python-caldav Documentation

Release 0.8.2

Cyril Robert

Jan 07, 2022

Contents

CHAPTER 1

Contents

1.1 `DAVClient` – A simple DAV client

class `caldav.davclient.`**`DAVClient`**(*url*, *proxy=None*, *username=None*, *password=None*, *auth=None*, *ssl_verify_cert=True*, *ssl_cert=None*)

Basic client for webdav, uses the requests lib; gives access to low-level operations towards the caldav server.

Unless you have special needs, you should probably care most about the constructor (__init__), the principal method and the calendar method.

`calendar`(***kwargs*)

Returns a calendar object.

Typically, an URL should be given as a named parameter (url)

No network traffic will be initiated by this method.

If you don't know the URL of the calendar, use client.principal().calendar(. . .) instead, or client.principal().calendars()

`delete`(*url*)

Send a delete request.

`mkcalendar`(*url*, *body=''*, *dummy=None*)

Send a mkcalendar request.

Parameters:

- url: url for the root of the mkcalendar
- body: XML request
- dummy: compatibility parameter

Returns

- DAVResponse

`mkcol` (*url*, *body*, *dummy=None*)
Send a MKCOL request.

MKCOL is basically not used with caldav, one should use MKCALENDAR instead. However, some calendar servers MAY allow "subcollections" to be made in a calendar, by using the MKCOL query. As for 2020-05, this method is not exercised by test code or referenced anywhere else in the caldav library, it's included just for the sake of completeness. And, perhaps this DAVClient class can be used for vCards and other WebDAV purposes.

Parameters:

- url: url for the root of the mkcol
- body: XML request
- dummy: compatibility parameter

Returns

- DAVResponse

`post` (*url*, *body*, *headers={}*)
Send a POST request.

`principal` (**largs*, ***kwargs*)
Convenience method, it gives a bit more object-oriented feel to write client.principal() than Principal(client).

This method returns a `caldav.Principal` object, with higher-level methods for dealing with the principals calendars.

`propfind` (*url=None*, *props="*, *depth=0*)
Send a propfind request.

Parameters:

- url: url for the root of the propfind.
- props = (xml request), properties we want
- depth: maximum recursion depth

Returns

- DAVResponse

`proppatch` (*url*, *body*, *dummy=None*)
Send a proppatch request.

Parameters:

- url: url for the root of the propfind.
- body: XML propertyupdate request
- dummy: compatibility parameter

Returns

- DAVResponse

`put` (*url*, *body*, *headers={}*)
Send a put request.

`report` (*url*, *query="*, *depth=0*)
Send a report request.

Parameters:

- url: url for the root of the propfind.
- query: XML request
- depth: maximum recursion depth

Returns

- DAVResponse

request (*url*, *method='GET'*, *body="*, *headers={}*)
Actually sends the request

verify_login (*url=None*, *method='PROPFIND'*, *body="*, *headers={}*)
Will do the following: * run a test request without auth towards the server. * assert it returns 401 * read the WWW-Authenticate header and decide what kind of auth object to create * run a test query with auth * assert it returns 2xx or 3xx In 0.9, it should return True or raise an exception In 0.8.2, it may log an error and return False

class caldav.davclient.**DAVResponse** (*response*)
This class is a response from a DAV request. It is instantiated from the DAVClient class. End users of the library should not need to know anything about this class. Since we often get XML responses, it tries to parse it into *self.tree*

expand_simple_props (*props=[]*, *multi_value_props=[]*, *xpath=None*)
The find_objects_and_props() will stop at the xml element below the prop tag. This method will expand those props into text.

Executes find_objects_and_props if not run already, then modifies and returns self.objects.

find_objects_and_props ()
Check the response from the server, check that it is on an expected format, find hrefs and props from it and check statuses delivered.

The parsed data will be put into self.objects, a dict {href: {proptag: prop_element}}. Further parsing of the prop_element has to be done by the caller.

self.sync_token will be populated if found, self.objects will be populated.

validate_status (*status*)
status is a string like "HTTP/1.1 404 Not Found". 200, 207 and 404 are considered good statuses. The SOGo caldav server even returns "201 created" when doing a sync-report, to indicate that a resource was created after the last sync-token. This makes sense to me, but I've only seen it from SOGo, and it's not in accordance with the examples in rfc6578.

1.2 `objects` – Object definitions

A "DAV object" is anything we get from the caldav server or push into the caldav server, notably principal, calendars and calendar events.

(This file has become huge and will be split up prior to the next release. I think it makes sense moving the CalendarObjectResource class hierarchy into a separate file)

class caldav.objects.**Calendar** (*client=None*, *url=None*, *parent=None*, *name=None*, *id=None*, *props=None*, ***extra*)
The *Calendar* object is used to represent a calendar collection. Refer to the RFC for details: https://tools.ietf.org/html/rfc4791#section-5.3.1

`add_event` (*ical*, *no_overwrite=False*, *no_create=False*)
Add a new event to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

`add_journal` (*ical*, *no_overwrite=False*, *no_create=False*)
Add a new journal entry to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

`add_todo` (*ical*, *no_overwrite=False*, *no_create=False*)
Add a new task to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

`build_date_search_query` (*start*, *end=None*, *compfilter='VEVENT'*, *expand='maybe'*)
Split out from the date_search-method below. The idea is that maybe the generated query can be amended, i.e. to filter out by category etc. To be followed up in https://github.com/python-caldav/caldav/issues/16

`calendar_multiget` (*event_urls*)
get multiple events' data @author mtorange@gmail.com @type events list of Event

`date_search` (*start*, *end=None*, *compfilter='VEVENT'*, *expand='maybe'*)
Search events by date in the calendar. Recurring events are expanded if they are occuring during the specified time frame and if an end timestamp is given.

Parameters:

- start = datetime.today().
- end = same as above.
- compfilter = defaults to events only. Set to None to fetch all calendar components.
- expand - should recurrent events be expanded? (to preserve backward-compatibility the default "maybe" will be changed into True unless the date_search is open-ended)

Returns:

- [CalendarObjectResource(), . . .]

`event_by_url` (*href*, *data=None*)
Returns the event with the given URL

`events` ()
List all events from the calendar.

Returns:

- [Event(), . . .]

`freebusy_request` (*start*, *end*)
Search the calendar, but return only the free/busy information.

Parameters:

- start = datetime.today().
- end = same as above.

Returns:

- [FreeBusy(), . . .]

`get_supported_components``()`
returns a list of component types supported by the calendar, in string format (typically ['VJOURNAL', 'VTODO', 'VEVENT'])

`journals``()`
List all journals from the calendar.

Returns:

- [Journal(), . . .]

`object_by_uid``(`*uid*, *comp_filter=None*`)`
Get one event from the calendar.

Parameters:

- uid: the event uid

Returns:

- Event() or None

`objects``(`*sync_token=None*, *load_objects=False*`)`
objects_by_sync_token aka objects

Do a sync-collection report, ref RFC 6578 and https://github.com/python-caldav/caldav/issues/87

This method will return all objects in the calendar if no sync_token is passed (the method should then be referred to as "objects"), or if the sync_token is unknown to the server. If a sync-token known by the server is passed, it will return objects that are added, deleted or modified since last time the sync-token was set.

If load_objects is set to True, the objects will be loaded - otherwise empty CalendarObjectResource objects will be returned.

This method will return a SynchronizableCalendarObjectCollection object, which is an iterable.

`objects_by_sync_token``(`*sync_token=None*, *load_objects=False*`)`
objects_by_sync_token aka objects

Do a sync-collection report, ref RFC 6578 and https://github.com/python-caldav/caldav/issues/87

This method will return all objects in the calendar if no sync_token is passed (the method should then be referred to as "objects"), or if the sync_token is unknown to the server. If a sync-token known by the server is passed, it will return objects that are added, deleted or modified since last time the sync-token was set.

If load_objects is set to True, the objects will be loaded - otherwise empty CalendarObjectResource objects will be returned.

This method will return a SynchronizableCalendarObjectCollection object, which is an iterable.

`save``()`
The save method for a calendar is only used to create it, for now. We know we have to create it when we don't have a url.

Returns:

- self

`save_event``(`*ical*, *no_overwrite=False*, *no_create=False*`)`
Add a new event to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

save_journal (*ical*, *no_overwrite=False*, *no_create=False*)
Add a new journal entry to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

save_todo (*ical*, *no_overwrite=False*, *no_create=False*)
Add a new task to the calendar, with the given ical.

Parameters:

- ical - ical object (text)

save_with_invites (*ical*, *attendees*, ***attendeeoptions*)
sends a schedule request to the server. Equivalent with save_event, save_todo, etc, but the attendees will be added to the ical object before sending it to the server.

search (*xml*, *comp_class=None*)
This method was partly written to approach https://github.com/python-caldav/caldav/issues/16 This is a result of some code refactoring, and after the next round of refactoring we've ended up with this:

todos (*sort_keys=('due', 'priority')*, *include_completed=False*, *sort_key=None*)
fetches a list of todo events.

Parameters:

- sort_keys: use this field in the VTODO for sorting (iterable of lower case string, i.e. ('priority','due')).
- include_completed: boolean - by default, only pending tasks are listed
- sort_key: DEPRECATED, for backwards compatibility with version 0.4.

class caldav.objects.**CalendarObjectResource** (*client=None*, *url=None*, *data=None*, *parent=None*, *id=None*, *props=None*)
Ref RFC 4791, section 4.1, a "Calendar Object Resource" can be an event, a todo-item, a journal entry, or a free/busy entry

add_attendee (*attendee*, *no_default_parameters=False*, ***parameters*)
For the current (event/todo/journal), add an attendee.

The attendee can be any of the following: * A principal * An email address prepended with "mailto:" * An email address without the "mailto:"-prefix * A two-item tuple containing a common name and an email address * (not supported, but planned: an ical text line starting with the word "ATTENDEE")

Any number of attendee parameters can be given, those will be used as defaults unless no_default_parameters is set to True:

partstat=NEEDS-ACTION cutype=UNKNOWN (unless a principal object is given) rsvp=TRUE role=REQ-PARTICIPANT schedule-agent is not set

add_organizer ()
goes via self.client, finds the principal, figures out the right attendee-format and adds an organizer line to the event

copy (*keep_uid=False*, *new_parent=None*)
Events, todos etc can be copied within the same calendar, to another calendar or even to another caldav server

data
vCal representation of the object

icalendar_instance
icalendar instance of the object

instance
vobject instance of the object

load()
Load the object from the caldav server.

save(*no_overwrite=False*, *no_create=False*, *obj_type=None*, *if_schedule_tag_match=False*)
Save the object, can be used for creation and update.

no_overwrite and no_create will check if the object exists. Those two are mutually exclusive. Some servers don't support searching for an object uid without explicitly specifying what kind of object it should be, hence obj_type can be passed. obj_type is only used in conjunction with no_overwrite and no_create.

Returns:

- self

vobject_instance
vobject instance of the object

class caldav.objects.**CalendarSet**(*client=None*, *url=None*, *parent=None*, *name=None*, *id=None*, *props=None*, ***extra*)
A CalendarSet is a set of calendars.

calendar(*name=None*, *cal_id=None*)
The calendar method will return a calendar object. If it gets a cal_id but no name, it will not initiate any communication with the server

Parameters:

- name: return the calendar with this name
- cal_id: return the calendar with this calendar id or URL

Returns:

- Calendar(…)-object

calendars()
List all calendar collections in this set.

Returns:

- [Calendar(), …]

make_calendar(*name=None*, *cal_id=None*, *supported_calendar_component_set=None*)
Utility method for creating a new calendar.

Parameters:

- name: the name of the new calendar
- cal_id: the uuid of the new calendar
- supported_calendar_component_set: what kind of objects (EVENT, VTODO, VFREEBUSY, VJOURNAL) the calendar should handle. Should be set to ['VTODO'] when creating a task list in Zimbra - in most other cases the default will be OK.

Returns:

- Calendar(…)-object

class caldav.objects.**DAVObject**(*client=None*, *url=None*, *parent=None*, *name=None*, *id=None*, *props=None*, ***extra*)
Base class for all DAV objects. Can be instantiated by a client and an absolute or relative URL, or from the parent object.

children(*type=None*)
List children, using a propfind (resourcetype) on the parent object, at depth = 1.

delete()
Delete the object.

get_properties(*props=None*, *depth=0*, *parse_response_xml=True*, *parse_props=True*)
Get properties (PROPFIND) for this object.

With parse_response_xml and parse_props set to True a best-attempt will be done on decoding the XML we get from the server - but this works only for properties that don't have complex types. With parse_response_xml set to False, a DAVResponse object will be returned, and it's up to the caller to decode. With parse_props set to false but parse_response_xml set to true, xml elements will be returned rather than values.

Parameters:

- props = [dav.ResourceType(), dav.DisplayName(), . . .]

Returns:

- {proptag: value, . . . }

save()
Save the object. This is an abstract method, that all classes derived from DAVObject implement.

Returns:

- self

set_properties(*props=None*)
Set properties (PROPPATCH) for this object.

- props = [dav.DisplayName('name'), . . .]

Returns:

- self

class caldav.objects.**Event**(*client=None*, *url=None*, *data=None*, *parent=None*, *id=None*, *props=None*)
The *Event* object is used to represent an event (VEVENT).

As of 2020-12 it adds nothing to the inheritated class. (I have frequently asked myself if we need those subclasses . . . perhaps not)

class caldav.objects.**FreeBusy**(*parent*, *data*, *url=None*, *id=None*)
The *FreeBusy* object is used to represent a freebusy response from the server. __init__ is overridden, as a FreeBusy response has no URL or ID. The inheritated methods .save and .load is moot and will probably throw errors (perhaps the class hierarchy should be rethought, to prevent the FreeBusy from inheritating moot methods)

Update: With RFC6638 a freebusy object can have an URL and an ID.

class caldav.objects.**Journal**(*client=None*, *url=None*, *data=None*, *parent=None*, *id=None*, *props=None*)
The *Journal* object is used to represent a journal entry (VJOURNAL).

As of 2020-12 it adds nothing to the inheritated class. (I have frequently asked myself if we need those subclasses . . . perhaps not)

class caldav.objects.**Principal**(*client=None*, *url=None*)
This class represents a DAV Principal. It doesn't do much, except keep track of the URLs for the calendar-home-set, etc.

A principal MUST have a non-empty DAV:displayname property (defined in Section 13.2 of [RFC2518]), and a DAV:resourcetype property (defined in Section 13.9 of [RFC2518]). Additionally, a principal MUST report the DAV:principal XML element in the value of the DAV:resourcetype property.

(TODO: the resourcetype is actually never checked, and the DisplayName is not stored anywhere)

calendar(*name=None*, *cal_id=None*)
The calendar method will return a calendar object. It will not initiate any communication with the server.

calendar_user_address_set()
defined in RFC6638

calendars()
Return the principials calendars

get_vcal_address()
Returns the principal, as an icalendar.vCalAddress object

make_calendar(*name=None*, *cal_id=None*, *supported_calendar_component_set=None*)
Convenience method, bypasses the self.calendar_home_set object. See CalendarSet.make_calendar for details.

class caldav.objects.**ScheduleInbox**(*client=None*, *principal=None*, *url=None*)

class caldav.objects.**ScheduleMailbox**(*client=None*, *principal=None*, *url=None*)
RFC6638 defines an inbox and an outbox for handling event scheduling.

TODO: As ScheduleMailboxes works a bit like calendars, I've chosen to inheritate the Calendar class, but this is a bit incorrect, a ScheduleMailbox is a collection, but not really a calendar. We should create a common base class for ScheduleMailbox and Calendar eventually.

get_items()
TODO: work in progress TODO: perhaps this belongs to the super class?

class caldav.objects.**ScheduleOutbox**(*client=None*, *principal=None*, *url=None*)

class caldav.objects.**SynchronizableCalendarObjectCollection**(*calendar*, *objects*, *sync_token*)
This class may hold a cached snapshot of a calendar, and changes in the calendar can easily be copied over through the sync method.

To create a SynchronizableCalendarObjectCollection object, use calendar.objects(load_objects=True)

objects_by_url()
returns a dict of the contents of the SynchronizableCalendarObjectCollection, URLs -> objects.

sync()
This method will contact the caldav server, request all changes from it, and sync up the collection

class caldav.objects.**Todo**(*client=None*, *url=None*, *data=None*, *parent=None*, *id=None*, *props=None*)
The *Todo* object is used to represent a todo item (VTODO). A Todo-object can be completed.

complete(*completion_timestamp=None*)
Marks the task as completed.

This method probably will do the wrong thing if the task is a recurring task, in version 1.0 this will likely be changed - see https://github.com/python-caldav/caldav/issues/127 for details.

Parameters:

- completion_timestamp - datetime object. Defaults to datetime.now().

caldav.objects.**errmsg**(*r*)
Utility for formatting a response xml tree to an error string

CHAPTER 2

Project home

The project currently lives on github, https://github.com/python-caldav/caldav - if you have problems using the library (including problems understanding the documentation), please feel free to report it on the issue tracker there.

CHAPTER 3

Objective and scope

The python caldav library should make interactions with calendar servers simple and easy. Simple operations (like find a list of all calendars owned, inserting an icalendar object into a calendar, do a simple date search, etc) should be trivial to accomplish even if the end-user of the library has no or very little knowledge of the caldav, webdav or icalendar standards. The library should be agile enough to allow "power users" to do more advanced stuff.

3.1 RFC 4791, 2518, 5545, 6638 et al

RFC 4791 (CalDAV) outlines the standard way of communicating with a calendar server. RFC 4791 is an extension of RFC 4918 (WebDAV). The scope of this library is basically to cover RFC 4791/4918, the actual communication with the caldav server. (The WebDAV standard also has quite some extensions, this library supports some of the relevant extensions as well).

There exists another library webdavclient3 for handling RFC 4918 (WebDAV), ideally we should be depending on it rather than overlap it.

RFC 6638/RFC 6047 is extending the CalDAV and iCalendar protocols for scheduling purposes, work is in progress to support RFC 6638. Support for RFC 6047 is considered mostly outside the scope of this library, though for convenience this library may contain methods like accept() on a calendar invite (which involves fetching the invite from the server, editing the calendar data and putting it to the server).

This library should make it trivial to fetch an event, modify the date and save it back to the server - but to do that it's also needed to support RFC 5545 (icalendar). It's outside the scope of this library to implement logic for parsing and modifying RFC 5545, instead we depend on another library for that.

There exists two libraries supporting RFC 5545, vobject and icalendar. The icalendar library seems to be more popular. Version 0.x depends on vobject, version 1.x will depend on icalendar. Version 0.7 and higher supports both, but the "alternative" library will only be loaded when/if needed, and the vobject support may be deprecated in the future.

3.2 Misbehaving server implementations

Some server implementations may have some "caldav"-support that either doesn't implement all of RFC 4791, breaks the standard a bit, or has extra features. As long as it doesn't add too much complexity to the code, hacks and workarounds for "badly behaving caldav servers" are considered to be within the scope. Ideally, users of the caldav library should be able to download all the data from one calendar server or cloud provider, upload it to another server type or cloud provider, and continue using the library without noticing any differences. To get there, it may be needed to add tweaks in the library covering the things the servers are doing wrong.

There exists an extention to the standard covering calendar color and calendar order, allegedly with an xml namespace `http://apple.com/ns/ical/`. That URL gives (301 https and then) 404. I've so far found no documentation at all on this extension - however, it seems to be supported by several caldav libraries, clients and servers. As of 0.7, calendar colors and order is available for "power users".

CHAPTER 4

Quickstart

All code examples below are snippets from the basic_usage_examples.py.

Setting up a caldav client object and a principal object:

```
client = caldav.DAVClient(url=url, username=username, password=password)
my_principal = client.principal()
```

Fetching calendars:

```
calendars = my_principal.calendars()
```

Creating a calendar:

```
my_new_calendar = my_principal.make_calendar(name="Test calendar")
```

Adding an event to the calendar:

```
my_event = my_new_calendar.save_event("""BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//Example Corp.//CalDAV Client//EN
BEGIN:VEVENT
UID:20200516T060000Z-123401@example.com
DTSTAMP:20200516T060000Z
DTSTART:20200517T060000Z
DTEND:20200517T230000Z
RRULE:FREQ=YEARLY
SUMMARY:Do the needful
END:VEVENT
END:VCALENDAR
""")
```

Do a date search in a calendar:

```
events_fetched = my_new_calendar.date_search(
    start=datetime(2021, 1, 1), end=datetime(2024, 1, 1), expand=True)
```

To modify an event:

event.vobject_instance.vevent.summary.value = 'Norwegian national day celebrations' event.save()

```
event.icalendar_instance
```

is also supported.

Find a calendar with a known URL without going through the Principal-object:

```
the_same_calendar = client.calendar(url=my_new_calendar.url)
```

Get all events from a calendar:

```
all_events = the_same_calendar.events()
```

Deleting a calendar (or, basically, any object):

```
my_new_calendar.delete()
```

Create a task list:

```
my_new_tasklist = my_principal.make_calendar(
            name="Test tasklist", supported_calendar_component_set=['VTODO'])
```

Adding a task to a task list:

```
my_new_tasklist.add_todo("""BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//Example Corp.//CalDAV Client//EN
BEGIN:VTODO
UID:20070313T123432Z-456553@example.com
DTSTAMP:20070313T123432Z
DTSTART;VALUE=DATE:20200401
DUE;VALUE=DATE:20200501
RRULE:FREQ=YEARLY
SUMMARY:Deliver some data to the Tax authorities
CATEGORIES:FAMILY,FINANCE
STATUS:NEEDS-ACTION
END:VTODO
END:VCALENDAR""")
```

Fetching tasks:

```
todos = my_new_tasklist.todos()
```

Date_search also works on task lists, but one has to be explicit to get the tasks:

```
todos = my_new_calendar.date_search(
    start=datetime(2021, 1, 1), end=datetime(2024, 1, 1),
    compfilter='VTODO', expand=True)
```

Mark a task as completed:

```
todos[0].complete()
```

CHAPTER 5

More examples

Check the examples folder, particularly basic examples. There is also a scheduling examples for sending, receiving and replying to invites, though this is not very well-tested so far. The test code also covers lots of stuff, though it's not much optimized for readability (at least not as of 2020-05). Tobias Brox is also working on a command line interface built around the caldav library.

CHAPTER 6

Notable classes and workflow

- You'd always start by initiating a *caldav.davclient.DAVClient* object, this object holds the authentication details for the server.
- From the client object one can get hold of a *caldav.objects.Principal* object representing the logged-in principal.
- From the principal object one can fetch / generate *caldav.objects.Calendar* objects.
- From the calendar object one can fetch / generate *caldav.objects.Event* objects and *caldav.objects.Todo* objects (as well as *caldav.objects.Journal* objects - does anyone use Journal objects?). Eventually the library may also spew out objects of the base class (*caldav.objects.CalendarObjectResource*) if the object type is unknown when the object is instantiated.
- If one happens to know the URLs, objects like calendars, principals and events can be instantiated without going through the Principal-object of the logged-in user. A path, relative URL or full URL should work, but the URL should be without authentication details.

For convenience, the classes above are also available as `caldav.DAVClient`, `caldav.Principal`, `caldav.Calendar`, `caldav.Event`, `caldav.Todo` etc.

CHAPTER 7

Compatibility

(This will probably never be completely up-to-date. CalDAV-servers tend to be a moving target, and I rarely recheck if things works in newer versions of the software after I find an incompatibility)

The test suite is regularly run against several calendar servers, see https://github.com/python-caldav/caldav/issues/45 for the latest updates. See `tests/compatibility_issues.py` for the most up-to-date list of compatibility issues. In early versions of this library test breakages was often an indication that the library did not conform well enough to the standards, but as of today it mostly indicates that the servers does not support the standard well enough. It may be an option to add tweaks to the library code to cover some of the missing functionality.

Here are some known issues:

- iCloud, Google and Zimbra are notoriously bad on their CalDAV-support.
- You may want to avoid non-ASCII characters in the calendar name, or some servers (at least Zimbra) may behave a bit unexpectedly.
- It's non-trivial to fix proper support for recurring events and tasks on the server side. DAViCal and Baikal are the only one I know of that does it right, all other calendar implementations that I've tested fails (but in different ways) on the tests covering recurrent events and tasks. Xandikos developer claims that it should work, I should probably revisit it again.
- Baikal does not support date search for todo tasks. DAViCal has slightly broken support for such date search.
- There are some special hacks both in the code and the tests to work around compatibility issues in Zimbra (this should be solved differently)
- Not all servers supports task lists, not all servers supports freebusy, and not all servers supports journals. Xandikos and Baikal seems to support them all.
- Calendar creation is actually not a mandatory feature according to the RFC, but the tests depends on it. The google calendar does support creating calendars, but not through their CalDAV adapter.
- iCloud may be a bit tricky, this is tracked in issue https://github.com/python-caldav/caldav/issues/3 - the list of incompatibilities found includes:
  - No support for freebusy-requests, tasks or journals (only support for basic events).
  - Broken (or no) support for recurring events

  - We’ve observed information reappearing even if it has been deleted (i.e. recreating a calendar with the same name as a deleted calendar, and finding that the old events are still there)
  - Seems impossible to have the same event on two calendars
  - Some problems observed with the propfind method
  - object_by_uid does not work (and my object_by_uid follows the example in the RFC)
- Google seems to be the new Microsoft, according to the issue tracker it seems like their CalDAV-support is rather lacking. At least they have a list … https://developers.google.com/calendar/caldav/v2/guide
- radicale will auto-create a calendar if one tries to access a calendar that does not exist. The normal method of accessing a list of the calendars owned by the user seems to fail.

CHAPTER 8

Some notes on Caldav URLs

CalDAV URLs can be quite confusing, some software requires the URL to the calendar, other requires the URL to the principal. The Python CalDAV library does support accessing calendars and principals using such URLs, but the recommended practice is to configure up the CalDAV root URL and tell the library to find the principal and calendars from that. Typical examples of CalDAV URLs:

- iCloud: `https://caldav.icloud.com/`. Note that there is no template for finding the calendar URL and principal URL for iCloud - such URLs contains some ID numbers, by simply sticking to the recommended practice the caldav library will find those URLs. A typical icloud calendar URL looks like `https://p12-caldav.icloud.com/12345/calendars/CALNAME`.
- Google: `https://www.google.com/calendar/dav/` - but this is a legacy URL, before using the officially supported URL https://github.com/python-caldav/caldav/issues/119 has to be resolved. There are some details on the new CalDAV endpoints at https://developers.google.com/calendar/caldav/v2/guide. The legacy calendar URL for the primary personal calendar seems to be of the format `https://www.google.com/calendar/dav/donald%40gmail.com/events`. When creating new calendars, they seem to end up under a global namespace.
- DAViCal: The caldav URL typically seems to be on the format `https://your.server.example.com/caldav.php/`, though it depends on how the web server is configured. The primary calendars have URLs like `https://your.server.example.com/caldav.php/donald/calendar` and other calendars have names like `https://your.server.example.com/caldav.php/donald/golfing_calendar`.
- Zimbra: The caldav URL is typically on the format `https://mail.example.com/dav/`, calendar URLs can be on the format `https://mail.example.com/dav/donald@example.com/My%20Golfing%20Calendar`. Display name always matches the last part of the URL.

CHAPTER 9

Unit testing

To start the tests code, install everything from the setup.tests_requires list and run:

```
$ python setup.py nosetests
```

(tox should also work, but it may be needed to look more into it)

It will run some unit tests and some functional tests. You may want to add your own private servers into tests/conf_private.py, see tests/conf_private.py.EXAMPLE

CHAPTER 10

Documentation

To build the documentation, install sphinx and run:

```
$ python setup.py build_sphinx
```

CHAPTER 11

License

Caldav is dual-licensed under the GNU GENERAL PUBLIC LICENSE Version 3 and the Apache License 2.0.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index