ticketutil Documentation

Release 0.0.1

kshirsal

Apr 29, 2022

Contents:

ticketutil is a Python module that allows you to easily interact with various ticketing tools using their REST APIs. Currently, the supported tools are `JIRA`, `RT`, `Redmine`, `Bugzilla` and `ServiceNow`. All tools support HTTP Basic authentication, while JIRA and RT also support Kerberos authentication.

This module allows you to create tickets, add comments, edit ticket fields, and change the status of tickets in each tool. Additional lower-level tool-specific functions are supported - adding and removing watchers in JIRA, adding attachments in JIRA, etc.

Simplify all of your ticketing operations with ticketutil:

```
from ticketutil.jira import JiraTicket
ticket = JiraTicket(<jira_url>, <project_key>, auth='kerberos')

# Create a ticket and perform some common ticketing operations.
t = ticket.create(summary='Ticket summary',
                  description='Ticket description')
t = ticket.add_comment('Test Comment')
t = ticket.change_status('Done')

# Check status of previous ticketing operation and print URL of ticket.
print(t.status)
print(t.url)

# Close Requests session.
ticket.close_requests_session()
```

CHAPTER 1

Installation

- Install ticketutil with:

```
pip install ticketutil
```

- ticketutil is compatible with Python 2.7, 3.3, 3.4, 3.5, and 3.6.

Note: For Python 2.6 and lower, an additional package, importlib, may need to be installed.

- If not installing with pip, a short list of packages defined in the requirements.txt file need to be installed. To install the required packages, type:

```
pip install -r requirements.txt
```

CHAPTER 2

Usage

2.1 Create a Ticket object & authenticate

Create a JiraTicket, RTTicket, RedmineTicket, BugzillaTicket, or ServiceNowTicket object with `<url>`, `<project>` and `<auth>`. This verifies that you are able to properly authenticate to the ticketing tool.

To use HTTP Basic Authentication, the `<auth>` parameter should contain a tuple of the form `auth=(<username>, <password>)`.

For tools that support kerberos authentication (JIRA and RT), the `<auth>` parameter should contain 'kerberos', i.e. `auth='kerberos'`.

To use API key authentication in Bugzilla, the `<auth>` parameter should contain a dictionary of the form `auth={'api_key': <your_api_key>}`.

To use Personal Access Token authentication in Jira, the `<auth>` parameter should contain a dictionary of the form `auth={'token': <your_token>}`.

2.2 Work with a new ticket

- Create a ticket with the `create()` method. This sets the `ticket_id` instance variable, allowing you to perform more tasks on the ticket.
- Add comments, edit ticket fields, add watchers, change the ticket status, etc on the ticket.
- Close ticket Requests session with `close_requests_session()`.

2.3 Work with an existing ticket

There is also a `set_ticket_id()` method for a Ticket object. This is useful if you are working with a Ticket object that already has the `<ticket_id>` instance variable set, but would like to begin working on a separate ticket. Instead

of creating a new Ticket object, you can simply pass an existing <ticket_id> in to the set_ticket_id() method to begin working on another ticket.

- To work on existing tickets, you can also pass in a fourth parameter when creating a Ticket object: <ticket_id>. The general workflow for working with existing tickets is as follows:
- Create a JiraTicket, RTTicket, RedmineTicket, BugzillaTicket or ServiceNowTicket object with <url>, <project_key>, <auth> and <ticket_id>.
- Add comments, edit ticket fields, add watchers, change the ticket status, etc on the ticket.
- Close ticket Requests session with close_requests_session().
- To return the current Ticket object's ticket_id or ticket_url, use the get_ticket_id() or get_ticket_url() methods.

See also:

See the docstrings in the code or the tool-specific sections in the documentation for more information on supported methods and examples.

2.4 Running unit tests

To run unit tests in Bash terminal use this command:

```
python3 -m unittest discover ./tests/
```

2.5 Return Statements

The main user-accessible methods in ticketutil return the status of the method (Success or Failure), the error message if the status is a Failure, and the URL of the ticket. An example is below.

```
# Create Ticket object and create a ticket.
ticket = JiraTicket(. . . .)
t = ticket.create(. . . .)

# View status of create(). Will either return 'Success' or 'Failure'.
print(t.status)

# View error message if status of create() is 'Failure'.
print(t.error_message)

# View URL of ticket.
print(t.url)

# Close Requests session.
ticket.close_requests_session()
```

Note: For JIRA, the remove_all_watchers() method returns a list of the watchers that were removed from the ticket. Access this data with t.watchers.

Note: For all ticketing tools the user-accessible methods return a `ticket_content` field, which contains a json representation of the current ticket's content. Access this data with `t.ticket_content`.

CHAPTER 3

JIRA

This document contains information on the methods available when working with a JiraTicket object. A list of the JIRA fields that have been tested when creating and editing tickets is included. Because each instance of JIRA can have custom fields and custom values, some of the tested fields may not be applicable to certain instances of JIRA. Additionally, your JIRA instance may contain ticket fields that we have not tested. Custom field names and values can be passed in as keyword arguments when creating and editing tickets, and the JIRA REST API should be able to process them. See JIRA's REST API documentation for more information on custom fields: https://docs.atlassian.com/jira/REST/cloud/

Note: A mapping of custom field names to ids for a given Jira instance can be determined through the following API endpoint: /rest/api/2/field. For example: https://<jira_url>/rest/api/2/field. The customfield_XXXXX ids can then be used in the create() or edit() methods below.

3.1 Methods

- *get_ticket_content()*
- *create()*
- *edit()*
- *add_comment()*
- *change_status()*
- *remove_all_watchers()*
- *remove_watcher()*
- *add_watcher()*
- *add_attachment()*

3.1.1 get_ticket_content()

```
get_ticket_content(self, ticket_id=None)
```

Queries the JIRA API to get ticket_content using ticket_id. The ticket_content is expressed in a form of dictionary as a result of JIRA API's get issue: https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/issue-getIssue

```
t = ticket.get_ticket_content(<ticket_id>)
returned_ticket_content = t.ticket_content
```

3.1.2 create()

```
create(self, summary, description, type, **kwargs)
```

Creates a ticket. The required parameters for ticket creation are summary, description and type. Keyword arguments are used for other ticket fields.

```
t = ticket.create(summary='Ticket summary',
                  description='Ticket description',
                  type='Task')
```

The following keyword arguments were tested and accepted by our particular JIRA instance during ticket creation:

```
summary='Ticket summary'
description='Ticket description'
priority='Major'
type='Task'
assignee='username'
reporter='username'
environment='Environment Test'
duedate='2017-01-13'
parent='KEY-XX'
customfield_XXXXX='Custom field text'
components=['component1', 'component2']
```

While creating a Sub task, parent ticket id is required, otherwise create() method fails with KeyError - "Parent field is required while creating a Sub Task". Components are referenced using their names.

3.1.3 edit()

```
edit(self, **kwargs)
```

Edits fields in a JIRA ticket. Keyword arguments are used to specify ticket fields.

```
t = ticket.edit(summary='Ticket summary')
```

The following keyword arguments were tested and accepted by our particular JIRA instance during ticket editing:

```
summary='Ticket summary'
description='Ticket description'
priority='Major'
type='Task'
assignee='username'
reporter='username'
environment='Environment Test'
```

(continues on next page)

(continued from previous page)

```
duedate='2017-01-13'
parent='KEY-XX'
customfield_XXXXX='Custom field text'
```

3.1.4 add_comment()

`add_comment(self, comment)`

Adds a comment to a JIRA ticket.

```
t = ticket.add_comment('Test comment')
```

3.1.5 change_status()

`change_status(self, status, **kwargs)`

Changes status of a JIRA ticket.

Some JIRA instances might not need extra parameters to change the status of the ticket. These instances will use the below code example.

```
t = ticket.change_status('In Progress')
```

With the future state of JIRA, some JIRA instances may not automatically set resolution when changing the status of the ticket to RESOLVED. We need to pass additional fields like 'resolution' or 'comment' to resolve the tickets via automation using TicketUtil itself. See examples below.

Below code example to pass additional parameters like resolution and comment.

```
t = ticket.change_status("Resolved", resolution="Rejected", comment="Resolved via␣
↪automated process.")
```

Below code example to pass only resolution parameter.

```
t = ticket.change_status("Resolved", resolution="Fixed")
```

3.1.6 remove_all_watchers()

`remove_all_watchers(self)`

Removes all watchers from a JIRA ticket.

```
t = ticket.remove_all_watchers()
```

3.1.7 remove_watcher()

`remove_watcher(self, watcher)`

Removes watcher from a JIRA ticket. Accepts an email or username.

```
t = ticket.remove_watcher('username')
```

3.1.8 add_watcher()

`add_watcher(self, watcher)`

Adds watcher to a JIRA ticket. Accepts an email or username.

```
t = ticket.add_watcher('username')
```

3.1.9 add_attachment()

`add_attachment(self, file_name)`

Attaches a file to a JIRA ticket.

```
t = ticket.add_attachment('filename.txt')
```

3.2 Examples

3.2.1 Create JIRATicket object

Authenticate through HTTP Basic Authentication:

```
>>> from ticketutil.jira import JiraTicket
>>> ticket = JiraTicket(<jira_url>,
                        <project_key>,
                        auth=(<username>, <password>))
```

Authenticate through Kerberos after running `kinit`:

```
>>> from ticketutil.jira import JiraTicket
>>> ticket = JiraTicket(<jira_url>,
                        <project_key>,
                        auth='kerberos')
```

Authenticate through a Personal Access Token. See the following URL for details on creating a Personal Access Token in Jira: https://confluence.atlassian.com/enterprise/using-personal-access-tokens-1026032365.html.

```
>>> from ticketutil.jira import JiraTicket
>>> ticket = JiraTicket(<jira_url>,
                        <project_key>,
                        auth={'token': <your_token>})
```

Use proxy to access Jira. See the following URL for details on configuring proxy with requests library: https://docs.python-requests.org/en/latest/user/advanced/#proxies

```
>>> from ticketutil.jira import JiraTicket
>>> ticket = JiraTicket(<jira_url>,
                        <project_key>,
                        auth={'token': <your_token>},
                        proxies={'https': <proxy_url>, 'http': <proxy_url>})
```

You should see the following response:

```
INFO:requests.packages.urllib3.connectionpool:Starting new HTTPS connection (1):
↪<jira_url>
INFO:root:Successfully authenticated to JIRA
```

You now have a `JiraTicket` object that is associated with the `<project_key>` project.

Some example workflows are found below. Notice that the first step is to create a JiraTicket object with a url and project key (and with a ticket id when working with existing tickets), and the last step is closing the Requests session with `t.close_requests_session()`.

When creating a JIRA ticket, `summary` and `description` are required parameters. Also, the Reporter is automatically filled in as the current kerberos principal.

Note: The tested parameters for the create() and edit() methods are found in the docstrings in the code and in the docs folder. Any other ticket field can be passed in as a keyword argument, but be aware that the value for non-tested fields or custom fields may be in a non-intuitive format. See JIRA's REST API documentation for more information: https://docs.atlassian.com/jira/REST/cloud/

3.2.2 Create and update JIRA ticket

```
from ticketutil.jira import JiraTicket

# Create a ticket object and pass the url and project key in as strings.
ticket = JiraTicket(<jira_url>,
                    <project_key>,
                    auth='kerberos')

# Create a ticket and perform some common ticketing operations.
t = ticket.create(summary='Ticket summary',
                  description='Ticket description',
                  type='Task',
                  priority='Major',
                  assignee='username')
t = ticket.get_ticket_content('Ticket_ID')
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='Critical',
                type='Bug')
t = ticket.remove_all_watchers()
t = ticket.add_watcher('username')
t = ticket.add_attachment('file_to_attach.txt')
t = ticket.change_status('In Progress')

# Close Requests session.
ticket.close_requests_session()
```

3.2.3 Update existing JIRA tickets

```
from ticketutil.jira import JiraTicket

# Create a ticket object and pass the url, project key, and ticket id in as strings.
ticket = JiraTicket(<jira_url>,
                    <project_key>,
                    auth='kerberos',
                    ticket_id=<ticket_id>)
```

(continues on next page)

(continued from previous page)

```
# Perform some common ticketing operations.
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='Critical',
                type='Bug')

# Check the actual ticket content after applied updates
t = ticket.get_ticket_content()
returned_ticket_content = t.ticket_content

# Work with a different ticket.
t = ticket.set_ticket_id(<new_ticket_id>)
t = ticket.remove_watcher('username')
t = ticket.add_watcher('username')
t = ticket.change_status('Done')

# Close Requests session.
ticket.close_requests_session()
```

3.2.4 Create a Sub-Task inside existing JIRA ticket

```
from ticketutil.jira import JiraTicket

# Create a ticket object and pass the url and project key in as strings.
t = JiraTicket(<jira_url>,
               <project_key>,
               auth=('username', 'password'))

# Create a ticket and perform some common ticketing operations.
t.create(summary='Sub Task summary',
         description='Sub Task description',
         assignee='username',
         type='Sub-task',
         parent='existing_ticket_id')
t.change_status('In Progress')

# Close Requests session.
t.close_requests_session()
```

CHAPTER 4

RT

This document contains information on the methods available when working with a RTTicket object. A list of the RT fields that have been tested when creating and editing tickets is included. Because each instance of RT can have custom fields and custom values, some of the tested fields may not be applicable to certain instances of RT. Additionally, your RT instance may contain ticket fields that we have not tested. Custom field names and values can be passed in as keyword arguments when creating and editing tickets, and the RT REST API should be able to process them. See RT's REST API documentation for more information on custom fields: https://rt-wiki.bestpractical.com/wiki/REST

4.1 Methods

- *get_ticket_content()*
- *create()*
- *edit()*
- *add_comment()*
- *change_status()*
- *add_attachment()*

4.1.1 get_ticket_content()

`get_ticket_content(self, ticket_id=None, option='show')`

Queries the RT API to get the ticket_content using ticket_id. Calls have different options ('show', 'comment', 'attachments', 'history') in dependence of what kind of content is required. The ticket_content is expressed as a dictionary for options 'show', 'attachments' and 'history' and as a list of strings representing lines in returned text for option 'comment'. The API calling is described in https://rt-wiki.bestpractical.com/wiki/REST#Ticket

```
t = ticket.get_ticket_content(<ticket_id>, option='attachments')
returned_ticket_content = t.ticket_content
```

4.1.2 create()

`create(self, subject, text, **kwargs)`

Creates a ticket. The required parameters for ticket creation are subject and text. Keyword arguments are used for other ticket fields.

```
t = ticket.create(subject='Ticket subject',
                  text='Ticket text')
```

The following keyword arguments were tested and accepted by our particular RT instance during ticket creation:

```
subject='Ticket subject'
text='Ticket text'
priority='5'
owner='username@mail.com'
cc='username@mail.com'
admincc=['username@mail.com', 'username2@mail.com']
```

NOTE: cc and admincc accept a string representing one user's email address, or a list of strings for multiple users.

4.1.3 edit()

`edit(self, **kwargs)`

Edits fields in a RT ticket. Keyword arguments are used to specify ticket fields.

```
t = ticket.edit(owner='username@mail.com')
```

The following keyword arguments were tested and accepted by our particular RT instance during ticket editing:

```
priority='5'
owner='username@mail.com'
cc='username@mail.com'
admincc=['username@mail.com', 'username2@mail.com']
```

NOTE: cc and admincc accept a string representing one user's email address, or a list of strings for multiple users.

4.1.4 add_comment()

`add_comment(self, comment)`

Adds a comment to a RT ticket.

```
t = ticket.add_comment('Test comment')
```

4.1.5 change_status

`change_status(self, status)`

Changes status of a RT ticket.

```
t = ticket.change_status('Resolved')
```

4.1.6 add_attachment()

`add_attachment(self, file_name)`

Attaches a file to a RT ticket.

```
t = ticket.add_attachment('filename.txt')
```

4.2 Examples

4.2.1 Create RTTicket object

Authenticate through HTTP Basic Authentication:

```
>>> from ticketutil.rt import RTTicket
>>> ticket = RTTicket(<rt_url>,
                      <project_queue>,
                      auth=('username', 'password'))
```

Authenticate through Kerberos after running `kinit`:

```
>>> from ticketutil.rt import RTTicket
>>> ticket = RTTicket(<rt_url>,
                      <project_queue>,
                      auth='kerberos')
```

You should see the following response:

```
INFO:requests.packages.urllib3.connectionpool:Starting new HTTPS connection (1): <rt_
↪url>
INFO:root:Successfully authenticated to RT
```

You now have a `RTTicket` object that is associated with the `<project_queue>` queue.

Some example workflows are found below. Notice that the first step is to create a RTTicket object with a url and project queue (and with a ticket id when working with existing tickets), and the last step is closing the Requests session with `t.close_requests_session()`.

When creating a RT ticket, `subject` and `text` are required parameters. Also, the Reporter is automatically filled in as the current kerberos principal.

Note: The tested parameters for the create() and edit() methods are found in the docstrings in the code and in the docs folder. Any other ticket field can be passed in as a keyword argument, but be aware that the value for non-tested fields or custom fields may be in a non-intuitive format. See RT's REST API documentation for more information: https://rt-wiki.bestpractical.com/wiki/REST

4.2.2 Create and update RT ticket

```
from ticketutil.rt import RTTicket

# Create a ticket object and pass the url and project queue in as strings.
ticket = RTTicket(<rt_url>,
                  <project_queue>,
                  auth='kerberos')
```

(continues on next page)

(continued from previous page)

```
# Create a ticket and perform some common ticketing operations.
t = ticket.create(subject='Ticket subject',
                  text='Ticket text',
                  priority='5',
                  owner='username@mail.com',
                  cc='username@mail.com,
                  admincc=['username@mail.com', 'username2@mail.com'])
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='4',
                cc='username1@mail.com')
t = ticket.add_attachment('file_to_attach.txt')
t = ticket.change_status('Resolved')

# Close Requests session.
t = ticket.close_requests_session()
```

4.2.3 Update existing RT tickets

```
from ticketutil.rt import RTTicket

# Create a ticket object and pass the url, project queue, and ticket id in as strings.
ticket = RTTicket(<rt_url>,
                  <project_queue>,
                  auth='kerberos',
                  ticket_id=<ticket_id>)

# Perform some common ticketing operations.
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='4',
                cc='username@mail.com')

# Check the ticket content.
t = ticket.get_ticket_id()
returned_ticket_content = t.ticket_content

# Work with a different ticket.
t = ticket.set_ticket_id(<new_ticket_id>)
t = ticket.change_status('Resolved')

# Close Requests session.
ticket.close_requests_session()
```

CHAPTER 5

Redmine

This document contains information on the methods available when working with a RedmineTicket object. A list of the Redmine fields that have been tested when creating and editing tickets is included. Because each instance of Redmine can have custom fields and custom values, some of the tested fields may not be applicable to certain instances of Redmine. Additionally, your Redmine instance may contain ticket fields that we have not tested. Custom field names and values can be passed in as keyword arguments when creating and editing tickets, and the Redmine REST API should be able to process them. See Redmine's REST API documentation for more information on custom fields: http://www.redmine.org/projects/redmine/wiki/Rest_api

Note: Redmine's REST API requires that you refer to many fields using their 'id' values instead of their 'name's, including Project, Status, Priority, and User. For these four fields, we have `_get_<field>_id()` methods, so you can use the name instead of having to look up the id.

5.1 Methods

5.1.1 get_ticket_content()

```
get_ticket_content(self, ticket_id=None)
```

Queries the Redmine API to get ticket_content using ticket_id. The ticket_content is expressed in a form of dictionary as a result of Redmine API's get issue: http://www.redmine.org/projects/redmine/wiki/Rest_Issues#Showing-an-issue

```
t = ticket.get_ticket_content(<ticket_id>)
returned_ticket_content = t.ticket_content
```

5.1.2 create()

`create(self, subject, description, **kwargs)`

Creates a ticket. The required parameters for ticket creation are subject and description. Keyword arguments are used for other ticket fields.

```
t = ticket.create(subject='Ticket subject',
                  description='Ticket description')
```

The following keyword arguments were tested and accepted by our particular Redmine instance during ticket creation:

```
subject='Ticket subject'
description='Ticket description'
priority='Urgent'
start_date='2017-01-20'
due_date='2017-01-25'
done_ratio='70'
assignee='username@mail.com'
```

5.1.3 edit()

`edit(self, **kwargs)`

Edits fields in a Redmine ticket. Keyword arguments are used to specify ticket fields.

```
t = ticket.edit(subject='Ticket subject')
```

The following keyword arguments were tested and accepted by our particular Redmine instance during ticket editing:

```
subject='Ticket subject'
description='Ticket description'
priority='Urgent'
start_date='2017-01-20'
due_date='2017-01-25'
done_ratio='70'
assignee='username@mail.com'
```

5.1.4 add_comment()

`add_comment(self, comment)`

Adds a comment to a Redmine ticket.

```
t = ticket.add_comment('Test comment')
```

5.1.5 change_status()

```
change_status(self, status)
```

Changes status of a Redmine ticket.

```
t = ticket.change_status('Resolved')
```

5.1.6 remove_watcher()

```
remove_watcher(self, watcher)
```

Removes watcher from a Redmine ticket. Accepts an email or username.

```
t = ticket.remove_watcher('username')
```

5.1.7 add_watcher()

```
add_watcher(self, watcher)
```

Adds watcher to a Redmine ticket. Accepts an email or username.

```
t = ticket.add_watcher('username')
```

5.1.8 add_attachment()

```
add_attachment(self, file_name)
```

Attaches a file to a Redmine ticket.

```
t = ticket.add_attachment('filename.txt')
```

5.2 Examples

5.2.1 Create RedmineTicket object

Currently, ticketutil supports HTTP Basic authentication for Redmine. When creating a RedmineTicket object, pass in your username and password as a tuple into the auth argument. You can also use an API key passed in as a username with a random password for <password>. For more details, see http://www.redmine.org/projects/redmine/wiki/Rest_api#Authentication.

```
>>> from ticketutil.redmine import RedmineTicket
>>> ticket = RedmineTicket(<redmine_url>,
                           <project_name>,
                           auth=(<username>, <password>))
```

You should see the following response:

```
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):
↪<redmine_url>
INFO:root:Successfully authenticated to Redmine
```

You now have a `RedmineTicket` object that is associated with the `<project_name>` project.

Some example workflows are found below. Notice that the first step is to create a RedmineTicket object with a url and project key (and with a ticket id when working with existing tickets), and the last step is closing the Requests session with `t.close_requests_session()`.

When creating a Redmine ticket, `subject` and `description` are required parameters. Also, the Reporter is automatically filled in as the current username.

Note: The tested parameters for the create() and edit() methods are found in the docstrings in the code and in the docs folder. Any other ticket field can be passed in as a keyword argument, but be aware that the value for non-tested fields or custom fields may be in a non-intuitive format. See Redmine's REST API documentation for more information: http://www.redmine.org/projects/redmine/wiki/Rest_api

5.2.2 Create and update Redmine ticket

```
from ticketutil.redmine import RedmineTicket

# Create a ticket object and pass the url and project name in as strings.
ticket = RedmineTicket(<redmine_url>,
                       <project_name>,
                       auth=(<username>, <password>))

# Create a ticket and perform some common ticketing operations.
t = ticket.create(subject='Ticket subject',
                  description='Ticket description',
                  priority='Urgent',
                  start_date='2017-01-20',
                  due_date='2017-01-25',
                  done_ratio='70',
                  assignee='username@mail.com')
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='Normal',
                due_date='2017-02-25')
t = ticket.add_attachment('file_to_attach.txt')
t = ticket.add_watcher('username1')
t = ticket.remove_watcher('username2')
t = ticket.change_status('Closed')

# Close Requests session.
ticket.close_requests_session()
```

5.2.3 Update existing Redmine tickets

```
from ticketutil.redmine import RedmineTicket

# Create a ticket object and pass the url, project name, and ticket id in as strings.
ticket = RedmineTicket(<redmine_url>,
                       <project_name>,
                       auth=(<username>, <password>),
                       ticket_id=<ticket_id>)

# Perform some common ticketing operations.
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='High',
```

(continues on next page)

(continued from previous page)

```
                done_ratio='90')

# Check the ticket content.
t = ticket.get_ticket_id()
returned_ticket_content = t.ticket_content

# Work with a different ticket.
t = ticket.set_ticket_id(<new_ticket_id>)
t = ticket.change_status('Resolved')

# Close Requests session.
ticket.close_requests_session()
```

CHAPTER 6

Bugzilla

This document contains information on the methods available when working with a BugzillaTicket object. A list of the Bugzilla fields that have been tested when creating and editing tickets is included. Because each instance of Bugzilla can have custom fields and custom values, some of the tested fields may not be applicable to certain instances of Bugzilla. Additionally, your Bugzilla instance may contain ticket fields that we have not tested. Custom field names and values can be passed in as keyword arguments when creating and editing tickets, and the Bugzilla REST API should be able to process them. See Bugzilla's REST API documentation for more information on custom fields: http://bugzilla.readthedocs.io/en/latest/api/index.html

6.1 Methods

- *get_ticket_content()*
- *create()*
- *edit()*
- *add_comment()*
- *change_status()*
- *remove_cc()*
- *add_cc()*
- *add_attachment()*

6.1.1 get_ticket_content()

`get_ticket_content(self, ticket_id=None)`

Queries the Bugzilla API to get ticket_content using ticket_id. The ticket_content is expressed in a form of dictionary as a result of Bugzilla API's get issue: http://bugzilla.readthedocs.io/en/latest/api/core/v1/bug.html#get-bug

```
t = ticket.create(ticket_id=<ticket_id>)
returned_ticket_content = t.ticket_content
```

6.1.2 create()

`create(self, summary, description, **kwargs)`

Creates a ticket. The required parameters for ticket creation are summary and description. Keyword arguments are used for other ticket fields.

```
t = ticket.create(summary='Ticket summary',
                  description='Ticket description')
```

The following keyword arguments were tested and accepted by our particular Bugzilla instance during ticket creation:

```
summary='Ticket summary'
description='Ticket description'
assignee='username@mail.com'
qa_contact='username@mail.com'
component='Test component'
version='version'
priority='high'
severity='medium'
alias='SomeAlias'
groups='GroupName'
```

6.1.3 edit()

`edit(self, **kwargs)`

Edits fields in a Bugzilla ticket. Keyword arguments are used to specify ticket fields.

```
t = ticket.edit(summary='Ticket summary')
```

The following keyword arguments were tested and accepted by our particular Bugzilla instance during ticket editing:

```
summary='Ticket summary'
assignee='username@mail.com'
qa_contact='username@mail.com'
component='Test component'
version='version'
priority='high'
severity='medium'
alias='SomeAlias'
groups='Group Name'
```

6.1.4 add_comment()

`add_comment(self, comment, **kwargs )`

Adds a comment to a Bugzilla ticket. Keyword arguments are used to specify comment options.

```
t = ticket.add_comment('Test comment')
```

6.1.5 change_status()

`change_status(self, status, **kwargs)`

Changes status of a Bugzilla ticket. Some status changes require a secondary field (i.e. resolution). Specify this as a keyword argument. A resolution of Duplicate requires dupe_of keyword argument with a valid bug ID.

```
t = ticket.change_status('NEW')
t = ticket.change_status('CLOSED', resolution='DUPLICATE', dupe_of='<bug_id>')
```

6.1.6 remove_cc()

`remove_cc(self, user)`

Removes user(s) from CC List of a Bugzilla ticket. Accepts a string representing one user's email address, or a list of strings for multiple users.

```
t = ticket.remove_cc('username@mail.com')
```

6.1.7 add_cc()

`add_cc(self, user)`

Adds user(s) to CC List of a Bugzilla ticket. Accepts a string representing one user's email address, or a list of strings for multiple users.

```
t = ticket.add_cc(['username1@mail.com', 'username2@mail.com'])
```

6.1.8 add_attachment()

`add_attachment(self, file_name, data, summary, **kwargs )`

Add attachment in a Bugzilla ticket. Keyword arguments are used to specify additional attachment options.

```
t = ticket.add_attachment(file_name='Name to be displayed on UI',
                          data='Location(path) or contents of the attachment',
                          summary='A short string describing the attachment.')
```

6.2 Examples

6.2.1 Create BugzillaTicket object

Currently, ticketutil supports `HTTP Basic authentication` and `API key authentication` for Bugzilla.

While creating a bugzilla ticket you can pass in your username and password as a tuple into the auth argument. The code then authenticates for subsequent API calls. For more details, see: http://bugzilla.readthedocs.io/en/latest/api/index.html.

```
>>> from ticketutil.bugzilla import BugzillaTicket
>>> ticket = BugzillaTicket(<bugzilla_url>,
                            <product_name>,
                            auth=(<username>, <password>))
```

OR, you can use API key authentication. Before you use API key authentication, you need to generate the API key for your account by clicking on the API Keys section under your user preferences in Bugzilla. When creating a BugzillaTicket object, you can pass in a dictionary of the form {'api_key': <your_api_key>} into the auth argument. The code then authenticates for subsequent API calls. For more details, see: http://bugzilla.readthedocs.io/en/latest/api/core/v1/general.html#authentication.

```
>>> from ticketutil.bugzilla import BugzillaTicket
>>> ticket = BugzillaTicket(<bugzilla_url>,
                            <product_name>,
                            auth={'api_key': <your_api_key>})
```

You now have a `BugzillaTicket` object that is associated with the `<product_name>` product.

Some example workflows are found below. Notice that the first step is to create a BugzillaTicket object with a url and product name (and with a ticket id when working with existing tickets), and the last step is closing the Requests session with `t.close_requests_session()`.

When creating a Bugzilla ticket, `summary` and `description` are required parameters. Also, the Reporter is automatically filled in as the current kerberos principal or username supplied during authentication.

Note: The tested parameters for the create() and edit() methods are found in the docstrings in the code and in the docs folder. Any other ticket field can be passed in as a keyword argument, but be aware that the value for non-tested fields or custom fields may be in a non-intuitive format. See Bugzilla's REST API documentation for more information: http://bugzilla.readthedocs.io/en/latest/api/index.html

6.2.2 Create and update Bugzilla ticket

```
from ticketutil.bugzilla import BugzillaTicket

# Create a ticket object and pass the url and product name in as strings.
ticket = BugzillaTicket(<bugzilla_url>,
                        <product_name>,
                        auth=(<username>, <password>))

# Create a ticket and perform some common ticketing operations.
t = ticket.create(summary='Ticket summary',
                  description='Ticket description',
                  component='Test component',
                  priority='high',
                  severity='medium',
                  assignee='username@mail.com',
                  qa_contact='username@mail.com',
                  groups='beta')
t = ticket.get_ticket_id()
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='medium',
                qa_contact='username@mail.com')
t = ticket.add_cc(['username1@mail.com', 'username2@mail.com'])
t = ticket.remove_cc('username1@mail.com')
t = ticket.change_status('Modified')

# Close Requests session.
ticket.close_requests_session()
```

6.2.3 Update existing Bugzilla tickets

```
from ticketutil.bugzilla import BugzillaTicket

# Create a ticket object and pass the url, product name, and ticket id in as strings.
ticket = BugzillaTicket(<bugzilla_url>,
                        <product_name>,
                        auth=(<username>, <password>)
                        ticket_id=<ticket_id>)

# Perform some common ticketing operations.
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='low',
                severity='low',
                groups='beta')

t = ticket.add_attchment(file_name='test_attachment.patch',
                         data=<contents/file-location>,
                         summary=<summary describing attachment>)

# Check the ticket content.
t = ticket.get_ticket_id()
returned_ticket_content = t.ticket_content

# Work with a different ticket.
t = ticket.set_ticket_id(<new_ticket_id>)
t = ticket.change_status(status='CLOSED', resolution='NOTABUG')

# Close Requests session.
ticket.close_requests_session()
```

CHAPTER 7

ServiceNow

This document contains information on the methods available when working with a ServiceNow object. A list of the ServiceNow fields that have been tested when creating and editing tickets is included.

Note: Because each instance of ServiceNow can have custom fields and custom values, some of the tested fields may not be applicable to certain instances of ServiceNow. Additionally, your ServiceNow instance may contain ticket fields that we have not tested. Keep in mind different ServiceNow tables contain different fields with different properties. Mandatory fields in one table may be optional or non-existent in another table.

Custom field names and values can be passed in as keyword arguments when creating and editing tickets. These are processed by the ServiceNow REST API right afterwards. You can also use *get_ticket_content* to get ticket content including preset field names.

For more information about REST API and fields see ServiceNow's Wiki:

- REST API
- Introduction to Fields
- Dot-Walking

7.1 Methods

- *set_ticket_id()*
- *create()*
- *get_ticket_content()*
- *edit()*
- *add_comment()*
- *change_status()*
- *add_cc()*
- *rewrite_cc()*

- *remove_cc()*
- *add_attachment()*

7.1.1 set_ticket_id()

`set_ticket_id(self, ticket_id)`

Updates ticket content for the current ticket object. This ticket object is assigned using `ticket_id` which is a string.

```
# switch to ticket 'ID0123456'
t = ticket.set_ticket_id('ID0123456')
```

7.1.2 create()

`create(self, short_description, description, category, item, **kwargs)`

Creates a ticket. The required parameters for ticket creation are short description and description of the issue. Keyword arguments are used for other ticket fields.

```
t = ticket.create(short_description='Ticket summary',
                  description='Ticket description',
                  category='Category',
                  item='ServiceNow')
```

The following keyword arguments were tested and accepted by our particular ServiceNow instance during ticket creation:

```
category = 'Category'
item = 'ServiceNow'
contact_type = 'Email'
opened_for = 'dept'
assigned_to = 'username'
impact = '2'
urgency = '2'
priority = '2'
email_from = 'username@domain.com'
hostname_affected = '127.0.0.1'
state = 'Work in Progress'
watch_list = ['username@domain.com', 'user2@domain.com', 'user3@domain.com']
comments = 'New comment to be added'
```

7.1.3 get_ticket_content()

`get_ticket_content(self, ticket_id)`

Retrieves ticket content as a dictionary. Optional parameter ticket_id specifies which ticket should be retrieved this way. If not used, method calls for ticket_id provided by ServiceNowTicket constructor (or create method).

```
# ticket content of the object t
t = ticket.get_ticket_content()
# ticket content of the ticket <ticket_id>
t = ticket.get_ticket_content(ticket_id=<ticket_id>)
```

7.1.4 edit()

`edit(self, **kwargs)`

Edits fields in a ServiceNow ticket. Keyword arguments are used to specify ticket fields. Most of the fields overwrite existing fields. One known exception to that rule is 'comments' which adds new comment when specified.

```
t = ticket.edit(short_description='Ticket summary')
```

The following keyword arguments were tested and accepted by our particular ServiceNow instance during ticket editing:

```
category = 'Category'
item = 'ServiceNow'
contact_type = 'Email'
opened_for = 'dept'
assigned_to = 'username'
impact = '2'
urgency = '2'
priority = '2'
email_from = 'username@domain.com'
hostname_affected = '127.0.0.1'
state = 'Work in Progress'
watch_list = ['username@domain.com', 'user2@domain.com', 'user3@domain.com']
comments = 'New comment to be added'
```

7.1.5 add_comment()

`add_comment(self, comment)`

Adds a comment to a ServiceNow ticket. Note that comments cannot be modified or deleted in the current implementation.

```
t = ticket.add_comment('Test comment')
```

7.1.6 change_status(self, status)

Changes status of a ServiceNow ticket.

```
t = ticket.change_status('Work in Progress')
```

7.1.7 add_cc()

`add_cc(self, user)`

Adds watcher(s) to a ServiceNow ticket. Accepts email addresses in the form of list of strings or one string representing one email address.

```
t = ticket.add_cc('username@domain.com')
```

7.1.8 rewrite_cc()

`rewrite_cc(self, user)`

Rewrites current watcher list in the ServiceNow ticket. Accepts email addresses in the form of list of strings or one string representing one email address.

```
t = ticket.rewrite_cc(['username@domain.com', 'user2@domain.com', 'user3@domain.com'])
```

7.1.9 remove_cc()

`remove_cc(self, user)`

Removes users from the current watcher list in the ServiceNow ticket. Accepts email addresses in the form of list of strings or one string representing one email address.

```
t = ticket.remove_cc(['username@domain.com', 'user3@domain.com'])
```

7.1.10 add_attachment()

`add_attachment(self, file_name, name=None)`

Attaches the file to existing ServiceNow ticket. Required parameter is file_name. If name is specified, it is used to rename the saved file.

```
t = ticket.add_attachment('scan01234.jpg', 'scan.jpg')
```

7.2 Examples

7.2.1 Create ServiceNowTicket object

Currently, ticketutil supports HTTP Basic Authentication for ServiceNow. When creating a ServiceNowTicket object, pass in your username and password as a tuple into the auth argument. The code then retrieves a token that will be used as authentication for subsequent API calls. For more details see documentation.

```
>>> from ticketutil.servicenow import ServiceNowTicket
>>> ticket = ServiceNowTicket(<servicenow_url>,
                              <table_name>,
                              auth=(<username>, <password>))
```

You should see the following response:

```
INFO:requests.packages.urllib3.connectionpool:Starting new HTTPS connection (1):
↪<servicenow_url>
INFO:root:Successfully authenticated to ServiceNow
```

You now have a `ServiceNowTicket` object that is associated with the `<table_name>` table.

Some example workflows are found below. Notice that the first step is to create a ServiceNowTicket object with an url table name (and with a ticket id when working with existing tickets), and the last step is closing the Requests session with `t.close_requests_session()`.

When creating a ServiceNow ticket, `short_description`, `description`, `category` and `item` are required parameters. Also, the Reporter is automatically filled in as the current kerberos principal or username supplied during authentication.

7.2.2 Create new ServiceNow ticket

```
from ticketutil.servicenow import ServiceNowTicket

# Create a ticket object and pass the url and table name in as strings
ticket = ServiceNowTicket(<servicenow_url>,
                          <table_name>,
                          auth=(<username>, <password>))

# Create a ticket and perform some common ticketing operations
t = ticket.create(short_description='TEST adding SNow API into ticketutil',
                  description='Ticket description',
                  category='Communication',
                  item='ServiceNow')
t = ticket.edit(assigned_to='pzubaty',
                priority='3')
t = ticket.add_cc(['username1@mail.com', 'username2@mail.com'])
t = ticket.remove_cc('username1@mail.com')
t = ticket.change_status('Work in Progress')
t = ticket.add_attachment('scan01234.jpg', 'scan.jpg')

# Retrieve ticket content
t = ticket.get_ticket_content()

# Close Requests session
ticket.close_requests_session()
```

7.2.3 Update existing ServiceNow tickets

```
from ticketutil.servicenow import ServiceNowTicket

ticket = ServiceNowTicket(<servicenow_url>,
                          <table_name>,
                          auth=(<username>, <password>),
                          ticket_id=<ticket_id>)
t = ticket.add_comment('Test Comment')
t = ticket.edit(priority='4',
                impact='4')

# Work with a different ticket
t = ticket.set_ticket_id(<new_ticket_id>)
t = ticket.change_status('Pending')

# Close Requests session
ticket.close_requests_session()
```

CHAPTER 8

Comments? / Questions? / Coming Soon

For questions / comments, email dranck@redhat.com. For anything specific to Bugzilla, email kshirsal@redhat.com. For ServiceNow related questions, email pzubaty@redhat.com.

The plan for ticketutil is to support more ticketing tools in the near future and to support more ticketing operations for the currently supported tools. Please let us know if there are any suggestions / requests. Thanks!