions)
```

Variables:

```
>>> portal = self.portal
>>> request = self.request
>>> setup = _api.get_setup()
>>> browser = self.getBrowser()
>>> globalrequest.setRequest(request)
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Create some basic objects for the test:

```
>>> setRoles(portal, TEST_USER_ID, ['Manager',])
>>> client = _api.create(portal.clients, "Client", Name="Happy Hills", ClientID="HH",␣
↪MemberDiscountApplies=True)
```

(continues on next page)

(continued from previous page)

```
>>> contact = _api.create(client, "Contact", Firstname="Rita", Lastname="Mohale")
>>> sampletype = _api.create(setup.bika_sampletypes, "SampleType", title="Water",␣
↪Prefix="W")
>>> labcontact = _api.create(setup.bika_labcontacts, "LabContact", Firstname="Lab",␣
↪Lastname="Manager")
>>> department = _api.create(setup.bika_departments, "Department", title="Chemistry",␣
↪Manager=labcontact)
>>> category = _api.create(setup.bika_analysiscategories, "AnalysisCategory", title=
↪"Metals", Department=department)
>>> Cu = _api.create(setup.bika_analysisservices, "AnalysisService", title="Copper",␣
↪Keyword="Cu", Price="15", Category=category.UID(), Accredited=True)
```

Setup the current instance as the queue server too:

```
>>> key = "senaite.queue.server"
>>> host = u'http://nohost/plone'
>>> plone_api.portal.set_registry_record(key, host)
>>> transaction.commit()
>>> api.get_queue()
<senaite.queue.server.utility.ServerQueueUtility object at...
```

5.10.2 Queued analyses

Disable the queue first, so *assign* transition is performed non-async:

```
>>> chunk_key = "senaite.queue.default"
>>> plone_api.portal.set_registry_record(chunk_key, 0)
>>> transaction.commit()
```

Create a worksheet with some analyses and set results:

```
>>> worksheet = new_worksheet(15)
>>> analyses = worksheet.getAnalyses()
>>> set_analyses_results(worksheet)
```

Enable the queue so we can trap the *submit* transition:

```
>>> plone_api.portal.set_registry_record(chunk_key, 5)
>>> transaction.commit()
```

Submit the analyses

```
>>> test_utils.handle_action(worksheet, analyses, "submit")
```

No analyses have been transitioned. All them have been queued:

```
>>> test_utils.filter_by_state(analyses, "to_be_verified")
[]
```

Pop a task and process:

```
>>> queue = api.get_queue()
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

Only the first chunk is transitioned and the samples they belong to can be transitioned as well:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> samples_transitions_allowed(transitioned)
True
```

While the rest cannot be transitioned, these analyses are still queued:

```
>>> samples_transitions_allowed(analyses)
False
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> samples_transitions_allowed(non_transitioned)
False
```

Pop a task and process again:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

The next chunk of analyses has been processed and again, only the Samples for those that have been transitioned can be transitioned too:

```
>>> transitioned = test_utils.filter_by_state(analyses, "to_be_verified")
>>> samples_transitions_allowed(transitioned)
True
```

While the rest of Samples (5) cannot be transitioned yet:

```
>>> samples_transitions_allowed(analyses)
False
```

```
>>> non_transitioned = test_utils.filter_by_state(analyses, "assigned")
>>> samples_transitions_allowed(non_transitioned)
False
```

Pop a task and process:

```
>>> popped = queue.pop("http://nohost")
>>> test_utils.process(browser, popped.task_uid)
'{...Processed...}'
```

All analyses have been processed at this point, so all samples can be transitioned now:

```
>>> samples_transitions_allowed(analyses)
True
```

CHAPTER 6

Release notes

6.1 Update from 1.0.1 to 1.0.2

With version 1.0.2, the legacy storage for queued tasks has changed and helper storages (e.g. for Worksheets) are no longer required. `IQueued` marker interface is no longer used neither. Most of the base code has been refactored keeping in mind the following objectives:

- Less complexity: less code, better code
- Less chance of transaction commit conflicts
- Boost performance: better experience, with no delays

All these changes also makes the add-on easier to extend and maintain. The downside is that old legacy storage is no longer used and therefore, tasks that were queued before the upgrade will be discarded.

- Be sure there are no remaining tasks in the queue before the upgrade
- If you have your own add-on extending `senaite.queue`, please review the changes and check if some parts of your add-on require modifications

A queue server has been introduced. Therefore, two zeo clients are recommended: one that acts as the server and at least another one in charge of consuming tasks. Also, this version now depends on three additional packages: `requests`, `senaite.jsonapi` and `cryptography`. Please read the installation instructions and run buildout to download the dependencies.

CHAPTER 7

Changelog

7.1 1.0.4 (unreleased)

- #15 Fix traceback on tasks for the reindex of objects security
- #14 Use intial task's default chunk size when creating subsequent tasks

7.2 1.0.3 (2021-07-24)

- #21 Improve the reindex security objects process
- Skip guard checks when current thread is a consumer
- Make the creation of WS with WST assignment more efficient
- Pin cryptography==3.1.1
- Fix client's queue tasks in "queued" status are not updated when "running"

7.3 1.0.2 (2020-11-15)

- Support for multiple consumers (up to 4 concurrent processes)
- Added JSON API endpoints for both queue server and clients
- Queue server-client implementation, without the need of annotations
- Added PAS plugin for authentication, with symmetric encryption
- Delegate the reindex object security to queue when linking contacts to users
- #7 Allow to queue generic worflow actions without specific adapter
- #7 Redux and better performance

- #6 Allow the prioritization of tasks
- #5 No actions can be done to worksheets with queued jobs

7.4 1.0.1 (2020-02-09)

- Allow to manually assign the username to the task to be queued
- Support for failed tasks
- Notify when the value for max_seconds_unlock is too low
- #3 New *queue_tasks* view with the list of tasks and statistics
- #2 Add max_retries setting for failing tasks
- #1 Add sample guard to prevent transitions when queued analyses

7.5 1.0.0 (2019-11-10)

First version

CHAPTER 8

License

SENAITE.QUEUE Copyright (C) 2019-2020 RIDING BYTES & NARALABS

SENAITE.QUEUE is available under the terms of the GNU General Public License, version 2 as published by the Free Software Foundation.

The source code of this software, together with a copy of the license can be found at this repository: https://github.com/senaite/senaite.queue

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.