lazr.restfulclient Documentation

Release 0.14.5

LAZR Developers

Oct 15, 2022

CONTENTS

A programmable client library that takes advantage of the commonalities among lazr.restful web services to provide added functionality on top of wadllib.

Please see https://dev.launchpad.net/LazrStyleGuide and https://dev.launchpad.net/Hacking for how to develop in this package.

CHAPTER

ONE

TOP-LEVEL OBJECTS

Every web service has a top-level "root" object.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

The root object provides access to service-wide objects.

```
>>> sorted(service.lp_entries)
['featured_cookbook']
```

```
>>> sorted(service.lp_collections)
['cookbooks', 'dishes', 'recipes']
```

1.1 Top-level entries

You can access a top-level entry through attribute access.

```
>>> print service.featured_cookbook.name
Mastering the Art of French Cooking
```

1.2 Top-level collections

You can access a top-level collection through attribute access.

```
>>> len(service.dishes)
3
```

Specific top-level collections may support key-based lookups. For instance, the recipe collection does lookups by recipe ID. This is custom code written for this specific web service, and it won't work in general.

```
>>> print service.recipes[1].dish.name
Roast chicken
```

Looking up an object in a top-level collection triggers an HTTP request, and if the object does not exist on the server, a KeyError is raised.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> debug_service = CookbookWebServiceClient()
send: 'GET ...'
...
```

```
>>> recipe = debug_service.recipes[4]
send: 'GET /1.0/recipes/4 ...'
...
```

```
>>> recipe = debug_service.recipes[1000]
Traceback (most recent call last):
...
KeyError: 1000
```

If you want to look up an object without triggering an HTTP request, you can use parentheses instead of square brackets.

```
>>> recipe = debug_service.recipes(4)
>>> nonexistent_recipe = debug_service.recipes(1000)
```

```
>>> sorted(recipe.lp_attributes)
['http_etag', 'id', 'instructions', ...]
```

The HTTP request, and any potential error, happens when you try to access one of the object's properties.

```
>>> print recipe.instructions
send: 'GET /1.0/recipes/4 ...'
...
Preheat oven to...
```

```
>>> print nonexistent_recipe.instructions
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

This is useful if you plan to invoke a named operation on the object instead of accessing its properties–this can save you an HTTP request and speed up your applicaiton.

Now, let's imagine that a top-level collection is misconfigured. We know that the top-level collection of recipes contains objects whose resource type is 'recipe'. But let's tell lazr.restfulclient that that collection contains objects of type 'cookbook'.

```
>>> from lazr.restfulclient.tests.example import RecipeSet
>>> print RecipeSet.collection_of
recipe
>>> RecipeSet.collection_of = 'cookbook'
```

Looking up an object will give you something that presents the interface of a cookbook.

```
>>> not_really_a_cookbook = debug_service.recipes(2)
>>> sorted(not_really_a_cookbook.lp_attributes)
['confirmed', 'copyright_date', 'cuisine'...]
```

But once you try to access one of the object's properties, and the HTTP request is made...

```
>>> print not_really_a_cookbook.resource_type_link
send: 'GET /1.0/recipes/2 ...'
...
http://cookbooks.dev/1.0/#recipe
```

...the server serves a recipe, and so the client-side object takes on the properties of a recipe. You can only fool lazr.restfulclient up to the point where it has real data to look at.

```
>>> sorted(not_really_a_cookbook.lp_attributes)
['http_etag', 'id', 'instructions', ...]
```

```
>>> print not_really_a_cookbook.instructions
Draw, singe, stuff, and truss...
```

This isn't just a defense mechanism: it's a useful feature when a top-level collection contains mixed subclasses of some superclass. For instance, the launchpadlib library defines the 'people' collection as containing 'team' objects, even though it also contains 'person' objects, which expose a subset of a team's functionality. All objects looked up in that collection start out as team objects, but once an object's data is fetched, if it turns out to actually be a person, it switches from the "team" interface to the "people" interface. (This bit of hackery is necessary because WADL doesn't have an inheritance mechanism.)

If you try to access a property based on a resource type the object doesn't really implement, you'll get an error.

```
>>> not_really_a_cookbook = debug_service.recipes(3)
>>> sorted(not_really_a_cookbook.lp_attributes)
['confirmed', 'copyright_date', 'cuisine'...]
>>> not_really_a_cookbook.cuisine
Traceback (most recent call last):
...
AttributeError: http://cookbooks.dev/1.0/recipes/3 object has no attribute 'cuisine'
```

Cleanup.

```
>>> httplib2.debuglevel = 0
>>> RecipeSet.collection_of = 'recipe'
```

1.3 Versioning

By passing in a 'version' argument to the client constructor, you can access different versions of the web service.

```
>>> print service.recipes[1].self_link
http://cookbooks.dev/1.0/recipes/1
```

```
>>> devel_service = CookbookWebServiceClient(version="devel")
>>> print devel_service.recipes[1].self_link
http://cookbooks.dev/devel/recipes/1
```

You can also forgo the 'version' argument and pass in a service root that incorporates a version string.

```
>>> devel_service = CookbookWebServiceClient(
...     service_root="http://cookbooks.dev/devel/", version=None)
>>> print devel_service.recipes[1].self_link
http://cookbooks.dev/devel/recipes/1
```

1.4 Error reporting

If there's an error communicating with the server, lazr.restfulclient raises HTTPError or an appropriate subclass. The error might be a client-side error (maybe you tried to access something that doesn't exist) or a server-side error (maybe the server crashed due to a bug). The string representation of the error should have enough information to help you figure out what happened.

This example demonstrates NotFound, the HTTPError subclass used when the server sends a 404 error For detailed information about the different HTTPError subclasses, see tests/test_error.py.

```
>>> from lazr.restfulclient.errors import HTTPError
>>> try:
...    service.load("http://cookbooks.dev/")
... except Exception, e:
...     pass
```

```
>>> raise e
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
Response headers:
---
...
content-type: text/plain
...
---
Response body:
---
...
---
```

```
>>> print isinstance(e, HTTPError)
True
```

CHAPTER

TWO

COLLECTIONS

lazr.restful makes collections of data available through Pythonic mechanisms like slices.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

You can iterate through all the items in a collection.

```
>>> names = sorted([recipe.dish.name for recipe in service.recipes])
>>> len(names)
5
>>> names
[u'Baked beans', ..., u'Roast chicken']
```

But it's almost always better to slice them.

```
>>> sorted([recipe.dish.name for recipe in service.recipes[:2]])
[u'Roast chicken', u'Roast chicken']
```

You can get a slice of any collection, so long as you provide start and end points keyed to the beginning of the list. You can't key a slice to the end of the list because it might be expensive to calculate how big the list is.

This set-up code creates a regular Python list of all recipes on the site, for comparison with a lazr.restful Collection object representing the same list.

```
>>> all_recipes = [recipe for recipe in service.recipes]
>>> recipes = service.recipes
```

Calling len() on the Collection object makes sure that the first page of representations is cached, which forces this test to test an optimization.

```
>>> ignored = len(recipes)
```

These tests demonstrate that slicing the collection resource gives the same results as collecting all the entries in the collection, and slicing an ordinary list.

```
>>> def slices_match(slice):
...     """Slice two lists of recipes, then make sure they're the same."""
...     list1 = recipes[slice]
...     list2 = all_recipes[slice]
...     if len(list1) != len(list2):
...         raise ("Lists are different sizes: %d vs. %d" %
...                (len(list1), len(list2)))
```

(continues on next page)

(continued from previous page)

```
...     for index in range(0, len(list1)):
...         if list1[index].id != list2[index].id:
...             raise ("%s doesn't match %s in position %d" %
...                    (list1[index].id, list2[index].id, index))
...     return True
```

```
>>> slices_match(slice(3))
True
>>> slices_match(slice(50))
True
>>> slices_match(slice(1,2))
True
>>> slices_match(slice(2,21))
True
>>> slices_match(slice(2,21,3))
True
```

```
>>> slices_match(slice(0, 200))
True
>>> slices_match(slice(30, 200))
True
>>> slices_match(slice(60, 100))
True
```

```
>>> recipes[5:]
Traceback (most recent call last):
...
ValueError: Collection slices must have a definite, nonnegative end point.
```

```
>>> recipes[10:-1]
Traceback (most recent call last):
...
ValueError: Collection slices must have a definite, nonnegative end point.
```

```
>>> recipes[-1:]
Traceback (most recent call last):
...
ValueError: Collection slices must have a nonnegative start point.
```

```
>>> recipes[:]
Traceback (most recent call last):
...
ValueError: Collection slices must have a definite, nonnegative end point.
```

You can slice a collection that's the return value of a named operation.

```
>>> e_recipes = service.cookbooks.find_recipes(search='e')
>>> len(e_recipes[1:3])
2
```

You can also access individual items in this collection by index.

```
>>> print e_recipes[1].dish.name
Foies de voilaille en aspic
```

```
>>> e_recipes[1000]
Traceback (most recent call last):
...
IndexError: list index out of range
```

2.1 When are representations fetched?

To avoid unnecessary HTTP requests, a representation of a collection is fetched at the last possible moment. Let's see what that means.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
```

```
>>> service = CookbookWebServiceClient()
send: ...
...
```

Just accessing a top-level collection doesn't trigger an HTTP request.

```
>>> recipes = service.recipes
>>> dishes = service.dishes
>>> cookbooks = service.cookbooks
```

Getting the length of the collection, or any entry from the collection, triggers an HTTP request.

```
>>> len(recipes)
send: 'GET /1.0/recipes...
...
```

```
>>> dish = dishes[1]
send: 'GET /1.0/dishes...
...
```

Invoking a named operation will also trigger an HTTP request.

```
>>> cookbooks.find_recipes(search="foo")
send: ...
...
```

Scoped collections work the same way: just getting a reference to the collection doesn't trigger an HTTP request.

```
>>> recipes = dish.recipes
```

But getting any information about the collection triggers an HTTP request.

```
>>> len(recipes)
send: 'GET /1.0/dishes/.../recipes ...
...
```

Cleanup.

```
>>> httplib2.debuglevel = None
```

CHAPTER

THREE

ENTRY MANIPULATION

Objects available through the web interface, such as cookbooks, have a readable interface which is available through direct attribute access.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

```
>>> recipe = service.recipes[1]
>>> print recipe.instructions
You can always judge...
```

These objects may have a number of attributes, as well as associated entries and collections.

```
>>> cookbook = recipe.cookbook
>>> print cookbook.name
Mastering the Art of French Cooking
```

```
>>> len(cookbook.recipes)
2
```

The lp_* introspection methods let you know what you can do with an object. You can also use dir(), but it'll be cluttered with all sorts of other stuff.

```
>>> sorted(dir(cookbook))
[..., 'confirmed', 'copyright_date', 'cover', ... 'find_recipes',
 ..., 'recipes', ...]
>>> sorted(cookbook.lp_attributes)
['confirmed', 'copyright_date', ..., 'resource_type_link', ...,
 'self_link', 'web_link']
```

```
>>> sorted(cookbook.lp_entries)
['cover']
>>> sorted(cookbook.lp_collections)
['recipes']
>>> sorted(cookbook.lp_operations)
['find_recipe_for', 'find_recipes', 'make_more_interesting',
 'replace_cover']
```

Some attributes can only take on certain values. The lp_values_for method will show you these values.

```
>>> sorted(cookbook.lp_values_for('cuisine'))
['American', 'Dessert', u'Fran\xe7aise', 'General', 'Vegetarian']
```

Some attributes don't have a predefined list of acceptable values. For them, lp_values_for() returns None.

```
>>> print cookbook.lp_values_for('copyright_date')
None
```

Some of these attributes can be changed. For example, a client can change a recipe's preparation instructions. When changing attribute values though, the changes are not pushed to the web service until the entry is explicitly saved. This allows the client to batch the changes over the wire for efficiency.

```
>>> recipe.instructions = 'Modified instructions'
>>> print service.recipes[1].instructions
You can always judge...
```

Once the changes are saved though, they are propagated to the web service.

```
>>> recipe.lp_save()
>>> print service.recipes[1].instructions
Modified instructions
```

An entry object is a normal Python object like any other. Attributes of an entry, like 'cuisine' or 'cookbook', are available as attributes on the resource, and may be set. Random strings that are not attributes of the entry cannot be set or read as Python attributes.

```
>>> recipe.instructions = 'Different instructions'
>>> recipe.is_great = True
Traceback (most recent call last):
...
AttributeError: 'Entry' object has no attribute 'is_great'
```

```
>>> recipe.is_great
Traceback (most recent call last):
...
AttributeError: http://cookbooks.dev/1.0/recipes/1 object has no attribute 'is_great'
```

The client can set more than one attribute on an entry at a time: they'll all be changed when the entry is saved.

```
>>> cookbook.cuisine
u'Fran\xe7aise'
>>> cookbook.description
u''
```

```
>>> cookbook.cuisine = 'Dessert'
>>> cookbook.description = "A new description"
>>> cookbook.lp_save()
```

```
>>> cookbook = service.recipes[1].cookbook
```

```
>>> print cookbook.cuisine
Dessert
>>> print cookbook.description
A new description
```

Some of an entry's attributes may take other resources as values.

```
>>> old_cookbook = recipe.cookbook
>>> other_cookbook = service.cookbooks['Everyday Greens']
>>> print other_cookbook.name
Everyday Greens
>>> recipe.cookbook = other_cookbook
>>> recipe.lp_save()
>>> print recipe.cookbook.name
Everyday Greens
```

```
>>> recipe.cookbook = old_cookbook
>>> recipe.lp_save()
```

3.1 Refreshing data

Here are two objects representing recipe #1. We'll fetch a representation for the first object right away...

```
>>> recipe_copy = service.recipes[1]
>>> print recipe_copy.instructions
Different instructions
```

...but retrieve the second object in a way that doesn't fetch its representation.

```
>>> recipe_copy_2 = service.recipes(1)
```

An entry is automatically refreshed after saving.

```
>>> recipe.instructions = 'Even newer instructions'
>>> recipe.lp_save()
>>> print recipe.instructions
Even newer instructions
```

If an old object representing that entry already has a representation, it will still show the old data.

```
>>> print recipe_copy.instructions
Different instructions
```

If an old object representing that entry doesn't have a representation yet, it will show the new data.

```
>>> print recipe_copy_2.instructions
Even newer instructions
```

You can also refresh a resource object manually.

```
>>> recipe_copy.lp_refresh()
>>> print recipe_copy.instructions
Even newer instructions
```

3.2 Bookmarking an entry

You can get an entry's URL from the 'self_link' attribute, save the URL for a while, and retrieve the entry later using the load() function.

```
>>> bookmark = recipe.self_link
>>> new_recipe = service.load(bookmark)
>>> print new_recipe.dish.name
Roast chicken
```

You can bookmark a URI relative to the version of the web service currently in use.

```
>>> cookbooks = service.load("cookbooks")
>>> assert isinstance(cookbooks._wadl_resource.url, basestring)
>>> print cookbooks._wadl_resource.url
http://cookbooks.dev/1.0/cookbooks
>>> print cookbooks['The Joy of Cooking'].self_link
http://cookbooks.dev/1.0/cookbooks/The%20Joy%20of%20Cooking
```

```
>>> cookbook = service.load("/cookbooks/The%20Joy%20of%20Cooking")
>>> assert isinstance(cookbook._wadl_resource.url, basestring)
>>> print cookbook._wadl_resource.url
http://cookbooks.dev/1.0/cookbooks/The%20Joy%20of%20Cooking
>>> print cookbook.self_link
http://cookbooks.dev/1.0/cookbooks/The%20Joy%20of%20Cooking
```

```
>>> service_root = service.load("")
>>> assert isinstance(service_root._wadl_resource.url, basestring)
>>> print service_root._wadl_resource.url
http://cookbooks.dev/1.0/
>>> print service_root.cookbooks['The Joy of Cooking'].name
The Joy of Cooking
```

But you can't provide the web service version and bookmark a URI relative to the service root.

```
>>> cookbooks = service.load("/1.0/cookbooks")
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

(That code attempts to load http://cookbooks.dev/1.0/1.0/cookbooks, which doesn't exist.)

You can't bookmark an absolute or relative URI that has nothing to do with the web service.

```
>>> bookmark = 'http://cookbooks.dev/'
>>> service.load(bookmark)
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

```
>>> service.load("/no-such-url")
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

You can't bookmark the return value of a named operation. This is not really desirable, but that's how things work right now.

```
>>> url_without_type = ('http://cookbooks.dev/1.0/cookbooks' +
...                     '?ws.op=find_recipes&search=a')
>>> service.load(url_without_type)
Traceback (most recent call last):
...
ValueError: Couldn't determine the resource type of...
```

3.3 Moving an entry

Some entries will move to different URLs when a client changes their data attributes. For instance, a cookbook's URL is determined by its name.

```
>>> cookbook = service.cookbooks['The Joy of Cooking']
>>> print cookbook.name
The Joy of Cooking
>>> old_link = cookbook.self_link
>>> print old_link
http://cookbooks.dev/1.0/cookbooks/The%20Joy%20of%20Cooking
>>> cookbook.name = "Another Name"
>>> cookbook.lp_save()
```

Change the name, and you change the URL.

```
>>> new_link = cookbook.self_link
>>> print new_link
http://cookbooks.dev/1.0/cookbooks/Another%20Name
```

Old bookmarks won't work anymore.

```
>>> print service.load(old_link)
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

```
>>> print service.load(new_link).name
Another Name
```

Under the covers though, a refresh of the original object has been retrieved from the web service, so it's safe to continue using, and changing it.

```
>>> cookbook.description = u'This cookbook was renamed'
>>> cookbook.lp_save()
>>> print service.load(new_link).description
This cookbook was renamed
```

It's just as easy to move this cookbook back to the old name.

```
>>> cookbook.name = 'The Joy of Cooking'
>>> cookbook.lp_save()
```

Now the old bookmark works again, and the new bookmark no longer works.

```
>>> print service.load(old_link).name
The Joy of Cooking
```

```
>>> print service.load(new_link)
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

3.4 Validation

Some attributes are subject to validation. For instance, a cookbook's cuisine is limited to one of a few selections.

```
>>> from lazr.restfulclient.errors import HTTPError
>>> def print_error_on_save(entry):
...     try:
...         entry.lp_save()
...     except HTTPError, error:
...         for line in sorted(error.content.splitlines()):
...             print line.decode("utf-8")
...     else:
...         print 'Did not get expected HTTPError!'
```

```
>>> cookbook.cuisine = 'No such cuisine'
>>> print_error_on_save(cookbook)
cuisine: Invalid value "No such cuisine". Acceptable values are: ...
>>> cookbook.cuisine = 'General'
```

Some attributes can't be modified at all.

```
>>> cookbook.copyright_date = None
>>> print_error_on_save(cookbook)
copyright_date: You tried to modify a read-only attribute.
```

If the client tries to save an entry that has more than one problem, it will get back an error message listing all the problems.

```
>>> cookbook.cuisine = 'No such cuisine'
>>> print_error_on_save(cookbook)
```

(continues on next page)

(continued from previous page)

```
copyright_date: You tried to modify a read-only attribute.
cuisine: Invalid value "No such cuisine". Acceptable values are: ...
```

3.5 Server-side data massage

Send bad data and your request will be rejected. But if you send data that's not quite what the server is expecting, the server may accept it while tweaking it. This means that the state of your object after you call lp_save() may be slightly different from the object before you called lp_save().

```
>>> cookbook.lp_refresh()
>>> cookbook.description = "   Some extraneous whitespace  "
>>> cookbook.lp_save()
>>> cookbook.description
u'Some extraneous whitespace'
```

3.6 Data types

Incoming data is serialized from JSON, and all the JSON data types appear to the end-user as native Python data types. But there's no standard serialization for JSON dates, so those are handled separately. From the perspective of the end-user, date and date-time fields always look like Python datetime objects or None.

```
>>> cookbook.copyright_date
datetime.datetime(1995, 1, 1,...)
```

```
>>> from datetime import datetime
>>> cookbook.last_printing = datetime(2009, 1, 1)
>>> cookbook.lp_save()
```

3.6.1 Avoiding conflicts

lazr.restful and lazr.restfulclient work together to try to avoid situations where one person unknowingly overwrites another's work. Here, two different clients are interested in the same lazr.restful object.

```
>>> first_client = CookbookWebServiceClient()
>>> first_cookbook = first_client.load(cookbook.self_link)
>>> first_description = first_cookbook.description
```

```
>>> second_client = CookbookWebServiceClient()
>>> second_cookbook = second_client.load(cookbook.self_link)
>>> second_cookbook.description == first_description
True
```

The first client decides to change the description.

```
>>> first_cookbook.description = 'A description.'
>>> first_cookbook.lp_save()
```

The second client tries to make a conflicting change, but the server detects that the second client doesn't have the latest information, and rejects the request.

```
>>> second_cookbook.description = 'A conflicting description.'
>>> second_cookbook.lp_save()
Traceback (most recent call last):
...
PreconditionFailed: HTTP Error 412: Precondition Failed
...
```

Now the second client has a chance to look at the changes that were made, before making their own changes.

```
>>> second_cookbook.lp_refresh()
>>> print second_cookbook.description
A description.
```

```
>>> second_cookbook.description = 'A conflicting description.'
>>> second_cookbook.lp_save()
```

Conflict detection works even when you operate on an object you retrieved from a collection.

```
>>> first_cookbook = first_client.cookbooks[:10][0]
>>> second_cookbook = second_client.cookbooks[:10][0]
>>> first_cookbook.name == second_cookbook.name
True
```

```
>>> first_cookbook.description = "A description"
>>> first_cookbook.lp_save()
```

```
>>> second_cookbook.description = "A conflicting description"
>>> second_cookbook.lp_save()
Traceback (most recent call last):
...
PreconditionFailed: HTTP Error 412: Precondition Failed
...
```

```
>>> second_cookbook.lp_refresh()
>>> print second_cookbook.description
A description
```

```
>>> second_cookbook.description = "A conflicting description"
>>> second_cookbook.lp_save()
```

```
>>> first_cookbook.lp_refresh()
>>> print first_cookbook.description
A conflicting description
```

3.7 Comparing entries

Two entries are equal if they represent the same state of the same server-side resource.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

What does this mean? Well, two distinct objects that represent the same resource are equal.

```
>>> recipe = service.recipes[1]
>>> recipe_2 = service.load(recipe.self_link)
>>> recipe is recipe_2
False
```

```
>>> recipe == recipe_2
True
>>> recipe != recipe_2
False
```

Two totally different entries are not equal.

```
>>> another_recipe = service.recipes[2]
>>> recipe == another_recipe
False
```

An entry can be compared to None, but the comparison never succeeds.

```
>>> recipe == None
False
```

If one entry represents the current state of the server, and the other is out of date or has client-side modifications, they will not be considered equal.

Here, 'recipe' has been modified and 'recipe_2' represents the current state of the server.

```
>>> recipe.instructions = "Modified for equality testing."
>>> recipe == recipe_2
False
```

After a save, 'recipe' is up to date, and 'recipe_2' is out of date.

```
>>> recipe.lp_save()
>>> recipe == recipe_2
False
```

Refreshing 'recipe_2' brings it up to date, and equality succeeds again.

```
>>> recipe_2.lp_refresh()
>>> recipe == recipe_2
True
```

If you make the *exact same* client-side modifications to two objects representing the same resource, the objects will be considered equal.

```
>>> recipe.instructions = "Modified again."
>>> recipe_2.instructions = recipe.instructions
>>> recipe == recipe_2
True
```

If you then save one of the objects, they will stop being equal, because the saved object has a new ETag.

```
>>> recipe.lp_save()
>>> recipe == recipe_2
False
```

3.8 Server-side permissions

The server may hide some data from you because you lack the permission to see it. To avoid objects that are mysteriously missing fields, the server will serve a special "redacted" value that lets you know you don't have permission to see the data.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

```
>>> cookbook = service.recipes[1].cookbook
>>> print cookbook.confirmed
tag:launchpad.net:2008:redacted
```

If you try to make an HTTP request for the "redacted" value (usually by following a link that you don't know is redacted), you'll get a helpful error.

```
>>> service.load("tag:launchpad.net:2008:redacted")
Traceback (most recent call last):
...
ValueError: You tried to access a resource that you don't have the
server-side permission to see.
```

3.8.1 Deleting an entry

Some entries can be deleted with the lp_delete method.

Before demonstrating this, let's acquire the underlying data model objects so that we can restore the entry later. This is a bit of a hack, but it's a lot less work than any alternative.

```
>>> from lazr.restful.example.base.interfaces import IRecipeSet
>>> from zope.component import getUtility
>>> recipe_set = getUtility(IRecipeSet)
>>> underlying_recipe = recipe_set.get(6)
>>> underlying_cookbook = underlying_recipe.cookbook
```

Now let's delete the entry.

```
>>> recipe = service.recipes[6]
>>> print recipe.lp_delete()
None
```

A deleted entry no longer exists.

```
>>> recipe.lp_refresh()
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

Some entries can't be deleted.

```
>>> cookbook.lp_delete()
Traceback (most recent call last):
...
MethodNotAllowed: HTTP Error 405: Method Not Allowed
...
```

Cleanup: restore the deleted recipe.

```
>>> recipe_set.recipes.append(underlying_recipe)
>>> underlying_cookbook.recipes.append(underlying_recipe)
```

3.8.2 When are representations fetched?

To avoid unnecessary HTTP requests, a representation of an entry is fetched at the last possible moment. Let's see what that means.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
```

```
>>> service = CookbookWebServiceClient()
send: ...
...
```

Here's an entry we got from a lookup operation on a top-level collection. The default top-level lookup operation fetches a representation of an entry immediately so as to immediately signal errors.

```
>>> recipe = service.recipes[1]
send: 'GET /1.0/recipes/1 ...'
...
```

But there's also a lookup operation that only triggers an HTTP request when we try to get some data from the entry:

```
>>> recipe1 = service.recipes(1)
```

This gives a recipe object, because CookbookWebServiceClient happens to know that the 'recipes' collection contains recipe objects.

Here's the dish associated with that original recipe entry. Traversing from one entry to another causes an HTTP request for the first entry (the recipe). Without this HTTP request, there's no way to know the URL of the dish.

```
>>> dish = recipe1.dish
send: 'GET /1.0/recipes/1 ...'
...
```

Note that this request is a request for the _recipe_, not the dish. We don't need to know anything about the dish yet. And now that we have a representation of the recipe, we can traverse from the recipe to its cookbook without making another request.

```
>>> cookbook = recipe1.cookbook
```

Accessing any information about an entry we've traversed to _will_ cause an HTTP request.

```
>>> print dish.name
send: 'GET /1.0/dishes/Roast%20chicken ...'
...
Roast chicken
```

Invoking a named operation also causes one (and only one) HTTP request.

```
>>> recipes = cookbook.find_recipes(search="foo")
send: 'GET /1.0/cookbooks/...ws.op=find_recipes...'
...
```

Even dereferencing an entry from another entry and then invoking a named operation causes only one HTTP request.

```
>>> recipes = recipe1.cookbook.find_recipes(search="bar")
send: 'GET /1.0/cookbooks/...ws.op=find_recipes...'
...
```

In all cases we are able to delay HTTP requests until the moment we need data that can only be found by making those HTTP requests (even if, as in the first example, that data is "does this object exist?). If it turns out we never need that data, we've eliminated a request entirely.

If CookbookWebServiceClient didn't know that the 'recipes' collection contained recipe objects, then doing a lookup on that collection *would* trigger an HTTP request. There'd simply be no other way to know what kind of object was at the other end of the URL.

```
>>> from lazr.restfulclient.tests.example import RecipeSet
>>> old_collection_of = RecipeSet.collection_of
>>> RecipeSet.collection_of = None
```

```
>>> recipe1 = service.recipes[1]
send: 'GET /1.0/recipes/1 ...'
...
```

On the plus side, at least accessing this object's properties doesn't require _another_ HTTP request.

```
>>> print recipe1.instructions
Modified again.
```

Cleanup.

```
>>> RecipeSet.collection_of = old_collection_of
>>> httplib2.debuglevel = 0
```

CHAPTER

FOUR

NAMED OPERATIONS

Entries and collections support named operations: one-off functionality that's been given a name and a set of parameters.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

Arguments to named operations are automatically converted to JSON for transmission over the wire. Here's a named operation that takes a boolean argument.

```
>>> [recipe for recipe in service.cookbooks.find_recipes(
...      search="Chicken", vegetarian=True)]
[]
```

Strings that happen to be numbers are handled properly. Here, if "1.234" were converted into a number at any point in the chain, the 'search' operation on the server wouldn't know how to handle it and the request would fail.

```
>>> [people for people in service.cookbooks.find_recipes(search="1.234")]
[]
```

4.1 Counts

Named operations that return a collection return either a link that lazr.restfulclient can follow to get the size of the collection or (under some circumstances) the length itself. The len() function hides this indirection from the end-user.

```
>>> results = service.cookbooks.find_recipes(search="Chicken")
>>> print len(results)
0
```

4.2 Special data types

lazr.restfulclient uses some data types that don't directly correspond to JSON data types. These data types can be used in named operations. For instance, a named operation can take a date or datetime object as one of its arguments.

```
>>> import datetime
>>> date = datetime.datetime(1994, 1, 1)
>>> cookbook = service.cookbooks.create(
...     name="New cookbook", cuisine="General",
```

(continues on next page)

(continued from previous page)

```
...     copyright_date=date, price=1.23, last_printing=date)
>>> print cookbook.name
New cookbook
```

A named operation can take an entry as one of its arguments. lazr.restfulclient lets you pass in the actual entry as the argument value.

```
>>> dish = service.recipes[1].dish
>>> cookbook = service.recipes[1].cookbook
>>> print cookbook.find_recipe_for(dish=dish)
http://cookbooks.dev/1.0/recipes/1
```

A named operation can take binary data as one of its arguments.

```
>>> cookbook.replace_cover(cover="\x00\xe2\xe3")
>>> cookbook.cover.open().read()
'\x00\xe2\xe3'
```

A named operation that returns a null value corresponds to a Python value of None.

```
>>> dish = service.recipes[4].dish
>>> print cookbook.find_recipe_for(dish=dish)
None
```

A named operation may change the resource's location so we get a 301 response with the new URL.

```
>>> from urllib import quote
>>> from lazr.restful.testing.webservice import WebServiceCaller
>>> webservice = WebServiceCaller(domain='cookbooks.dev')
>>> url = quote("/cookbooks/New cookbook")
>>> print webservice.named_post(url, 'make_more_interesting')
HTTP/1.1 301 Moved Permanently
...
Location: http://cookbooks.dev/devel/cookbooks/The%20New%20New%20cookbook
...
```

4.3 JSON-encoding

lazr.restfulclient encodes most arguments (even string arguments) as JSON before sending them over the wire. This way, a named operation that takes a string argument can take a string that looks like a JSON object without causing confusion.

```
>>> cookbooks = service.cookbooks.find_for_cuisine(cuisine="General")
>>> len([cookbook for cookbook in cookbooks]) > 0
True
```

```
>>> cookbook = service.cookbooks.create(
...     name="null", cuisine="General",
...     copyright_date=date, price=1.23, last_printing=date)
>>> cookbook.name
u'null'
```

```
>>> cookbook = service.cookbooks.create(
...     name="4.56", cuisine="General",
...     copyright_date=date, price=1.23, last_printing=date)
>>> cookbook.name
u'4.56'
```

```
>>> cookbook = service.cookbooks.create(
...     name='"foo"', cuisine="General",
...     copyright_date=date, price=1.23, last_printing=date)
>>> cookbook.name
u'"foo"'
```

A named operation that takes a non-string object (such as a float) will not accept a string that's the JSON representation of the object.

```
>>> try:
...     service.cookbooks.create(
...         name="Yet another 1.23 cookbook", cuisine="General",
...         copyright_date=date, last_printing=date, price="1.23")
... except Exception, e:
...     print e.content
price: got 'unicode', expected float, int: u'1.23'
```

4.4 Named operations on collections don't fetch the collections

If you invoke a named operation on a collection, the only HTTP request made is the one for the named operation. You don't have to get a representation of the collection to invoke the operation.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> service = CookbookWebServiceClient()
send: ...
...
```

```
>>> print service.cookbooks.find_recipes(
...      search="Chicken", vegetarian=True)
send: 'GET /1.0/cookbooks?...vegetarian=true...'
...
```

Cleanup.

```
>>> httplib2.debuglevel = None
```

CHAPTER

FIVE

HOSTED FILES

Some resources published by lazr.restful are externally hosted files that can have binary representations. lazr.restfulclient gives you access to these resources.

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service = CookbookWebServiceClient()
```

An example of a hosted binary file is the cover of a cookbook. "Everyday Greens" starts off with no cover.

```
>>> greens = service.cookbooks['Everyday Greens']
>>> cover = greens.cover
>>> sorted(dir(cover))
[..., 'open']
```

```
>>> cover.open()
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

You can open a hosted file for write access and write to it as though it were a file on disk.

```
>>> image = "Pretend this is an image."
>>> len(image)
25
```

```
>>> file_handle = cover.open("w", "image/png", "a-cover.png")
>>> file_handle.content_type
'image/png'
>>> file_handle.filename
'a-cover.png'
>>> print file_handle.last_modified
None
>>> file_handle.write(image)
>>> file_handle.close()
```

Once it exists on the server, you can open a hosted file for read access and read it.

```
>>> file_handle = cover.open()
>>> file_handle.content_type
'image/png'
```

(continues on next page)

(continued from previous page)

```
>>> file_handle.filename
'0'
>>> last_modified = file_handle.last_modified
>>> last_modified is None
False
>>> len(file_handle.read())
25
```

Note that the filename is '0', not 'a-cover.png'. The filename from the server is implementation-dependent and may not have anything to do with the filename the client sent. If the server implementation uses lazr.librarian, it will serve files with the originally uploaded filename, but the example web service uses its own, simpler implementation which serves the file's ID as the filename.

Modifying a file will change its 'last_modified' attribute.

```
>>> file_handle = cover.open("w", "image/png", "another-cover.png")
>>> file_handle.write(image)
>>> file_handle.close()
```

```
>>> file_handle = cover.open()
>>> file_handle.filename
'1'
>>> last_modified_2 = file_handle.last_modified
>>> last_modified == last_modified_2
False
```

Once a file exists, it can be deleted.

```
>>> cover.delete()
>>> cover.open()
Traceback (most recent call last):
...
NotFound: HTTP Error 404: Not Found
...
```

5.1 Comparing hosted files

Two hosted file objects are the same if they point to the same server-side resource.

```
>>> cover = service.cookbooks['Everyday Greens'].cover
>>> cover_2 = service.cookbooks['Everyday Greens'].cover
>>> cover == cover_2
True
```

```
>>> other_cover = service.cookbooks['The Joy of Cooking'].cover
>>> cover == other_cover
False
```

A hosted file can be compared to None, but the comparison never succeeds.

```
>>> cover == None
False
```

5.2 Error handling

The only access modes supported are 'r' and 'w'.

```
>>> cover.open("r+")
Traceback (most recent call last):
...
ValueError: Invalid mode. Supported modes are: r, w
```

When opening a file for write access, you must specify the content_type argument.

```
>>> cover.open("w")
Traceback (most recent call last):
...
ValueError: Files opened for write access must specify content_type.
```

```
>>> cover.open("w", "image/png")
Traceback (most recent call last):
...
ValueError: Files opened for write access must specify filename.
```

When opening a file for read access, you must *not* specify the content_type or filename arguments–they come from the server.

```
>>> cover.open("r", "image/png")
Traceback (most recent call last):
...
ValueError: Files opened for read access can't specify content_type.
```

```
>>> cover.open("r", filename="foo.png")
Traceback (most recent call last):
...
ValueError: Files opened for read access can't specify filename.
```

5.3 Caching

Hosted file resources implement the normal server-side caching mechanism.

```
>>> file_handle = cover.open("w", "image/png", "image.png")
>>> file_handle.write(image)
>>> file_handle.close()
```

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> service = CookbookWebServiceClient()
```

(continues on next page)

(continued from previous page)

```
send: ...
>>> cover = service.cookbooks['Everyday Greens'].cover
send: ...
```

The first request for a file retrieves the file from the server.

```
>>> len(cover.open().read())
send: ...
reply: '...303 See Other...
reply: '...200 Ok...
25
```

The second request retrieves the file from the cache.

```
>>> len(cover.open().read())
send: ...
reply: '...303 See Other...
reply: '...304 Not Modified...
25
```

Finally, some cleanup code that deletes the cover.

```
>>> cover.delete()
send: 'DELETE...
reply: '...200...
```

```
>>> httplib2.debuglevel = 0
```

CHAPTER

SIX

CACHING

lazr.restfulclient automatically caches the responses to its requests in a temporary directory.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
```

```
>>> from lazr.restfulclient.tests.example import CookbookWebServiceClient
>>> service_with_cache = CookbookWebServiceClient()
send: 'GET /1.0/ ...
reply: ...200...
...
header: Content-Type: application/vnd.sun.wadl+xml
...
send: 'GET /1.0/ ...
reply: ...200...
...
header: Content-Type: application/json
...
```

```
>>> print service_with_cache.recipes[4].instructions
send: 'GET /1.0/recipes/4 ...
reply: ...200...
...
Preheat oven to...
```

The second and subsequent times you request some object, it's likely that lazr.restfulclient will make a conditional HTTP GET request instead of a normal request. The HTTP response code will be 304 instead of 200, and lazr.restfulclient will use the cached representation of the object.

```
>>> print service_with_cache.recipes[4].instructions
send: 'GET /1.0/recipes/4 ...
reply: ...304...
...
Preheat oven to...
```

This is true even if you initially got the object as part of a collection.

```
>>> recipes = service_with_cache.recipes[:10]
send: ...
reply: ...200...
```

```
>>> first_recipe = recipes[0]
>>> first_recipe.lp_refresh()
send: ...
reply: ...304...
```

Note that if you get an object as part of a collection and then get it some other way, a conditional GET request will *not* be made. This is a shortcoming of the library.

```
>>> service_with_cache.recipes[first_recipe.id]
send: ...
reply: ...200...
```

The default lazr.restfulclient cache directory is a temporary directory that's deleted when the Python process ends. (If the process is killed, the directory will stick around in /tmp.) It's much more efficient to keep a cache directory across multiple uses of lazr.restfulclient.

You can provide a cache directory name as argument when creating a Service object. This directory will fill up with cached HTTP responses, and since it's a directory you control it will persist across lazr.restfulclient sessions.

```
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
```

```
>>> first_service = CookbookWebServiceClient(cache=tempdir)
send: 'GET /1.0/ ...
reply: ...200...
...
send: 'GET /1.0/ ...
reply: ...200...
...
```

```
>>> print first_service.recipes[4].instructions
send: 'GET /1.0/recipes/4 ...
reply: ...200...
...
Preheat oven to...
```

This will save you a *lot* of time in subsequent sessions, because you'll be able to use cached versions of the initial (very expensive) documents. A new client will not re-request the service root at all.

```
>>> second_service = CookbookWebServiceClient(cache=unicode(tempdir))
```

You'll also be able to make conditional requests for many resources and avoid transferring their full representations.

```
>>> print second_service.recipes[4].instructions
send: 'GET /1.0/recipes/4 ...
reply: ...304...
...
Preheat oven to...
```

Of course, if you ever need to clear the cache directory, you'll have to do it yourself.

Cleanup.

```
>>> import shutil
>>> shutil.rmtree(tempdir)
```

6.1 Cache expiration

The '1.0' version of the example web service, which we've been using up til now, sets a long cache expiry time for the service root. That's why we were able to create a second client that didn't request the service root at all–just fetched the representations from its cache.

The 'devel' version of the example web service sets a cache expiry time of two seconds. Let's see what that looks like on the client side.

```
>>> tempdir = tempfile.mkdtemp()
>>> first_service = CookbookWebServiceClient(
...     cache=tempdir, version='devel')
send: 'GET /devel/ ...
reply: ...200...
...
send: 'GET /devel/ ...
reply: ...200...
...
```

Now let's wait for three seconds to make sure the representations become stale.

```
>>> from time import sleep
>>> sleep(3)
```

When the representations are stale, a new client makes *conditional* requests for the representations. If the conditions fail (as they do here), the cached representations are considered to have been refreshed, just as if the server had sent them again.

```
>>> second_service = CookbookWebServiceClient(
...     cache=tempdir, version='devel')
send: 'GET /devel/ ...
reply: ...304...
...
send: 'GET /devel/ ...
reply: ...304...
...
```

Let's quickly create another client before the representation grows stale again.

```
>>> second_service = CookbookWebServiceClient(
...     cache=tempdir, version='devel')
```

When the representations are not stale, a new client does not make any HTTP requests at all–it fetches representations direct from the cache.

Cleanup.

```
>>> httplib2.debuglevel = 0
>>> shutil.rmtree(tempdir)
```

6.2 Cache filenames

lazr.restfulclient caches HTTP repsonses in individual files named after the URL accessed. This is behavior derived from httplib2, but lazr.restfulclient does two things differently from httplib2.

To see these two things, let's set up a client that uses a temporary directory as a cache file. The directory starts out empty.

```
>>> from os import listdir
>>> tempdir = tempfile.mkdtemp()
>>> len(listdir(tempdir))
0
```

As soon as we create a client object, though, lazr.restfulclient fetches a JSON and a WADL representation of the service root, and caches them individually.

```
>>> service = CookbookWebServiceClient(cache=tempdir)
>>> cache_contents = listdir(tempdir)
>>> for file in sorted(cache_contents):
...     print file
cookbooks.dev...application,json...
cookbooks.dev...vnd.sun.wadl+xml...
```

This is the first difference between lazr.restfulclient's caching and httplib2's. httplib2 would store all requests for the service root in a filename based solely on the URL. This effectively limits httplib2 to a single representation of a given resource: the WADL representation would be overwritten with the JSON representation. lazr.restfulclient incorporates the media type in the cache filename, so that WADL and JSON representations are stored separately.

The second difference has to do with filename length limits. httplib2 caps filenames at about 240 characters so that cache files can be stored on filesystems with 255-character filename length limits. For compatibility with eCryptfs filesystems, lazr.restfulclient goes further, and caps filenames at 143 characters.

To test out the limit, let's create a cookbook with an incredibly long name.

```
>>> long_name = (
...     "This cookbook name is amazingly long; so long that it will "
...     "surely be truncated when it is incorporated into a file "
...     "name for the cache. The cache file will contain a cached "
...     "HTTP respone containing a JSON representation of of this "
...     "cookbook, whose name, I repeat, is very long indeed.")
>>> len(long_name)
281
```

```
>>> import datetime
>>> date = datetime.datetime(1994, 1, 1)
>>> book = service.cookbooks.create(
...     name=long_name, cuisine="General", copyright_date=date,
...     price=10.22, last_printing=date)
```

lazr.restfulclient automatically fetched a JSON representation of the new cookbook, so it's already present in the cache. Because a cookbook's URL incorporates its name, and this cookbook's name is incredibly long, it must have been truncated to fit on disk.

```
>>> [cookbook_cache_filename] = [file for file in listdir(tempdir)
...                              if 'amazingly' in file]
```

Indeed, the filename has been truncated to fit in the rough 143-character safety limit for eCryptfs filesystems.

```
>>> len(cookbook_cache_filename)
143
```

Despite the truncation, some of the useful information from the cookbook's name makes it into the filename, making it easy to find when manually crawling through the cache directory.

```
>>> print cookbook_cache_filename
cookbooks.dev...This%20cookbook%20name%20is%20amazingly%20long...
```

To avoid conflicts caused by truncation, the filename always ends with an MD5 sum derived from the untruncated URL. Let's create a second cookbook whose name differs from the first cookbook only at the end.

```
>>> longer_name = long_name + ": The Sequel"
>>> book = service.cookbooks.create(
...     name=longer_name, cuisine="General", copyright_date=date,
...     price=10.22, last_printing=date)
```

This cookbook's URL is identical to the first cookbook's URL for far longer than 143 characters. But since the truncated filename incorporates an MD5 sum based on the full URL, the two cookbooks are cached in separate files.

```
>>> [file1, file2] = [file for file in listdir(tempdir)
...                   if 'amazingly' in file]
```

The filenames are identical up to the last 32 characters, which is where the MD5 sum begins. But because the MD5 sums are different, they are not completely identical.

```
>>> file1[:-32] == file2[:-32]
True
```

```
>>> file1 == file2
False
```

Cleanup.

```
>>> import shutil
>>> shutil.rmtree(tempdir)
```

CHAPTER

SEVEN

AUTHORIZERS

Authorizers are objects that encapsulate knowledge about a particular web service's authentication scheme. lazr.restfulclient includes authorizers for common HTTP authentication schemes.

7.1 The BasicHttpAuthorizer

This authorizer handles HTTP Basic Auth. To test it, we'll create a fake web service that serves some sample WADL.

```
>>> import pkg_resources
>>> wadl_string = pkg_resources.resource_string(
...     'wadllib.tests.data', 'launchpad-wadl.xml')
```

```
>>> responses = { 'application/vnd.sun.wadl+xml' : wadl_string,
...               'application/json' : '{}' }
```

```
>>> def sample_application(environ, start_response):
...     media_type = environ['HTTP_ACCEPT']
...     content = responses[media_type]
...     start_response(
...         '200', [('Content-type', media_type)])
...     return [content]
```

The WADL file will be protected with HTTP Basic Auth. To access it, you'll need to provide a username of "user" and a password of "password".

```
>>> def authenticate(username, password):
...     """Accepts "user/password", rejects everything else.
...
...     :return: The username, if the credentials are valid.
...              None, otherwise.
...     """
...     if username == "user" and password == "password":
...         return username
...     return None
```

```
>>> from lazr.authentication.wsgi import BasicAuthMiddleware
>>> def protected_application():
...     return BasicAuthMiddleware(
...         sample_application, authenticate_with=authenticate)
```

Finally, we'll set up a WSGI intercept so that we can test the web service by making HTTP requests to http://api.launchpad.dev/. (This is the hostname mentioned in the WADL file.)

```
>>> import wsgi_intercept
>>> from wsgi_intercept.httplib2_intercept import install
>>> install()
>>> wsgi_intercept.add_wsgi_intercept(
...     'api.launchpad.dev', 80, protected_application)
```

With no HttpAuthorizer, a ServiceRoot can't get access to the web service.

```
>>> from lazr.restfulclient.resource import ServiceRoot
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

We can't get access if the authorizer doesn't have the right credentials.

```
>>> from lazr.restfulclient.authorize import BasicHttpAuthorizer
```

```
>>> bad_authorizer = BasicHttpAuthorizer("baduser", "badpassword")
>>> client = ServiceRoot(bad_authorizer, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

If we provide the right credentials, we can retrieve the WADL. We'll still get an exception, because our fake web service is too fake for ServiceRoot–its 'service root' resource doesn't match the WADL–but we're able to make HTTP requests without getting 401 errors.

Note that the HTTP request includes the User-Agent header, but that that header contains no special information about the authorization method. This will change when the authorization method is OAuth.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
```

```
>>> authorizer = BasicHttpAuthorizer("user", "password")
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
send: 'GET / ...user-agent: lazr.restfulclient ...'
...
```

The BasicHttpAuthorizer allows you to adds proper basic auth headers to the request, when asked to, using the username and password information it already knows about.

```
>>> headers = {}
>>> authorizer.authorizeRequest('/', 'GET', '', headers)
>>> headers.get('authorization')
'Basic dXNlcjpwYXNzd29yZA=='
```

Teardown.

```
>>> httplib2.debuglevel = 0
>>> _ = wsgi_intercept.remove_wsgi_intercept("api.launchpad.dev", 80)
```

7.2 The OAuthAuthorizer

This authorizer handles OAuth authorization. To test it, we'll protect the sample application with a piece of OAuth middleware. The middleware will accept only one consumer/token combination, though it will also allow anonymous access: if you pass in an empty token and secret, you'll get a lower level of access.

```
>>> from oauth.oauth import OAuthConsumer, OAuthToken
>>> valid_consumer = OAuthConsumer("consumer", '')
>>> valid_token = OAuthToken("token", "secret")
>>> empty_token = OAuthToken("", "")
```

Our authenticate() implementation checks against the one valid consumer and token.

```
>>> def authenticate(consumer, token, parameters):
...     """Accepts the valid consumer and token, rejects everything else.
...
...     :return: The consumer, if the credentials are valid.
...              None, otherwise.
...     """
...     if token.key == '' and token.secret == '':
...         # Anonymous access.
...         return consumer
...     if consumer == valid_consumer and token == valid_token:
...         return consumer
...     return None
```

Our data store helps the middleware look up consumer and token objects from the information provided in a signed OAuth request.

```
>>> from lazr.authentication.testing.oauth import SimpleOAuthDataStore
```

```
>>> class AnonymousAccessDataStore(SimpleOAuthDataStore):
...     """A data store that will accept any consumer."""
...     def lookup_consumer(self, consumer):
...         """If there's no matching consumer, just create one.
...
...         This will let anonymous requests succeed with any
...         consumer key."""
...         consumer = super(
...             AnonymousAccessDataStore, self).lookup_consumer(
...             consumer)
...         if consumer is None:
...             consumer = OAuthConsumer(consumer, '')
...         return consumer
```

```
>>> data_store = AnonymousAccessDataStore(
...     {valid_consumer.key : valid_consumer},
```

(continues on next page)

(continued from previous page)

```
...     {valid_token.key : valid_token,
...      empty_token.key : empty_token})
```

Now we're ready to protect the sample_application with OAuthMiddleware, using our authenticate() implementation and our data store.

```
>>> from lazr.authentication.wsgi import OAuthMiddleware
>>> def protected_application():
...     return OAuthMiddleware(
...         sample_application, realm="OAuth test",
...         authenticate_with=authenticate, data_store=data_store)
>>> wsgi_intercept.add_wsgi_intercept(
...     'api.launchpad.dev', 80, protected_application)
```

Let's try out some clients. As you'd expect, you can't get through the middleware with no HTTPAuthorizer at all.

```
>>> from lazr.restfulclient.authorize.oauth import OAuthAuthorizer
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

Invalid credentials are also no help.

```
>>> authorizer = OAuthAuthorizer(
...     valid_consumer.key, access_token=OAuthToken("invalid", "token"))
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

But valid credentials work fine (again, up to the point at which lazr.restfulclient runs against the limits of this simple web service). Note that the User-Agent header mentions the consumer key.

```
>>> httplib2.debuglevel = 1
>>> authorizer = OAuthAuthorizer(
...     valid_consumer.key, access_token=valid_token)
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
send: 'GET /...user-agent: lazr.restfulclient...; oauth_consumer="consumer"...'
...
```

If the OAuthAuthorizer is created with an application name as well as a consumer key, the application name is mentioned in the User-Agent header as well.

```
>>> authorizer = OAuthAuthorizer(
...     valid_consumer.key, access_token=valid_token,
...     application_name="the app")
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
send: 'GET /...user-agent: lazr.restfulclient...; application="the app"; oauth_consumer=
↪"consumer"...'
...
```

```
>>> httplib2.debuglevel = 0
```

It's even possible to get anonymous access by providing an empty access token.

```
>>> authorizer = OAuthAuthorizer(
...     valid_consumer.key, access_token=empty_token)
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
```

Because of the way the AnonymousAccessDataStore (defined earlier in the test) works, you can even get anonymous access by specifying an OAuth consumer that's not in the server-side list of valid consumers.

```
>>> authorizer = OAuthAuthorizer(
...     "random consumer", access_token=empty_token)
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
```

A ServiceRoot object has a 'credentials' attribute which contains the Authorizer used to authorize outgoing requests.

```
>>> from lazr.restfulclient.resource import ServiceRoot
>>> root = ServiceRoot(authorizer, "http://api.launchpad.dev/")
>>> root.credentials
<lazr.restfulclient.authorize.oauth.OAuthAuthorizer object...>
```

If you try to provide credentials with an unrecognized OAuth consumer, you'll get an error–even if the credentials are valid. The data store used in this test only lets unrecognized OAuth consumers through when they request anonymous access.

```
>>> authorizer = OAuthAuthorizer(
...     'random consumer', access_token=valid_token)
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

```
>>> authorizer = OAuthAuthorizer(
...     'random consumer', access_token=OAuthToken("invalid", "token"))
>>> client = ServiceRoot(authorizer, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
Unauthorized: HTTP Error 401: Unauthorized
...
```

Teardown.

```
>>> _ = wsgi_intercept.remove_wsgi_intercept("api.launchpad.dev", 80)
```

CHAPTER

EIGHT

RETRY REQUESTS ON SERVER ERROR

If lazr.restfulclient talks to a server that sends out a server-side error with status codes 502 or 503, the client will wait a few seconds and try the request again. Eventually it will give up and escalate the error code in the form of an exception.

To test this, let's simulate a lazr.restful server prone to transient errors using a WSGI application.

```
>>> import pkg_resources
>>> wadl_string = pkg_resources.resource_string(
...     'wadllib.tests.data', 'launchpad-wadl.xml')
>>> representations = { 'application/vnd.sun.wadl+xml' : wadl_string,
...                     'application/json' : '{}' }
```

This application will cause one request to fail for every item in its BROKEN_RESPONSES list.

```
>>> BROKEN_RESPONSES = []
>>> def broken_application(environ, start_response):
...     if len(BROKEN_RESPONSES) > 0:
...         start_response(str(BROKEN_RESPONSES.pop()),
...                        [('Content-type', 'text/plain')])
...         return ["Sorry, I'm still broken."]
...     else:
...         media_type = environ['HTTP_ACCEPT']
...         content = representations[media_type]
...         start_response(
...             '200', [('Content-type', media_type)])
...         return [content]
```

```
>>> def make_broken_application():
...     return broken_application
```

```
>>> import wsgi_intercept
>>> wsgi_intercept.add_wsgi_intercept(
...     'api.launchpad.dev', 80, make_broken_application)
>>> BROKEN_RESPONSES = []
```

```
>>> from wsgi_intercept.httplib2_intercept import install
>>> install()
```

Here's a fake implementation of time.sleep() so that this test doesn't take a really long time to run, and so we can visualize sleep() being called as lazr.restfulclient retries over and over again.

```
>>> def fake_sleep(time):
...     print "sleep(%s) called" % time
>>> import lazr.restfulclient._browser
>>> old_sleep = lazr.restfulclient._browser.sleep
>>> lazr.restfulclient._browser.sleep = fake_sleep
```

As it starts out, the application isn't broken at all.

```
>>> from lazr.restfulclient.resource import ServiceRoot
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
```

Let's queue up one broken response. The client will sleep once and try again.

```
>>> BROKEN_RESPONSES = [502]
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
sleep(0) called
```

Now the application will fail six times and then start working.

```
>>> BROKEN_RESPONSES = [502, 503, 502, 503, 502, 503]
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
sleep(0) called
sleep(1) called
sleep(2) called
sleep(4) called
sleep(8) called
sleep(16) called
```

Now the application will fail seven times and then start working. But the client will give up before then–it will only retry the request six times.

```
>>> BROKEN_RESPONSES = [502, 503, 502, 503, 502, 503, 502]
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
ServerError: HTTP Error 502:
...
```

By increasing the 'max_retries' constructor argument, we can make the application try more than six times, and eventually succeed.

```
>>> BROKEN_RESPONSES = [502, 503, 502, 503, 502, 503, 502]
>>> client = ServiceRoot(None, "http://api.launchpad.dev/",
...                      max_retries=10)
sleep(0) called
sleep(1) called
sleep(2) called
sleep(4) called
sleep(8) called
sleep(16) called
sleep(32) called
```

Now the application will fail once and then give a 400 error. The client will not retry in hopes that the 400 error will go away–400 is a client error.

```
>>> BROKEN_RESPONSES = [502, 400]
>>> client = ServiceRoot(None, "http://api.launchpad.dev/")
Traceback (most recent call last):
...
BadRequest: HTTP Error 400:
...
```

Teardown.

```
>>> _ = wsgi_intercept.remove_wsgi_intercept("api.launchpad.dev", 80)
>>> lazr.restfulclient._browser.sleep = old_sleep
```

CHAPTER

NINE

CONTRIBUTING

To run this project's tests, use tox.

To update the project's documentation you need to trigger a manual build on the project's dashboard on https://readthedocs.org.

9.1 Getting help

If you find bugs in this package, you can report them here:

> https://launchpad.net/lazr.restfulclient

If you want to discuss this package, join the team and mailing list here:

> https://launchpad.net/~lazr-developers

or send a message to:

> lazr-developers@lists.launchpad.net

CHAPTER

TEN

NEWS FOR LAZR.RESTFULCLIENT

10.1 0.14.5 (2022-10-15)

- Add `pre-commit` configuration.
- Publish documentation on Read the Docs.
- Apply black code formatter.
- Apply inclusive naming via the woke pre-commit hook.
- Fix `HostedFile.open(mode="w")` on Python 3.

10.2 0.14.4 (2021-09-13)

- Drop support for Python < 2.6.
- Adjust versioning strategy to avoid importing pkg_resources, which is slow in large environments.

10.3 0.14.3 (2020-01-27)

- Restore from_string, to_string, and __str__ methods of lazr.restfulclient.authorize.oauth.AccessToken, unintentionally removed in 0.14.0.

10.4 0.14.2 (2018-11-17)

- Fix compatibility with httplib2 0.12.0 for Python 3. [bug=1803754]
- Really fix compatibility with httplib2 < 0.9.
- Fix compatibility with httplib2 0.9 for Python 3.
- Require httplib2 >= 0.7.7 for Python 3.

10.5 0.14.1 (2018-11-16)

- Add compatibility with httplib2 0.12.0. [bug=1803558]

10.6 0.14.0 (2018-05-08)

- Switch from buildout to tox.
- Port from oauth to oauthlib. Some tests still need to use oauth until lazr.authentication is ported. [bug=1672458]
- Use the distro module rather than platform.linux_distribution, since the latter is deprecated in Python 3.5 and will be removed in 3.7. [bug=1473577]

10.7 0.13.5 (2017-09-04)

- Fix bytes vs. unicode in json.loads calls. [bug=1403524]
- Decode header before comparison. [bug=1414075]
- Fix urllib unquote imports. [bug=1414055]
- Fix urllib urlencode imports. [bug=1425609]
- Tolerate httplib2 versions earlier than 0.9 again.
- Fix handling of 304 responses with an empty body on Python 3. [bug=1714960]

10.8 0.13.4 (2014-12-05)

- Port to python3.
- Support proxy settings from environment by default.

10.9 0.13.3 (2013-03-22)

- Fall back to httplib2's default certificate path if the Debian/Ubuntu one doesn't exist. The default bundle might work, but a path that doesn't exist is never going to. New httplib2 bundles contain the required CA certs.

10.10 0.13.2 (2012-12-06)

- lazr.restfulclient is almost exclusively used with launchpad.net, but httplib2's cert bundle doesn't include launchpad's CA. Therefore with the default setup launchpadlib doesn't work unless cert checking is disabled. This is mitigated by the fact that Ubuntu carries a patch to httplib2 to make it use the system CA certs. This release makes that the default approach in lazr.restfulclient so that launchpad.net can be used by anyone with the Debian/Ubuntu CA certs path (/etc/ssl/certs/ca-certificates.crt), regardless of whether they are using Ubuntu's patched version of httplib2. Any platforms that don't have that path remain broken.

10.11 0.13.1 (2012-09-26)

- Named POST operations may result in a resource moving to a new location. Detect the redirect and reload the resource from its new URL.

10.12 0.13.0 (2012-06-19)

- Add environment variable, LP_DISABLE_SSL_CERTIFICATE_VALIDATION, to disable SSL certificate checks. Most useful when testing against development servers.

10.13 0.12.3 (2012-05-17)

- Implement the mocked out authorizeRequest for the BasicHttpAuthorizer object.

10.14 0.12.2 (2012-04-16)

- Fix ServiceRoot.load() so that it properly handles relative URLs in a way that doesn't break subsequent API calls (bug 681767).

10.15 0.12.1 (2012-03-28)

- Made the cache safe for use by concurrent threads and processes.

10.16 0.12.0 (2011-06-30)

- Give a more useful AttributeError

10.17 0.11.2 (2011-02-03)

- The 'web_link' parameter now shows up in lp_attributes, not lp_entries.

10.18 0.11.1 (2010-11-04)

- Restored compatibility with Python 2.4.

10.19 0.11.0 (2010-10-28)

- Make it possibly to specify an "application name" separate from the OAuth consumer key. If present, the application name is used in the User-Agent header; otherwise, the OAuth consumer key is used.
- Add a "system-wide consumer" which can be used to authorize a user's entire account to use a web service, rather than doing it one application at a time.

10.20 0.10.0 (2010-08-12)

- Add compatibility with lazr.restful 0.11.0

10.21 0.9.21 (2010-07-19)

- Ensure that all JSON representations are converted to Unicode.
- Restore the old behavior of CollectionWithKeyBasedLookup, which is less efficient but easier to understand. That is, the following code will work as it did in 0.9.17, performing the lookup immediately and raising a KeyError if the object doesn't exist on the server side.

  service.collection['key']

  The more efficient behavior (which doesn't perform the lookup until you actually need the object) is still available, but you have to write this code instead:

  service.collection('key')
- Exceptional conditions will now raise an appropriate subclass of HTTPError instead of always raising HTTPError.
- Credential files are now created as being user-readable only. (In launchpadlib, they were created using the default umask and then made user-readable with chmod.)

10.22 0.9.20 (2010-06-25)

- It's now possible to pass a relative URL (relative to the versioned service root) into load().

10.23 0.9.19 (2010-06-21)

- When the representation of a resource, as retrieved from the server, is of a different type than expected, the server value now takes precedence. This means that, in rare situations, a resource may start out presumed to be of one type, and change its capabilities once its representation is fetched from the server.

10.24 0.9.18 (2010-06-16)

- Made it possible to avoid fetching a representation of every single object looked up from a CollectionWithKeyBasedLookup (by defining .collection_of on the class), potentially improving script performance.

10.25 0.9.17 (2010-05-10)

- Switched back to asking for compression using the standard Accept-Encoding header. Using the TE header has never worked in a real situation due to HTTP intermediaries.

10.26 0.9.16 (2010-05-03)

- If a server returns a 502 or 503 error code, lazr.restfulclient will retry its request a configurable number of times in hopes that the error is transient.
- It's now possible to invoke lazr.restful destructor methods, with the lp_delete() method.

10.27 0.9.15 (2010-04-27)

- Clients will no longer fetch a representation of a collection before invoking a named operation on the collection.

10.28 0.9.14 (2010-04-15)

- Clients now send a useful and somewhat customizable User-Agent string.
- Added a workaround for a bug in httplib2.
- Removed the software dependency on lazr.restful except when running the full test suite. (The standalone_test test suite tests basic functionality of lazr.restfulclient to make sure the code base doesn't fundamentally depend on lazr.restful.)

10.29 0.9.13 (2010-03-24)

- Removed some no-longer-needed compatibility code for buggy servers, and fixed the tests to work with the new release of simplejson.
- The fix in 0.9.11 to avoid errors on eCryptfs filesystems wasn't strict enough. The maximum filename length is now 143 characters.

10.30 0.9.12 (2010-03-09)

- Fixed a bug that prevented a unicode string from being used as a cache filename.

10.31 0.9.11 (2010-02-11)

- If a lazr.restful web service publishes multiple versions, you can now specify which version to use in a separate constructor argument, rather than sticking it on to the end of the service root.
- Filenames in the cache will never be longer than 150 characters, to avoid errors on eCryptfs filesystems.
- Added a proof-of-concept test for OAuth-signed anonymous access.
- Fixed comparisons of entries and hosted files with None.

10.32 0.9.10 (2009-10-23)

- lazr.restfulclient now requests the correct WADL media type.
- Made HTTPError strings more verbose.
- Implemented the equality operator for entry and hosted-file resources.
- Resume setting the 'credentials' attribute on ServerRoot to avoid breaking compatibility with launchpadlib.

10.33 0.9.9 (2009-10-07)

- The WSGI authentication middleware has been moved from lazr.restful to the new lazr.authentication library, and lazr.restfulclient now uses the new library.

10.34 0.9.8 (2009-10-06)

- Added support for OAuth.

10.35 0.9.7 (2009-09-30)

- Added support for HTTP Basic Auth.

10.36 0.9.6 (2009-09-16)

- Made compatible with lazr.restful 0.9.6.

10.37 0.9.5 (2009-08-28)

- Removed debugging code.

10.38 0.9.4 (2009-08-26)

- Removed unnecessary build dependencies.
- Updated tests for newer version of simplejson.
- Made tests less fragile by cleaning up lazr.restful example filemanager between tests.
- normalized output of simplejson to unicode.

10.39 0.9.3 (2009-08-05)

Removed a sys.path hack from setup.py.

10.40 0.9.2 (2009-07-16)

- Fields that can contain binary data are no longer run through simplejson.dumps().
- For fields that can take on a limited set of values, you can now get a list of possible values.

10.41 0.9.1 (2009-07-13)

- The client now knows to look for multipart/form-data representations and will create them as appropriate. The upshot of this is that you can now send binary data when invoking named operations that will accept binary data.

10.42 0.9 (2009-04-29)

- Initial public release

CHAPTER

ELEVEN

IMPORTABLE

The lazr.restfulclient package is importable, and has a version number.

```
>>> import lazr.restfulclient
>>> print 'VERSION:', lazr.restfulclient.__version__
VERSION: ...
```