|  |
| --- |
| # Copyright 2011-2014 Splunk, Inc. |
|  | # |
|  | # Licensed under the Apache License, Version 2.0 (the "License"): you may |
|  | # not use this file except in compliance with the License. You may obtain |
|  | # a copy of the License at |
|  | # |
|  | # http://www.apache.org/licenses/LICENSE-2.0 |
|  | # |
|  | # Unless required by applicable law or agreed to in writing, software |
|  | # distributed under the License is distributed on an "AS IS" BASIS, WITHOUT |
|  | # WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the |
|  | # License for the specific language governing permissions and limitations |
|  | # under the License. |
|  |  |
|  | """The \*\*splunklib.data\*\* module reads the responses from splunkd in Atom Feed  |
|  | format, which is the format used by most of the REST API. |
|  | """ |
|  |  |
|  | from xml.etree.ElementTree import XML |
|  |  |
|  | \_\_all\_\_ = ["load"] |
|  |  |
|  | # LNAME refers to element names without namespaces; XNAME is the same |
|  | # name, but with an XML namespace. |
|  | LNAME\_DICT = "dict" |
|  | LNAME\_ITEM = "item" |
|  | LNAME\_KEY = "key" |
|  | LNAME\_LIST = "list" |
|  |  |
|  | XNAMEF\_REST = "{http://dev.splunk.com/ns/rest}%s" |
|  | XNAME\_DICT = XNAMEF\_REST % LNAME\_DICT |
|  | XNAME\_ITEM = XNAMEF\_REST % LNAME\_ITEM |
|  | XNAME\_KEY = XNAMEF\_REST % LNAME\_KEY |
|  | XNAME\_LIST = XNAMEF\_REST % LNAME\_LIST |
|  |  |
|  | # Some responses don't use namespaces (eg: search/parse) so we look for |
|  | # both the extended and local versions of the following names. |
|  |  |
|  | def isdict(name): |
|  |  return name == XNAME\_DICT or name == LNAME\_DICT |
|  |  |
|  | def isitem(name): |
|  |  return name == XNAME\_ITEM or name == LNAME\_ITEM |
|  |  |
|  | def iskey(name): |
|  |  return name == XNAME\_KEY or name == LNAME\_KEY |
|  |  |
|  | def islist(name): |
|  |  return name == XNAME\_LIST or name == LNAME\_LIST |
|  |  |
|  | def hasattrs(element): |
|  |  return len(element.attrib) > 0 |
|  |  |
|  | def localname(xname): |
|  |  rcurly = xname.find('}') |
|  |  return xname if rcurly == -1 else xname[rcurly+1:] |
|  |  |
|  | def load(text, match=None): |
|  |  """This function reads a string that contains the XML of an Atom Feed, then  |
|  |  returns the  |
|  |  data in a native Python structure (a ``dict`` or ``list``). If you also  |
|  |  provide a tag name or path to match, only the matching sub-elements are  |
|  |  loaded. |
|  |  |
|  |  :param text: The XML text to load. |
|  |  :type text: ``string`` |
|  |  :param match: A tag name or path to match (optional). |
|  |  :type match: ``string`` |
|  |  """ |
|  |  if text is None: return None |
|  |  text = text.strip() |
|  |  if len(text) == 0: return None |
|  |  nametable = { |
|  |  'namespaces': [], |
|  |  'names': {} |
|  |  } |
|  |  root = XML(text) |
|  |  items = [root] if match is None else root.findall(match) |
|  |  count = len(items) |
|  |  if count == 0:  |
|  |  return None |
|  |  elif count == 1:  |
|  |  return load\_root(items[0], nametable) |
|  |  else: |
|  |  return [load\_root(item, nametable) for item in items] |
|  |  |
|  | # Load the attributes of the given element. |
|  | def load\_attrs(element): |
|  |  if not hasattrs(element): return None |
|  |  attrs = record() |
|  |  for key, value in element.attrib.iteritems():  |
|  |  attrs[key] = value |
|  |  return attrs |
|  |  |
|  | # Parse a <dict> element and return a Python dict |
|  | def load\_dict(element, nametable = None): |
|  |  value = record() |
|  |  children = list(element) |
|  |  for child in children: |
|  |  assert iskey(child.tag) |
|  |  name = child.attrib["name"] |
|  |  value[name] = load\_value(child, nametable) |
|  |  return value |
|  |  |
|  | # Loads the given elements attrs & value into single merged dict. |
|  | def load\_elem(element, nametable=None): |
|  |  name = localname(element.tag) |
|  |  attrs = load\_attrs(element) |
|  |  value = load\_value(element, nametable) |
|  |  if attrs is None: return name, value |
|  |  if value is None: return name, attrs |
|  |  # If value is simple, merge into attrs dict using special key |
|  |  if isinstance(value, str): |
|  |  attrs["$text"] = value |
|  |  return name, attrs |
|  |  # Both attrs & value are complex, so merge the two dicts, resolving collisions. |
|  |  collision\_keys = [] |
|  |  for key, val in attrs.iteritems(): |
|  |  if key in value and key in collision\_keys: |
|  |  value[key].append(val) |
|  |  elif key in value and key not in collision\_keys: |
|  |  value[key] = [value[key], val] |
|  |  collision\_keys.append(key) |
|  |  else: |
|  |  value[key] = val |
|  |  return name, value |
|  |  |
|  | # Parse a <list> element and return a Python list |
|  | def load\_list(element, nametable=None): |
|  |  assert islist(element.tag) |
|  |  value = [] |
|  |  children = list(element) |
|  |  for child in children: |
|  |  assert isitem(child.tag) |
|  |  value.append(load\_value(child, nametable)) |
|  |  return value |
|  |  |
|  | # Load the given root element. |
|  | def load\_root(element, nametable=None): |
|  |  tag = element.tag |
|  |  if isdict(tag): return load\_dict(element, nametable) |
|  |  if islist(tag): return load\_list(element, nametable) |
|  |  k, v = load\_elem(element, nametable) |
|  |  return Record.fromkv(k, v) |
|  |  |
|  | # Load the children of the given element. |
|  | def load\_value(element, nametable=None): |
|  |  children = list(element) |
|  |  count = len(children) |
|  |  |
|  |  # No children, assume a simple text value |
|  |  if count == 0: |
|  |  text = element.text |
|  |  if text is None:  |
|  |  return None |
|  |  text = text.strip() |
|  |  if len(text) == 0:  |
|  |  return None |
|  |  return text |
|  |  |
|  |  # Look for the special case of a single well-known structure |
|  |  if count == 1: |
|  |  child = children[0] |
|  |  tag = child.tag |
|  |  if isdict(tag): return load\_dict(child, nametable) |
|  |  if islist(tag): return load\_list(child, nametable) |
|  |  |
|  |  value = record() |
|  |  for child in children: |
|  |  name, item = load\_elem(child, nametable) |
|  |  # If we have seen this name before, promote the value to a list |
|  |  if value.has\_key(name): |
|  |  current = value[name] |
|  |  if not isinstance(current, list):  |
|  |  value[name] = [current] |
|  |  value[name].append(item) |
|  |  else: |
|  |  value[name] = item |
|  |  |
|  |  return value |
|  |  |
|  | # A generic utility that enables "dot" access to dicts |
|  | class Record(dict): |
|  |  """This generic utility class enables dot access to members of a Python  |
|  |  dictionary. |
|  |  |
|  |  Any key that is also a valid Python identifier can be retrieved as a field.  |
|  |  So, for an instance of ``Record`` called ``r``, ``r.key`` is equivalent to  |
|  |  ``r['key']``. A key such as ``invalid-key`` or ``invalid.key`` cannot be  |
|  |  retrieved as a field, because ``-`` and ``.`` are not allowed in  |
|  |  identifiers. |
|  |  |
|  |  Keys of the form ``a.b.c`` are very natural to write in Python as fields. If  |
|  |  a group of keys shares a prefix ending in ``.``, you can retrieve keys as a  |
|  |  nested dictionary by calling only the prefix. For example, if ``r`` contains |
|  |  keys ``'foo'``, ``'bar.baz'``, and ``'bar.qux'``, ``r.bar`` returns a record |
|  |  with the keys ``baz`` and ``qux``. If a key contains multiple ``.``, each  |
|  |  one is placed into a nested dictionary, so you can write ``r.bar.qux`` or  |
|  |  ``r['bar.qux']`` interchangeably. |
|  |  """ |
|  |  sep = '.' |
|  |  |
|  |  def \_\_call\_\_(self, \*args): |
|  |  if len(args) == 0: return self |
|  |  return Record((key, self[key]) for key in args) |
|  |  |
|  |  def \_\_getattr\_\_(self, name): |
|  |  try: |
|  |  return self[name] |
|  |  except KeyError:  |
|  |  raise AttributeError(name) |
|  |  |
|  |  def \_\_delattr\_\_(self, name): |
|  |  del self[name] |
|  |  |
|  |  def \_\_setattr\_\_(self, name, value): |
|  |  self[name] = value |
|  |  |
|  |  @staticmethod |
|  |  def fromkv(k, v): |
|  |  result = record() |
|  |  result[k] = v |
|  |  return result |
|  |  |
|  |  def \_\_getitem\_\_(self, key): |
|  |  if key in self: |
|  |  return dict.\_\_getitem\_\_(self, key) |
|  |  key += self.sep |
|  |  result = record() |
|  |  for k,v in self.iteritems(): |
|  |  if not k.startswith(key): |
|  |  continue |
|  |  suffix = k[len(key):] |
|  |  if '.' in suffix: |
|  |  ks = suffix.split(self.sep) |
|  |  z = result |
|  |  for x in ks[:-1]: |
|  |  if x not in z: |
|  |  z[x] = record() |
|  |  z = z[x] |
|  |  z[ks[-1]] = v |
|  |  else: |
|  |  result[suffix] = v |
|  |  if len(result) == 0: |
|  |  raise KeyError("No key or prefix: %s" % key) |
|  |  return result |
|  |   |
|  |  |
|  | def record(value=None):  |
|  |  """This function returns a :class:`Record` instance constructed with an  |
|  |  initial value that you provide. |
|  |   |
|  |  :param `value`: An initial record value. |
|  |  :type `value`: ``dict`` |
|  |  """ |
|  |  if value is None: value = {} |
|  |  return Record(value) |