pyral Documentation

Release 1.5.1

Broadcom

Jul 21, 2021

Contents

The official name for this software is **Python toolkit for the Rally REST API**.

The actual name of the package installed is **pyral**.

Contents:

CHAPTER 1

Python toolkit for the Rally REST API

Rally supports a REST API that enables you to retrieve representations of entities in Rally, create entities in Rally, update existing entities in Rally and with proper permissions, delete entities in Rally.

Once you have the **pyral** package installed, all you need is a valid subscription to Rally, working credentials and the name of the workspace and project you want to interact with and you're on your way!

For more information on obtaining a Rally subscription, visit the Rally website.

For more information on how workspaces and projects in Rally are set up and configured, consult the Rally documentation available via the 'Help' link from the Rally landing page displayed after your initial login.

CHAPTER 2

Simple Use

Here's a prototype of simple use of the **pyral** package.:

```
import sys

from pyral import Rally, rallyWorkset

options = [opt for opt in sys.argv[1:] if opt.startswith('--')]
server, user, password, apikey, workspace, project = rallyWorkset(options)
rally = Rally(server, user, password, workspace=workspace, project=project)
rally.enableLogging('rally.simple-use.log')

response = rally.get('Release', fetch="Project,Name,ReleaseStartDate,ReleaseDate",
                     order="ReleaseDate")

for rls in response:
    rlsStart = rls.ReleaseStartDate.split('T')[0]  # just need the date part
    rlsDate  = rls.ReleaseDate.split('T')[0]       # ditto
    print("%-6.6s  %-16.16s   %s  -->  %s" % \
          (rls.Project.Name, rls.Name, rlsStart, rlsDate))
```

CHAPTER 3

Rally Data Model

All Rally entities belong to a hierarchical data model and every Rally entity ultimately is a descendent of the PersistableObject class. There are several branches in the data model, and each branch has its own set of attributes differentiated according to the functional capabilities and information tracking needs that characterize the branch. For more information on the Rally data model, consult the Rally documentation available via the 'Help' link from the Rally page displayed after the initial login.

CHAPTER 4

Rally Entities and Artifacts

In the Rally vernacular, a logical entity is called a *type*. Some examples of Rally *types* are UserStory, Defect, Release, UserProfile. There is a subset of *types* that are usually what a user of **pyral** will be interested in called *artifacts*. An *artifact* is either a UserStory, Defect, Task, DefectSuite, TestCase or TestSet. The Python toolkit for the Rally REST API (**pyral**) is primarily oriented towards operations with artifacts. But, it is not limited to those as it is very possible to view/operate on other Rally entities such as Workspace, Project, UserProfile, Release, Iteration, TestCaseResult, TestFolder, Tag and others.

CHAPTER 5

Full CRUD capability

The Python toolkit for the Rally REST API offers the full spectrum of CRUD capabilities that the credentials supplied for your subscription/workspace/project permit. Rally REST API did not originally support bulk operations when this toolkit was written. Since the 2017/2018 timeframe the Rally REST WSAPI has provided some bulk operations, but this toolkit doesn't use those or provide access to them. There are example usages of **pyral** that you can adapt to offer the end-user or scriptwriter the capability of specifying ranges of identifiers of artifacts for querying/updating/deleting.

CHAPTER 6

Rally Introspection

The Python toolkit for the Rally REST API makes it easy to obtain the names of Rally types (entities) and the attributes associated with each type. You can also use **pyral** capabilities to obtain the list of allowed values for Rally type attributes that have a pre-allocated list of values.

CHAPTER 7

Queries and Results

The Rally REST API has two interesting characteristics that the Python toolkit for the Rally REST API insulates the scriptwriter from having to deal with. The first is that the Rally REST API has a maximum "pagesize" to limit volume and prevent unwarranted hijacking of the Rally SaaS servers. But, having script writers deal with this directly to obtain further "pages" would be burdensome and out of character with the mainstream of Python interfaces to SaaS services. The Python toolkit for the Rally REST API (**pyral**) takes care of the paging transparently, allowing the scriptwriter to treat a result set as an iterator, merely looping through the results without any indication of any sequence of further requests on the Rally server.

The second characteristic is that the Rally REST API for some queries and results returns not a scalar value but a reference to yet another entity in the Rally system. A Project or a Release are good examples of these. Say your query specified the retrieval of some Stories, and you listed the Project as a field to return with these results. From an end-user perspective, seeing the project name as opposed to an URL with an ObjectID value would be far more intuitive.

The Python toolkit for the Rally REST API offers this sort of intuitive behavior by "chasing" the URL to obtain the more human friendly and intuitive information for display. This sort of behavior is also present in so-called "lazy-evaluation" of entity attributes that may be containers (collections)

> as well as references. The scriptwriter merely has to refer to the attribute with the dot ('.') notation and **pyral** takes care of the communication with the Rally server

to obtain the value. There are two significant advantages to this, one being lightening the load on the server with the reduction of data returned and the other being easy and intuitive attribute access syntax.

The query relational operators that pyral supports are: = != > < >= <= contains !contains in !in between !between

The contains and !contains relational operators a helpful in expressing a condition where you are looking for a field that does (or does not) contain a specific substring. For example 'Name contains "Prior Art"' or 'ThermalPhase !contains "hot lava"'. The in and !in relational operators are commonly used for expressions involving subsets of a finite set of values. For example 'Severity in High, Burning, Explosive' or 'Priority !in Moribund, Meh'. The between and !between relation operators are commonly used for expressions involving date ranges. For example 'CreatedDate between 2018 and 2022' or 'LastUpdated !between 2021-09-22T00:00:00.000Z and 2021-09-22T07:59:59.999Z'.

CHAPTER 8

Custom Fields

Most Artifact types in Rally can be augmented with custom fields. As of Rally WSAPI v2.0, the ElementName for a custom field is prefixed with '**c_**'. The **pyral** toolkit allows you to reference these fields without having to use the '**c_**' prefix. For example, if your custom field has a DisplayName of 'Burnt Offerings Index' you can use the String of 'BurntOfferingsIndex' in a fetch clause or a query clause or refer to the field directly on an artifact as artifact.BurntOfferingsIndex.

CHAPTER 9

PortfolioItem tips

Rally has 4 standard PortfolioItem sub-types (Theme, Strategy, Initiative, and Feature). In this toolkit, for the primary methods (get, create, update, delete), you must supply a entity name (eg, 'Story', 'Defect', 'Task', etc). For a PortfolioItem sub-type you may specify just the name of the sub-type, ie., 'Feature' or you may fully qualify it as 'PortfolioItem/Feature'.

CHAPTER 10

Introduction of Dyna-Types

Prior to the release of Rally WebServices API v2.0, Rally introduced a modification of their data model, which is termed dyna-types. This modification offers a means of establishing and using a parent type and defining sub-types of that parent. The PortfolioItem type is now an "abstract" type from which there are some pre-defined sub-types (Theme, Strategy, Initiative, Feature). By convention, the preferred way to identify a PortfolioItem sub-type is via slashed naming, eg. 'PortfolioItem/Feature'. While it is possible to identify a PortfolioItem sub-type by the sub-type name, eg, (Theme), this is not the preferred means. The reason for the latter statement is that with dyna-types it is possible to define new "abstract" types and define sub-types therefrom that may have names identical to a sub-type whose parent differs from your newly defined "abstract" type.

An example of this is a fictional "abstract" parent type named "Bogutrunk" (for a type that encompasses stories about requests for things you'll never implement and aren't bugs, but you want to track them anyway). Additionally, let's say you define some sub-types whose parent type is "Bogutrunk" and are named "Outlandish", "NonScalable", "Theme" and "Feature". Now, identifying a specific NonScalable Bogutrunk item is unambiguous; you'd just specify the entity in any pyral get/put/post/delete as a "NonScalable". But, you cannot use that convention for a "Feature". You'd need to specify one as a "Bogutrunk/Feature" to disambiguate from a "PortfolioItem/Feature". The main take-away here is that if you don't use PortfolioItem instances with pyral, you don't have worry about this. If you use PortfolioItem instances with pyral or you've defined your own "abstract" parent types and specific sub-types thereof, you are strongly encouraged to use the slashed specification to avoid ambiguity in identifying the Rally entity type.

In the event your organization has created a sub-type with the same name as a standard Rally entity (eg, 'Project', 'Release', 'Milestone', etc.) you will be unable to use this toolkit to access those items. There will be no consideration given to supporting any custom PortfolioItem sub-type whose name conflicts with a Rally standard entity name.

CHAPTER 11

Primary *pyral* classes and functions

For the most part, you'll only be utilizing two main entry points to the **pyral** package.

The first is the **rallySettings** convenience function that you'll use to obtain target and credential values. If you are using 1.1.x or beyond, you may want to alternatively use the **rallyWorkset** convenience function to obtain target and credential values if you intend to use a Rally API Key for credentials. The second is the **Rally** class, from which you'll obtain an instance and then treat that as a direct link to the Rally SaaS. An instance of the Rally class has the basic four CRUD operations as well as several convenience methods to obtain information about workspaces, projects, users and Rally type/value metadata.

You'll also be using the results of queries issued to Rally that are returned as instances of the **RallyRESTResponse** class. Instances of this class allow easy dot ('.') notation access to attributes of the representation of the Rally entity, whether the attribute is a simple value or a reference to another Rally entity.

CHAPTER 12

rallyWorkset

New in 1.1.0.

This function takes into account your environment and arguments provided to this function to arrive at and return information necessary to establish a useful *connection* to the Rally server. This convenience function differs from **rallySettings** by also including the Rally API Key in the set of values considered and returned.

The process consists of a priority chain where some reasonable default information is established first and then overridden with subsequent steps in the chain (if they exist). After following the priority chain, values for server, user, password, apikey, workspace, project are returned to the caller.

The priority chain consists of these steps:

- establish baseline values from values defined in the module containing the rallyWorkset
- **override with any environment variables present from this list:**
  - RALLY_SERVER
  - RALLY_USER
  - RALLY_PASSWORD
  - APIKEY
  - RALLY_WORKSPACE
  - RALLY_PROJECT
  - RALLY_PING
- if present, use information from a rally-<version>.cfg file in the current directory, where <version> matches the Rally WSAPI version defined in the pyral.config module. Currently, that version is defined as v2.0.
- if present, use the contents of a file named in the RALLY_CONFIG environment variable.
- if present, use the contents of a config named on the command line via the –config-<filename> option
- if present, use the values of individual credential/target settings provided as command line options via the –<option>=<value>.

The specific syntax available for these levels is detailed below.

Files

The general syntax is:

- CAP_NAME = value

Valid entries are:

- SERVER = <RallyServer>
- USER = <validUserName>
- PASSWORD = <validPassword>
- APIKEY = <validAPIKey>
- WORKSPACE = <validWorkspaceName>
- PROJECT = <validProjectName>
- RALLY_PING = True | False

Command line options

–rallyConfig=<configFileName>

or –config=<configFileName>

or –conf=<configFileName>

or –cfg=<configFileName>

—rallyServer=<serverName>

–rallyUser=<validRallyUserName>

—rallyPassword=<validRallyPassword>

–apikey=<validRallyAPIKeyValue>

—workspace=<validWorkspaceName>

–project=<validProjectName>

—ping=True|False|true|false|yes|no|1|0

This mechanism provides the ability to centrally locate a configuration file that can be used by many members of a team where server, workspace, project are common to all members and each individual can have their own appropriately secured config file with their credentials. Using this mechanism can save tedious and error-prone entry of target information and credentials on the command line or having credential information in clear text in unsecured files.

The use of a Rally API Key value for identification/authentication is new in pyral 1.1.x. If used, you do not need to provide a username / password combination. In order to use this, you must first obtain a valid API Key value from the Rally Application Manager (API Keys) that you can access from https://rally1.rallydev.com/login. Once obtained, you should treat the key with the same level of protection as you would any user/password information; once presented to Rally via the Rally Web Services API, a connection has all the rights associated with the user whose key was presented. Consult the Rally help documentation for further information.

Example use:

```
% export RALLY_SERVER="rally1.rallydev.com"
% export RALLY_USER="crazedwiley@acmeproducts.com"

% ls -l current.cfg
```

(continues on next page)

(continued from previous page)

```
  -rw-------  1 wiley  eng  173 Jul 14 07:02 current.cfg

% cat current.cfg

  USER = wiley@acme.com
  WORKSPACE = General Products Umbrella
  PROJECT = Dairy Farm Automation

% cat basic.py

import sys
from rally import rallyWorkset

options = [opt for opt in sys.argv[1:] if opt.startswith('--')]
server, user, password, apikey, workspace, project = rallyWorkset(options)
print " ".join(['|%|' % opt for opt in [server, user, password, apikey, workspace,␣
↪project]]


% python basic.py --config=current --rallyProject="Livestock Mgmt" --ping=yes

|rally1.rallydev.com| |wiley@acme.com| |*****| |*****| |General Products Umbrella|␣
↪|Livestock Mgmt|
```

Note that for convenience purposes a configuration file name may be fully specified or you may elect to not specify the '.cfg' suffix.

Returns a tuple of (server, username, password, apikey, workspace, project)

CHAPTER 13

rallySettings

This is deprecated as of v1.2.0. The preferred function is **rallyWorkset** which will have ongoing support. The **rallySettings** function will be removed in v2.0.0.

This function takes into account your environment and arguments provided to this function to arrive at and return information necessary to establish a useful *connection* to the Rally server.

The process consists of a priority chain where some reasonable default information is established first and then overridden with subsequent steps in the chain (if they exist). After following the priority chain, values for server, user, password, workspace, project are returned to the caller.

The priority chain consists of these steps:

- establish baseline values from values defined in the module containing the rallySettings
- **override with any environment variables present from this list:**
  - RALLY_SERVER
  - RALLY_USER
  - RALLY_PASSWORD
  - RALLY_WORKSPACE
  - RALLY_PROJECT
- if present, use information from a rally-<version>.cfg file in the current directory, where <version> matches the Rally WSAPI version defined in the pyral.config module. Currently, that version is defined as v2.0.
- if present, use the contents of a file named in the RALLY_CONFIG environment variable.
- if present, use the contents of a config named on the command line via the –config-<filename> option
- if present, use the values of individual credential/target settings provided as command line options via the –<option>=<value>.

The specific syntax available for these levels is detailed below.

Files

The general syntax is:

- CAP_NAME = value

Valid entries are:

- SERVER = <RallyServer>
- USER = <validUserName>
- PASSWORD = <validPassword>
- APIKEY = <validAPIKey>
- WORKSPACE = <validWorkspaceName>
- PROJECT = <validProjectName>

Command line options

–rallyConfig=<configFileName>

or –config=<configFileName>

or –conf=<configFileName>

or –cfg=<configFileName>

—rallyServer=<serverName>

–rallyUser=<validRallyUserName>

—rallyPassword=<validRallyPassword>

–apikey=<validRallyAPIKeyValue>

—workspace=<validWorkspaceName>

–project=<validProjectName>

This mechanism provides the ability to centrally locate a configuration file that can be used by many members of a team where server, workspace, project are common to all members and each individual can have their own appropriately secured config file with their credentials. Using this mechanism can save tedious and error-prone entry of target information and credentials on the command line or having credential information in clear text in unsecured files.

The use of a Rally API Key value for identification/authentication is new in pyral 1.1.x. If used, you do not need to provide a username / password combination. In order to use this, you must first obtain a valid API Key value from the Rally Application Manager (API Keys) that you can access from https://rally1.rallydev.com/login. Once obtained, you should treat the key with the same level of protection as you would any user/password information; once presented to Rally via the Rally Web Services API, a connection has all the rights associated with the user whose key was presented. Consult the Rally help documentation for further information.

Example use:

```
% export RALLY_SERVER="rally1.rallydev.com"
% export RALLY_USER="crazedwiley@acmeproducts.com"

% ls -l current.cfg

  -rw-------  1 wiley  eng  173 Jul 14 07:02 current.cfg

% cat current.cfg

  USER = wiley@acme.com
  WORKSPACE = General Products Umbrella
```

(continues on next page)

(continued from previous page)

```
  PROJECT = Dairy Farm Automation

% cat basic.py

import sys
from rally import rallyWorkset

options = [opt for opt in sys.argv[1:] if opt.startswith('--')]
server, user, password, apikey, workspace, project = rallyWorkset(options)
print " ".join(['|%|' % opt for opt in [server, user, password, apikey, workspace,␣
↪project]]


% python basic.py --config=current --rallyProject="Livestock Mgmt"

|rally1.rallydev.com| |wiley@acme.com| |*****| |*****| |General Products Umbrella|␣
↪|Livestock Mgmt|
```

Note that for convenience purposes a configuration file name may be fully specified or you may elect to not specify the '.cfg' suffix.

Returns a tuple of (server, username, password, apikey, workspace, project)

CHAPTER 14

Rally

The Rally class is the central focus of the **pyral** package. Instantiation of this class with appropriate and valid target/credential information then provides a means of interacting with the Rally server.

To instantiate a Rally object, you'll need to provide these arguments:

- **server** usually rally1.rallydev.com unless you are using an OnPrem version
- **user** Rally UserName
- **password** Rally password for the given user

either in this specific order or as keyword arguments.

You must either have default **workspace** and **project** values set up for your account

OR

you must provide **workspace** and **project** values that are valid and accessible for your account.

You can optionally specify the following as keyword arguments:

- apikey (alternate credential specification)
- workspace (name of the Rally workspace)
- project (name of the Rally project)
- verify_ssl_cert (True or False, default is True)
- **warn (True or False, default is True)** Controls whether a warning is issued if no project is specified and the default project for the user is not in the workspace specified. Under those conditions, the project is changed to the first project (alphabetic ordering) in the list of projects for the specified workspace.
- **server_ping (True or False, default in v1.3.0 + is False)** Specifies whether a ping attempt will be made to confirm network connectivity to the Rally server prior to making a Rally WSAPI REST request. Organizations may have disabled the ability to make ICMP requests so the ping attempt may fail even though there is network connectivity to the Rally server. For this reason, the use of the ping=True option is discouraged going forward. The ping operation itself will be dropped in the next major release (2.0.0).

- **isolated_workspace (True or False, default in v1.2.0 + is False)** Specifies that the Rally instance will only be used for interacting with a single workspace (either the user's default workspace or the named workspace). Using isolated_workspace=True provides performance benefits for a subscription with many workspaces, but it also means you cannot change the workspace you are working within a single instance of a Rally class, nor can you provide a workspace keyword argument to the get, create, update or delete methods that differs from the workspace identified at instantiation time. For subscriptions with a small to moderate number of workspaces (up to a few dozen), the performance savings will be relatively minor when using isolated_workspace=True vs. isolated_workspace=False. However, for subscriptions with a large number of workspaces, using isolated_workspace=False results in a request to Rally for each workspace, which can result in a noticeable lag before the instantiation statement returns a ready-for-use Rally instance.
- headers dict with entries for name, vendor, version of software/integration using this package.

If you use an apikey value, any user name and password you provide is not considered, the connection attempt will only use the apikey. Consult the Rally Help documentation for Rally Application Manager for information on how to generate an API Key and how to reset or delete an API Key.

Note: If your Subscription administrator has set up your Rally Subscription as "SSO only", then to use **pyral**, you must have your account added to the whitelist in Rally so that you can use either BasicAuth (username and password) or the API Key to authenticate to Rally.

Note: As of the 1.2.2 release, **pyral** offers a means of precisely identifying a Project whose name appears in multiple locations within the forest of Projects with a Workspace. For example, your organization may have several "base" level Projects with sub-trees of Projects. In this scenario, you might have multiple Projects named 'AgileTeam-X' or 'SalesPrep'. By using a Project path component separator of ' // ' (<space><slash><slash><space>) you can specify the unambiguous and unique path to the specific Project of interest. Example: Omnibus // Metallic // Conductive // Copper // Wire . You only have to use this syntax to specify a particular Project if you have multiple instances of that Project that have the same name. There is no provision for supporting the scenario where a Project of the same name exists in the same structural location.

---

class Rally(*server*, *user=None*, *password=None*, *apikey=None*, *workspace=None*, *project=None*, *warn=True*, *server_ping=False*)

Examples:

```
rally = Rally('rally1.rallydev.com', 'chester@corral.com', 'bAbYF@cerZ', server_
↪ping=True)

rally = Rally(server='rally1.rallydev.com', user='mchunko', password='mySEk^et')

rally = Rally(server, user, password, workspace='Division #1 Products', project='ABC')

rally = Rally(server, user, password, workspace='Brontoville', verify_ssl_cert=False,␣
↪warn=False)

rally = Rally(server, apikey="_some-more-numbers", workspace='RockLobster', project=
↪'Fence Posts')

rally = Rally('rally1.rallydev.com', 'chester@corral.com', 'bAbYF@cerZ', headers={
↪'name': 'Fungibles Goods Burn Up/Down', 'vendor': 'Archimedes', 'version': '1.2.3'})
```

14.1 Core REST methods and CRUD aliases

put(*entityName*, *itemData*, *workspace=None*, *project=None*)
This method allows for the creation of a single Rally entity for the given entityName. The data is supplied in a dict and must include settings for all required fields. An attempt to create an entity record for which the operational credentials do not include the privileges to create Rally entity entries will result in a RallyRESTException being generated.

Returns a representation of the item as an instance of a class named for the entity.

create()
alias for put

get(*entityName*, *fetch=False | True | comma_separated_list_of_fields*, *query=None*, *order=None*, ***kwargs*)
This method allows for the retrieval of records for the given entityName. A fetch value of False results in a "shell" record returned with only basic ref attributes having values. If the fetch value is True, a fully hydrated record for each qualifying entity is returned. If the fetch value is a string with a list of comma separated attribute names, those name attributes will be members of each returned entity record.

keyword arguments:

- fetch = True/False or "List,Of,Attributes,We,Are,Interested,In"
- query = 'FieldName = "some value"' or ['EstimatedHours = 10', 'MiddleName != "Shamu"', 'Name contains "foogelhorn pop-tarts"', etc.]
- instance = True/False (defaults to False)
- pagesize = n (defaults to 500)
- start = n (defaults to 1)
- limit = n (defaults to no limit)
- workspace = workspace_name (defaults to current workspace selected)
- project = project_name (defaults to current project selected)
- projectScopeUp = True/False (defaults to False)
- projectScopeDown True/False (defaults to False)
- threads = n (value of 1 insures single-threading, any other value is advisory)

Returns a RallyRESTResponse object that has errors and warnings attributes that should be checked before any further operations on the object are attempted. The Response object supports the iteration protocol so that the results of the `get` can be iterated over via either `for rec in response:` or `response.next()`.

If the instance keyword value is True, then an instance of a Rally entity will be returned instead of a RallyRESTResponse. This can be useful when retrieving an item you know exists and is uniquely identified by your query argument.

The query keyword argument can consist of a String, a List of Strings as <*name*> <*relation*> <*value*> conditions or as a Dictionary where the key-value pairs have an implicit equality relationship and all the resulting conditions are AND'ed together.

Note: If you use a simple query, eg., 'SomeField = "Abc"' then _you_ don't need to use parens (although the Rally REST API does...). If you specify the conditions as in the list variation (see the second example in the query keyword explanation above), then the conditions are AND'ed together in a form suitable for consumption by the Rally REST API.

Caution: If there are any paren characters in a query string, then the toolkit takes a hands-off policy and lets you take the responsibility for specifying the query in a form suitable for the Rally REST WSAPI. (See the Help page for for the Rally REST WSAPI in the Rally web-based product).

If you need to have any OR'ing of conditions, you'll have to construct the entire query yourself in the form of a single String with paren characters in the correct locations to make the query syntactically conformant with the Rally REST WSAPI. Example: query=((Name contains "ABC") OR ((Priority = "1-Critical") AND (Severity != "3-Minor"))) Yes, it's kind of a pain in the ...

Using the characters of '**~**' or '**&**' or 'I' or a backslash '\' within a query expression (eg. 'Name contains "I"') are problematic with the use of this toolkit. A REST request will be issued, but even if there are actual qualifying items that you could observe by using the Rally web GUI, the Rally WSAPI response will not have the correct count or content of the qualifying items. Other workarounds are recommended to deal with this; one way is to post-process the results of a less restrictive criteria to filter or qualify the results to your specific criteria.

Use the instance keyword with **caution**, as an exception will be generated if the query produces no qualifying results. If the query produces more than one qualifying result, you'll only get get the first result with no means to obtain any further qualifying items.

---

find()
alias for get

post(*entityName*, *itemData*, *workspace=None*, *project=None*)
This method allows for updating a single Rally entity record with the data contained in the itemData dict. The itemData dict *must* include a key-value pair for either the ObjectID or when applicable, the FormattedID, that will uniquely identify the entity to be updated. The itemData dict may *not* attempt to change the ObjectID value of the entity as the value for the ObjectID is used to identify the Rally entity to update. An attempt to update an entity record for which the operational credentials do not include the privileges to update will result in a RallyRESTException being generated.

Returns a representation of the updated item as an instance of a class named for the entity.

update()
alias for post

delete(*entityName*, *itemIdent*, *workspace=None*, *project=None*)
This method allows for deleting a single Rally entity record whose ObjectID (or FormattedID) must be present in the itemIdent parameter. An attempt to delete an entity record for which the operational credentials do not include the privileges to delete will result in the generation of a RallyRESTException.

Returns a boolean indication of the disposition of the attempt to delete the item.

search(*keywords*, ***kwargs*)
Given a list of keywords or a string with space separated words, issue the relevant Rally WSAPI search request to find artifacts within the search scope that have any of the keywords in any of the artifact's text fields.

NOTE: The search functionality must be turned on for your subscription to use this method.

keyword arguments:

- projectScopeUp = true/false (defaults to false)
- projectScopeDown = true/false (defaults to false)
- pagesize = n (defaults to 500)
- start = n (defaults to 1)
- limit = n (defaults to no limit)

14.2 pyral.Rally instance convenience methods

enableLogging(*dest=sys.stdout*, *attrget=False*, *append=False*)
Use this to enable logging. *dest* can set to the name of a file or an open file/stream (writable). If *attrget* is set to True, all Rally REST requests that are executed to obtain attribute information will also be logged. Be careful with that as the volume can get quite large. The *append* parameter controls whether any existing file will be appended to or overwritten.

disableLogging()
Disables logging to whatever destination has been previously set up.

subscriptionName()
Returns the name of the subscription for the credentials used to establish the connection with Rally.

setWorkspace(*workspaceName*)
Given a workspaceName, set that as the current workspace and use the ref for that workspace in subsequent interactions with Rally.

getWorkspace()
Returns an instance of a Workspace entity with information about the workspace in the currently active context.

getWorkspaces()
Return a list of Workspace instances that are available for the credentials used to establish the connection with Rally.

setProject(*projectName*)
Given a projectName, set that as the current project and use the ref for that project in subsequent interractions with Rally.

getProject(*name=None*)
Returns a minimally hydrated Project entity instance with the Name and ref of the project in the currently active context if the name keyword arg is not supplied or the Name and ref of the project identified by the value of the name parameter as long as the name identifies a valid project in the currently selected workspace. Returns None if a name parameter is supplied that does not identify a valid project in the currently selected workspace.

getProjects(*workspace=None*)
Return a list of Project instances that are available for the workspace context identified by the workspace keyword argument. If no workspace keyword argument is supplied (or is supplied as None), then the workspace context is that of the currently selected workspace.

getUserInfo(*oid=None*, *username=None*, *name=None*)
A convenience method to collect the information associated with a specific user.

Caller must provide at least one keyword arg and non-None / non-empty value to identify the user target on which to obtain information. The *name* keyword arg is associated with the User.DisplayName attribute. The *username* keyword arg is associated with the User.UserName attribute. If provided, the *oid* keyword argument is used, even if other keyword args are provided. Similarly, if the *username* keyword arg is provided it is used even if the *name* keyword argument is provided.

Returns either a single instance of a User entity when the oid keyword argument matches a User in the system, or a list of User entity items when the username or name keywords are given and are matched by at least one User in the system. Returns None if there is no match in the Rally subscription/workspace for the keyword argument used to identify the user target.

getAllUsers(*workspace=None*)
This method offers a convenient one-stop means of obtaining usable information about all users in the named workspace. If no workspace is specified, then the current context's workspace is used. NOTE: Unless you are using credentials associated with a SubscriptionAdministrator or WorkspaceAdministrator, you will not be able to access a user's UserProfile other than yourself.

Return a list of User instances (fully hydrated for scalar attributes) whose ref and collection attributes will be lazy eval'ed upon access.

typedef (*entityName*)
This method returns a TypeDefinition instance for the given entityName. The is handy for occasions where you need identify a specific entity for something like 'Feature' or 'Theme' when creating or updating a PortfolioItem subclass. Intended usage is to use the return *.ref* attribute. For example, within an info dict, "PortfolioItemType" : rally.typedef('Feature').ref .

getCollection (*collection_url*)
Given a collection_url of the form:

http(s)://<server>(:<port>)/slm/webservice/v2.0/<entity>/OID/<attribute>

issue a request for the url and return back a list of hydrated instances for each item in the collection.

getState (*entityName*, *stateName*)
As of Rally WSAPI 1.37 (Sep 2012), the State attribute is no longer a String value for many entities, it is itself an entity (aka Rally Type). To be able to create (or update) an Artifact's State attribute, you must provide a reference (_ref or ref) in the information dictionary used to populate the Artifact's attributes. This method provides an easy means of obtaining the appropriate entity for the particular entity and state Name you want. Typically the usage would be along the lines of this example:

```
info = { ...., "State" : rally.getState('Feature', 'Discovering').ref, ... })
```

Warning: This method only works with PortfolioItem subclasses at this time. (Theme, Strategy, Initiative, Feature)

getStates (*entityName*)
Given an entityName, returns a list of State instances populated with information about each state value permitted for the entityName.

getAllowedValues (*entityName*, *attributeName*[, *workspace=None*])
Given an entityName and and attributeName (which must be valid for the entityName) issue a request to obtain a list of allowed values for the attribute. For standard attributes in the set of ('Artifacts', 'Attachments', 'Changesets', 'Children', 'Collaborators', 'Defects', 'DefectSuites', 'Discussion', 'Duplicates', 'Milestones', 'Iteration', 'Release', 'Project', 'Owner', 'SubmittedBy', 'Predecessors', 'Successors', 'Tasks', 'TestCases', 'TestSets', 'Results', 'Steps', 'Tags') this method will return a [True] value if the entity identified by entityName actually has the attributeName specified. To get the values associated with the attributes in the aforementioned list you should use the **get()** method with the entityName as the first argument and the singular form of the attribute name as the target of the fetch keyword argument. Of course, this only works with an entity that exists (such as 'Attachment' or 'Milestone' or 'Tag') but not entities named above like 'Discussion', or 'SubmittedBy' or 'Result'. For custom fields though there is no such "disqualification", that is the return value will be either a single value or a list of values regardless of whether the values are relevant to every such entity type or the values are a list that can vary per specific instance of the entity type.

addAttachment (*artifact*, *filename*, *mime_type='text/plain'*)
Given an artifact (actual or FormattedID for an artifact), validate that it exists and then attempt to add an Attachment with the name and contents of filename into Rally and associate that Attachment with the Artifact. Returns the Attachment item.

addAttachments (*artifact*, *attachments*)
Given an artifact (either actual or FormattedID) and a list of dicts with each dict having keys and values for name (or Name), mime_type (or MimeType) and content_type (or ContentType), add an Attachment corresponding to each dict in the attachments list and associate it with the referenced Artifact.

getAttachment (*artifact*, *filename*)
Given a real artifact instance or the FormattedID of an existing artifact, obtain the attachment named by filename. If there is such an attachment, return an Attachment instance with hydration for Name, Size, ContentType, Content, CreationDate and the User that supplied the attachment. If no such attachment is present, return None

getAttachmentNames (*artifact*)
Given a real artifact instance that is hydrated for at least the Attachments attribute, return the names (filenames) of the Attachments associated with the artifact.

getAttachments (*artifact*)
Given a real artifact instance, return a list of Attachment records. Each Attachment record will look like a Rally WSAPI Attachment with the additional Content attribute that will contain the decoded AttachmentContent.

rankAbove (*reference_artifact*, *target_artifact*)
Rank the target_artifact above the reference_artifact.

rankBelow (*reference_artifact*, *target_artifact*)
Rank the target_artifact below the reference_artifact.

rankTop (*target_artifact*)
Rank the target_artifact at the top of the list of ranked Artifacts that the target_artifact exists in.

rankBottom (*target_artifact*)
Rank the target_artifact at the bottom of the list of ranked Artifacts that the target_artifact exists in.

14.3 pyral.Rally experimental convenience methods

addCollectionItems (*target_item*, *collection_items*)
Given a target_item and a homogenous list of items whose type appears as a One to Many relationship in the target item, add the collection_items to the corresponding attribute in the target_item.

```
...
milestones = [milestone_1, milestone_2, milestone_3]
story = rally.get('story', 'US123')
rally.addCollectionItems(story, milestones)
```

Warning: This method only works when the collection attribute on the target_item is Modifiable. Consult the Rally WSAPI documentation for the target_item attributes to see whether the attribute of interest has a notation of 'Collection Modifiable yes'. If there is no 'Colletion Modifiable' notation or the value for that is 'no', then use of this method should not be attempted. At this time, the Rally WSAPI schema endpoint does not include information about 'Collection Modifiable' for any of the attributes, you'll have to consult the documentation.

dropCollectionItems (*target_item*, *collection_items*)
Given a target_item and a homogenous list of items whose type appears as a One to Many relationship in the target item, delete the collection_items to the corresponding attribute in the target_item from the current collection contents for the target_item.

Warning: See note above for the 'addCollectionItems' method. The restrictions there are also applicable to this method.

CHAPTER 15

RallyRESTResponse

A RallyRESTResponse instance is returned from a call to `get` (find) and several of the convenience methods. A instance has the following useful state attributes:

- resource = partial URL identifying the resource for the HTTP Request
- status_code = numeric code for the HTTP Response
- headers = HTTP headers returned
- content = a dict produced by JSON'ifying the HTTP response body
- errors = a list of strings with any Error information
- warnings = a list of strings with any Warning information
- startIndex = natural number index (ie., 1 to _X_)
- pageSize = chunk size returned
- resultCount = total number of items in the set meeting the selection criteria

In addition and usually more importantly, a RallyRESTResponse instance can be used as an iterator over the results.

There are two common means of exercising the iterative nature of the reponse. Use a for loop to obtain each item (you can use this in a list comprehension also) or use the *next* method to obtain the next item in the qualifying result set.

Examples:

```
# regular for loop

response = rally.get('Defect', query=..., ...)
for item in response: print item

# in a list comprehension

response = rally.get('UserStory', query=..., ...)
story_titles = [story.Name for story in response]
```

(continues on next page)

(continued from previous page)

```
# using the next method

response = rally.get('Task', query=..., ...)
task1 = response.next()
```

class RallyRESTResponse

next()

Returns the next item from the set of qualifying items. This method handles any further requests to the server if the next qualifying item is not in the current page of results returned from Rally. If all qualifying items have been returned via this method, this method generates a StopIteration exception.

CHAPTER 16

Item Attributes

Item instances returned from iterating on a RallyRESTResponse object are representations of Rally items. The attributes of each item are accessible via the standard dot (.) notation. The names are identical to those documented in the Rally WS API.

Generally, every concrete instance in the Rally system will have a Name attribute. You can use the **attributes()** method on an instance to obtain the names of all of the attributes available on your specific instance.

So, to obtain the name of a TestCase if you have a TestCase instance, you use testcase.Name, to obtain the formatted ID of a story, use story.FormattedID.

There are two special attributes, *oid* and *ref* that are convenient meta-attributes provided with every instance. The *oid* attribute is an alias for ObjectID and the *ref* attribute is the portion of the _ref attribute containing the entity name and ObjectID value. The ref attribute is suitable for use whenever you want/need to specify the value of a reference field.

Attributes that are classified as references (as opposed to a simple string or integer value) can be accessed and attributes on the referenced item can be obtained. A UserStory (alias for HierarchicalRequirement) can have a parent story. To obtain the parent's FormattedID attribute value, you'd specify thusly: story.Parent.FormattedID.

An attribute can also be a collection. For example, Tasks associated with a UserStory. To access these tasks, you'd iterate over them as in:

```
response = rally.get('UserStory', fetch=True, query='State != "Closed"')
if not response.errors:
    for story in response:
        for task in story.Tasks:
            print task.oid, task.Name, task.ActualHours
```

details()
: This convenience method is available on all *WorkspaceDomain* subclass instances and provides an organized and easy to read multiline string with the content of the instance.

Example:

```
response = rally.get('UserStory', fetch=True, query='FormattedID = S321')
story1 = response.next()
print story1.details()

HierarchicalRequirement
    oid               : 12345678
    ref               : hierarchicalrequirement/12345678
    ObjectID          : 12345678
    _ref              : https://rallydev.rallydev.com/slm/webservice/v2.0/
↪hierarchicalrequirement/412345678
    _CreatedAt        : today at 3:14 am
    _hydrated         : True
    Name              : Filbert nuts should be added to all energy bars
    Subscription      : Subscription.ref   (OID  400060  Name: Company 1)
    Workspace         : Workspace.ref      (OID  722746  Name: Prime Cuts Workspace)
    FormattedID       : S321

    AcceptedDate      : None
    AccountingProjec  : None
    AccountingTask    : None
    AffectedCustomer  :
    Attachments       : []
    Blocked           : False
    Blocker           : None
    Capitalizable     : None
    Changesets        : []
    Children          : []
    CreationDate      : 2016-07-12T09:14:35.852Z
    DefectStatus      : NONE
    Defects           : []
    Description       : As a health conscious PO, I want better nutritional content␣
↪in all bars
    Discussion        : []
    IdeaURL           : <pyral.entity.CustomField object at 0x101931290>
    IdeaVotes         : None
    InProgressDate    : 2016-07-12T09:14:36.098Z
    Iteration         : Iteration.ref               (OID  1242381  Name Iteration 5␣
↪(Summer))
    KanbanState       : Accepted
    LastUpdateDate    : 2016-07-12T09:14:36.237Z
    ...
```

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Index