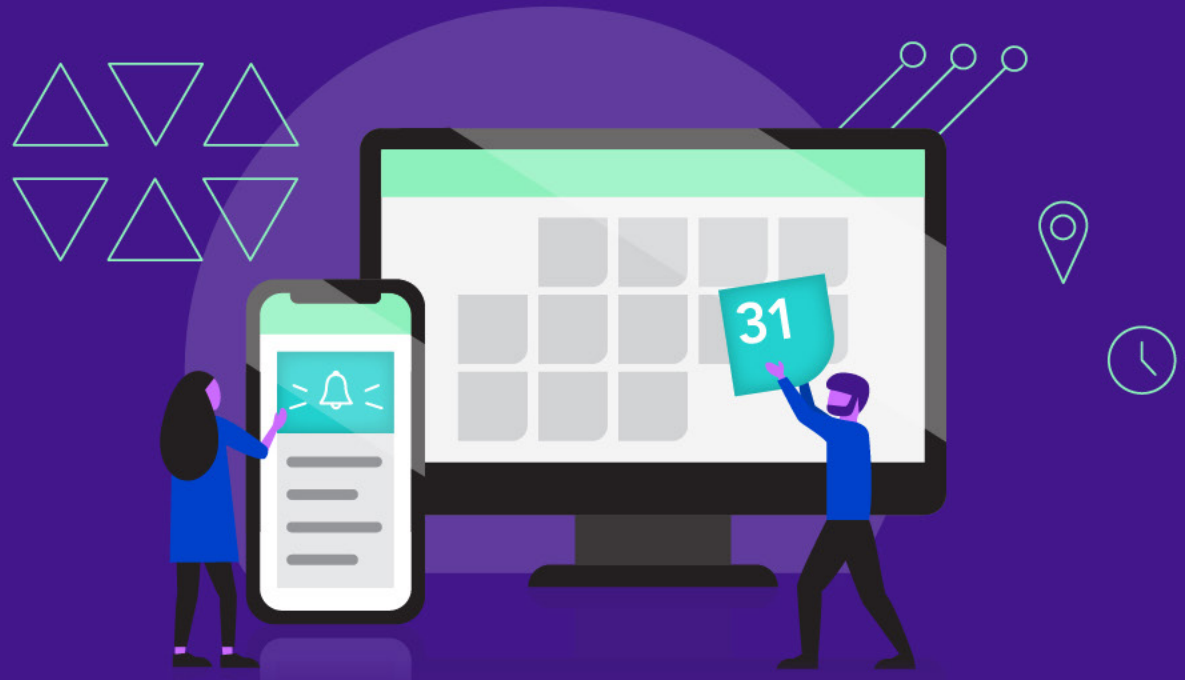
Nylas

How to Integrate Calendars Into Your App

Learn how to avoid common pitfalls
of the integration and set yourself up for scale.



2019

About Nylas

Nylas is a developer platform that powers applications with email, calendar, and contacts integrations through a REST API. The Nylas API handles more than 100 million API requests per day and has synced more than 15 billion emails. 22,000 developers are signed up to use the API. We're excited to share some of our learnings here.

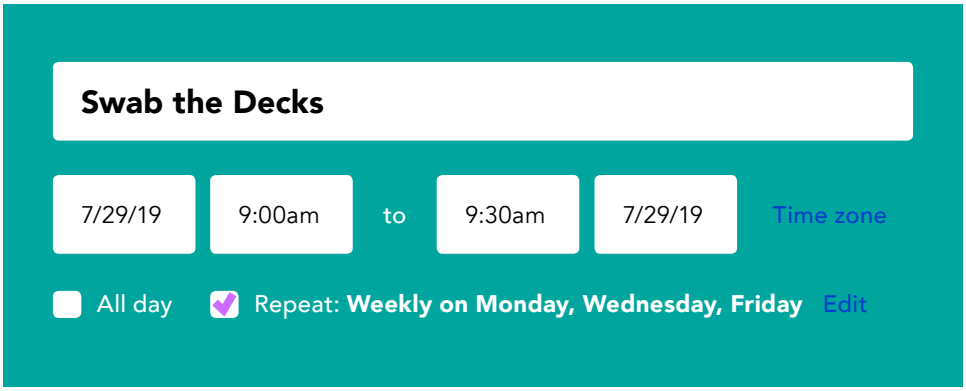
Overview

Our calendars help us keep track of our lives from the most mundane to the most significant moments: dentist appointments, car repairs, baby showers, surprise birthday parties, business meetings, flight details - you name it. Integrating these calendars into the software tools businesses and consumers use has become mission-critical for many software companies — but (ask any developer), building these integrations is a complex process that can take months. Google, Outlook, Microsoft Exchange, Yahoo, and the rest have their own API quirks and process data in different ways.

We created this guide to help make your calendar integration process easy. In this guide, you will learn:

The Deceptively Complex World of Calendar Events and RRULEs

Daily meetings, birthdays, chores, and personal reminders. These are all common types of calendar events which repeat on a set schedule, and modern calendar applications easily support creating them.



Calendar interface for creating recurring events.

But underneath this simple "Repeat" checkbox is a surprising amount of complexity resulting from years of legacy standards with backward compatibility. What happens when an RFC meets the real world?

Repeat as necessary.

Working with repeating events is important for a few different kinds of apps. If you're building a calendar UI, you want to make sure the events you show match what the user sees at the source. However, any kind of scheduling app needs to display events in order to accurately show a user's availability.

There are two ways to work with repeating events:

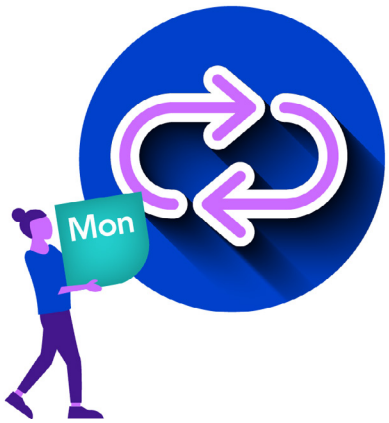
1. Take a single event and the information for it repeats, and generate all the occurrences of that event, or;

2. Use an API that expands the occurrences for you, and treat them more like standalone events.

Let's start from the ground up — a single event which repeats.

The RRULE.

The key to any repeating event is the recurrence rule, a way of describing how that event repeats. These are also referred to as RRULEs.



Recurrence rules are primarily defined in RFC 2445, section 4.8.5.4, which also describes the full "iCalendar" spec for `.ics` files. Calendar providers like iCloud and Google Calendar provide downloads of these files for apps.

The RRULE format encapsulates a repeating pattern, such as "every Thursday". Combined with the event's starting time, you can figure out exactly when each future occurrence of the event should begin. Note that the RRULE itself doesn't encode the starting times.

A simple RRULE for an event which repeats every day looks like this:

```
RRULE:FREQ=DAILY
```

The RRULE syntax can also specify a total number of instances, or an end time:

```
RRULE:FREQ=DAILY;COUNT=10;
RRULE:FREQ=DAILY;UNTIL=20150919T063000Z
```

You can choose one or more days of the week to repeat on, and even alternate between specific days:

```
RRULE:FREQ=WEEKLY;BYDAY=TH                # every Thursday
RRULE:FREQ=WEEKLY;BYDAY=MO,WE,FR          # every Mon, Wed and Fri
RRULE:FREQ=WEEKLY;BYDAY=TU;INTERVAL=2     # every other Tuesday
```

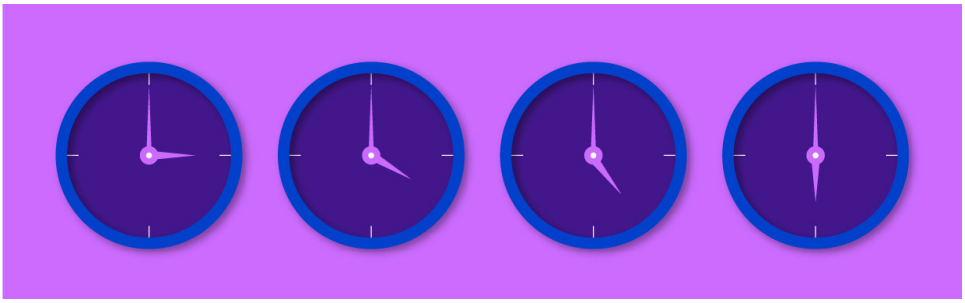
RRULE syntax goes far beyond these simple examples, including support for day of month (e.g. the third Thursday in November), week numbers, repeating on the same numerical day of a month, and plenty more. If you want to experiment more with specifying RRULEs, the rrule.js demo is a superb place to do so.

The `python-dateutil` module in Python has a parser which makes it easier to work with RRULEs:

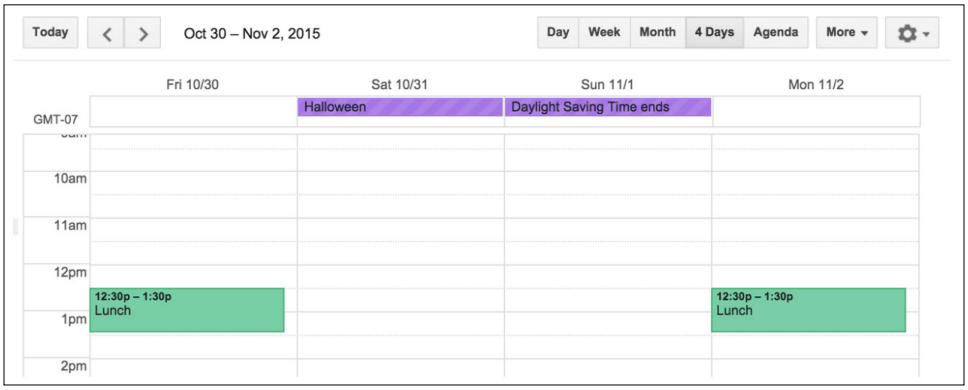
```
from dateutil.rrule import rrulestr
from datetime import datetime

rule_string = “RRULE:FREQ=WEEKLY;BYDAY=TH”

# Use rrulestr to parse a RFC-formatted string
# Without a start time, it assumes the rule starts from now.
rule = rrulestr(rule_string)

# Get the next occurrence
rule.after(datetime.now())

# Get all the occurrences in December
december_1 = datetime.now().replace(month=12, day=1)
december_31 = december_1.replace(day=31)
rule.between(after=december_1, before=december_31)
```

Given a calendar event with an RRULE property, you can figure out all the times that event actually happens. This is usually fairly straightforward, but what happens when the clocks go backward?

Building Timezone-Proof Calendar Integrations



When a timezone transitions into or out of daylight savings, repeating events are expected to remain at the same local time. For example, lunch is always scheduled for 12:30, even if the underlying UTC time is an hour earlier or later, as Google Calendar shows here:



Google Calendar interface with lunch always scheduled at the same time.

This can cause its share of headaches, especially when you represent datetimes globally in UTC. One alternative way to implement this when using `dateutil.rrule` is to normalize with an event's timezone throughout, which ensures that daylight savings is accounted for when we convert the final event times back to UTC.

Here's an example where we expand the recurrence rule for an event that spans a DST change (in this case, the switch from PDT to PST on 11/1/15):

```
from datetime import datetime

from dateutil.rrule import rrulestr
from dateutil.tz import gettz

rule_string = "RRULE:FREQ=WEEKLY;BYDAY=MO,TU,WE,TH,FR"

start = datetime(2015, 7, 6, 12, 30).replace(tzinfo=gettz('US/
Pacific'))
# start = arrow.get(2015,07,06,12,30,00,0,)
rule = rrulestr(rule_string, dtstart=start)

# When expanding the rule, we get 12:30pm US/Pacific
times = rule.between(
    after=datetime(2015,10,30,00,00,01,0).replace(tzinfo=gettz('US/
    Pacific')),
    before=datetime(2015,11,2,23,59,59,0).replace(tzinfo=gettz('US/
    Pacific')),
    inc=True)  # List[datetime]

print(times)
# When converted, 12:30pm on 10/30 becomes 19:30 UTC, and
# 12:30pm on 11/2 becomes 20:30 UTC due to the daylight change
on 11/1.
print([t.astimezone(gettz('UTC')) for t in times])
```

Exceptions to the rule.

Given an RRULE, you can figure out when a specific repeating event is going to occur. But what about one-off changes to the event? This happens often when repeating meetings are moved for one day, or their agenda/location is changed, or they are canceled altogether.

Cancellations.

Cancellations to a specific repeating instance are fairly straightforward: the iCalendar spec includes support for exception dates when repeating events do not occur on a specific cycle. For example, you may cancel a daily meeting on Christmas Day. These exceptions are expressed in the EXDATE field:

```
RRULE:FREQ=DAILY
EXDATE:20151225T173000Z
```

You'll notice the EXDATE is in fact a datetime (not just a date) represented in ISO 8601. When dealing with repeated events, this means you need to keep careful track of the start time of the original event, and use that to determine at what time the event should repeat. An easier way to identify these individual repetitions is by their full UTC datetime. You can also conveniently use the same identifier when specifying repetitions which don't exist.

In `dateutil`, to expand a recurrence rule with an EXDATE we need to convert our singular `rrule` into a `rruleset`:

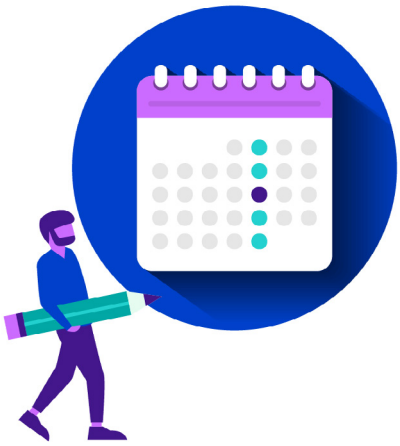
```
from datetime import datetime
from dateutil.rrule import rruleset

# Create a daily recurrence starting on 12/20 at 17:30
daily = rrulestr("RRULE:FREQ=DAILY",
                 dtstart=datetime(2015,12,20,17,30,00))
rules = rruleset()
rules.rrule(daily)       # Add the daily RRULE to the set

# Exclude 12/25 at 17:30
excl_date = datetime(2015,12,25,17,30,00)
rules.exdate(excl_date)  # Add the excluded date to the set

rules.between(datetime(2015,12,24), datetime(2015,12,27))

# >>> [datetime.datetime(2015, 12, 24, 17, 30),
#      datetime.datetime(2015, 12, 26, 17, 30)]
```

You may have noticed that the `rruleset.exdate` method takes a `datetime` instance rather than an EXDATE string. This is a bit annoying, and means you'll need to parse the EXDATE string into datetimes yourself. Here's an example of how to do that.

```
from datetime import datetime

from dateutil.tz import gettz
from pytz import all_timezones, UTC

def parse_rrule_datetime(datetime_str, tzinfo=None):
```

```
    # format: 20140904T133000Z (datetimes) or 20140904 (dates)
    if datetime_str[-1] == 'Z':
        tzinfo = 'UTC'
        datetime_str = datetime_str[:-1]
    if len(datetime_str) == 8:
        dt = datetime.strptime(datetime_str, '%Y%m%d').
        replace(tzinfo=UTC)
    else:
        dt = datetime.strptime(datetime_str, '%Y%m%dT%H%M%S')
    if tzinfo and tzinfo != 'UTC':
        if tzinfo not in all_timezones:
            raise ValueError('Unknown timezone.')
        dt = dt.replace(tzinfo=gettz(tzinfo))
    return dt

def parse_exdate(exdate):
    # Parse the EXDATE string and return a list of timezone-
    aware datetimes
    if not exdate:
        raise ValueError('Invalid exdate.')

    excl_dates = []
    name, values = exdate.split(':', 1)
    tzinfo = 'UTC'
    for p in name.split(';'):
        # Handle TZID in EXDATE
        if p.startswith('TZID'):
           tzinfo = p[5:]
    for v in values.split(','):
        if not v:
            raise ValueError('Invalid date found in exdate: ' + v)

        # convert to timezone-aware dates
        t = parse_rrule_datetime(v, tzinfo)
        excl_dates.append(t)

    return excl_dates
```

Modifying events.



When a change is made to a specific instance of a repeating event, you will have to move out of RFC territory and into something more like a Calendar Wild West. The seemingly logical thing to do is to cancel the instance (using EXDATE) and create a brand new one-off event with the changed information.

From the point of view of the original event, this looks identical to a real cancellation. (In the following example, fields are cherry-picked from the full event.)

Original event:

```
BEGIN:VEVENT
RRULE:FREQ=DAILY;COUNT=5
SUMMARY:Treasure Hunting
DTSTART;TZID=America/Los_Angeles:20150706T120000
DTEND;TZID=America/Los_Angeles:20150706T130000
END:VEVENT
```

With one event in the series modified:

```
BEGIN:VEVENT
RRULE:FREQ=DAILY;COUNT=5
EXDATE;TZID=America/Los_Angeles:20150707T120000
SUMMARY:Treasure Hunting
DTSTART;TZID=America/Los_Angeles:20150706T120000
DTEND;TZID=America/Los_Angeles:20150706T130000
END:VEVENT

BEGIN:VEVENT
SUMMARY:Treasure Hunting
LOCATION:The other island
DTSTART;TZID=America/Los_Angeles:20150707T120000
DTEND;TZID=America/Los_Angeles:20150707T130000
END:VEVENT
```

By disconnecting the modified event from its parent series, you can run into a misleading situation. It looks like the parent isn't repeating on that specific day,

but it actually still is! If you delete or change the parent event, the modified exception event will stick around regardless.

Instead, the prevailing approach is to add metadata to the modified event that points back at its parent, and not update the EXDATE:

```
BEGIN:VEVENT
UID:0000001
RRULE:FREQ=DAILY;COUNT=5
SUMMARY:Treasure Hunting
DTSTART;TZID=America/Los_Angeles:20150706T120000
DTEND;TZID=America/Los_Angeles:20150706T130000
END:VEVENT

BEGIN:VEVENT
UID:0000001
SUMMARY:Treasure Hunting
LOCATION:The other island
DTSTART;TZID=America/Los_Angeles:20150707T120000
DTEND;TZID=America/Los_Angeles:20150707T130000
RECURRENCE-ID;TZID=America/Los_Angeles:20150707T120000
END:VEVENT
```

If the `RECURRENCE-ID` is the original start time of the modified event, and the `UID` on both events is the same, you can connect the dots and figure out that the exception event replaces an instance in the series which was originally to occur at that time. Let's look at this in practice with an example that works directly with the Google Calendar API.

Recurring events for Google Calendar.

The Google Calendar docs say that recurrence information for an event is available via the `recurrence` field. This contains the RRULE and other recurrence information for an event (in practice, almost always just the RRULE).

Unfortunately, this isn't sufficient to figure out exactly what's going on with a repeating event due to cancellations and exceptions.

Google Calendar: recurring event cancellations.

Google Calendar exposes canceled events as separate, individual events alongside

the original repeating events. By default, the API hides cancellations, but this can be disabled by including `showDeleted=True` as a URL parameter. This is by design because the Google Calendar API **does not update the EXDATE field** when an event is canceled.

A canceled event is returned here as an abbreviated event object, without fields such as the title and location:

```
{
  "id": "uid1234_20150707T150000Z",
  "status": "cancelled",
  "recurringEventId": "uid1234",
  "originalStartTime": {
    "dateTime": "2015-07-07T08:00:00-07:00"
  }
}
```

There are several clues to connect this back to the parent event:

- `recurringEventId` is actually the parent event `id`
- `originalStartTime` is the originally scheduled start time for this instance
- the event's `id` is a combination of these two, with the time in UTC

Modifications.

Modifications to recurring events via the Google Calendar API look very similar to cancellations, but contain the full event information (title, location, etc). Again, the EXDATE does not change.

Expanding RRULEs with Google Calendar.

In order to find all the occurrences of a repeating event, including cancellations and one-off modifications, we must expand the "master" RRULE and iterate through "child" events which are linked back to the master via their IDs. In addition, keeping track of the event timezone is critical when attempting to match a child event based purely on the intended original start date, particularly as repeating events cross daylight savings boundaries. Google Calendar provides timezones in `start`, `end`, and `originalStartTime` properties.

One major downside of working with events this way is that a seemingly-simple query like "get all events on my calendar between these times" is substantially harder to write. Instead of retrieving all events which start within the supplied times, you need need to check if any previously defined repeating events will occur inside that window. When

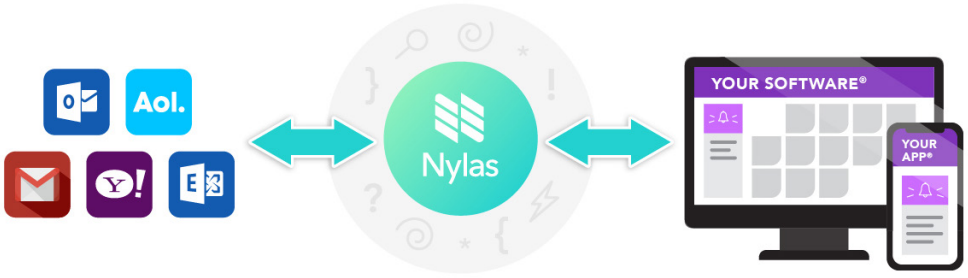
you have large numbers of recurring events combined with large time ranges, you start to run into memory limits as well!

Recurring events with Microsoft Exchange.

Unfortunately, the world of Microsoft Exchange is totally different, and the underlying Exchange ActiveSync protocol expresses recurrences and exceptions in a completely different format via WBXML like this:

```
<Recurrence>
  <Type>3</Type>
  <Interval>1</Interval>
  <WeekOfMonth>4</WeekOfMonth>
  <DayOfWeek>32</DayOfWeek>
  <CalendarType>0</CalendarType>
</Recurrence>
```

Connect to Any Calendar in 4 Lines of Code With the Nylas Calendar API



You can build unique connections with every individual calendar provider, or you can use the Nylas Calendar API and save months of developer time through one simple integration.

The Nylas events API makes it simple to generate an accurate representation of a user's calendar. The original recurrence information is available in RRULE format as `recurrence` on an event, but you can also simply add `expand_recurring=True` as a URL parameter to automatically expand all recurring events. This is a quick way to focus on building features, rather than figure out the details of repetitions, cancellations, and exceptions yourself.

With Nylas, you get all the features of integrating directly with the calendar service provider itself and more, including: full CRUD, powerful analytics, and enterprise-grade security.

Ready to get started?

Get started with a free 30 day trial, or learn more from one of our platform specialists.

Nylas

Nylas.com

Github.com/Nylas

@Nylas