Using iCalendar RRULE in Postgres

Posted on **June 22, 2023**

RRULE is an iCalendar specification for computing recurring calendar events. Something like, on first Tuesday of each month pay this bill. This is done as a rule that generates the occurrences of the event as needed. This solves the issue of a continually recurring event having to be physically stored as set of occurrences. The link above has some examples and there is this site RRULE generator where you can explore the options. This post will be a light introduction on how to store to, retrieve from a Postgres database the rules using Python and Javascript. Then use that information to populate a Javascript calendar in a Flask application. For Python the **rrule** module of the dateutil program will be used. In Javascript the rrule.js program which is a port of dateutil.rrule.

Setting up Python dateutil:

```
from dateutil.parser import parse
from dateutil.rrule import *

all for rrule is

["rrule", "rruleset", "rrulestr",

"YEARLY", "MONTHLY", "WEEKLY", "DAILY",

"HOURLY", "MINUTELY", "SECONDLY",

"MO", "TU", "WE", "TH", "FR", "SA", "SU"]
```

Examples.

Note the use of count. This is good habit to get into until you are sure of what the rule is going to produce. Unless you want to produce an infinite list of occurrences and bring your computer to its knees:). Don't ask me how I know.

Start at dstart and reoccur every month on same day of month for five occurences.

```
list(rrule(freq=MONTHLY, count=5, dtstart=parse("06/22/23")))

[datetime.datetime(2023, 6, 22, 0, 0),
 datetime.datetime(2023, 7, 22, 0, 0),
 datetime.datetime(2023, 8, 22, 0, 0),
```

```
 datetime.datetime(2023, 9, 22, 0, 0),
 datetime.datetime(2023, 10, 22, 0, 0)]
```

Same as above but specify occurrences to be on 31st of month. This skips month with < 31 days as the RRULE specification requires incorrect dates and/or times to be skipped not 'rounded' down.

```
list(rrule(freq=MONTHLY, bymonthday=31, count=5, dtstart=parse("06/22/23")))

[datetime.datetime(2023, 7, 31, 0, 0),
 datetime.datetime(2023, 8, 31, 0, 0),
 datetime.datetime(2023, 10, 31, 0, 0),
 datetime.datetime(2023, 12, 31, 0, 0),
 datetime.datetime(2024, 1, 31, 0, 0)]
```

bymonthday supports negative indexing, so to get last day of month regardless of its day number use -1.

```
list(rrule(freq=MONTHLY, bymonthday=-1, count=5, dtstart=parse("06/22/23")))

[datetime.datetime(2023, 6, 30, 0, 0),
 datetime.datetime(2023, 7, 31, 0, 0),
 datetime.datetime(2023, 8, 31, 0, 0),
 datetime.datetime(2023, 9, 30, 0, 0),
 datetime.datetime(2023, 10, 31, 0, 0)]
```

To get a better idea of what is possible I recommend looking at the examples here rrule examples

Incorporating RRULE into Postgres.

Create database table to hold rules and associated information.

```
CREATE TABLE public.rrule_example(
    task_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    task_title varchar NOT NULL,
    task_desc varchar NOT NULL,
    task_rrule varchar NOT NULL,
    start_date date NOT NULL,
    until_date date
);
```

Underlying RRULE is a string format that is fully explained in the RFC. The quick and dirty way to derive that in dateutil.rrule is to use the str() method on a rrule.

```
r = rrule(freq=WEEKLY, interval=2,   dtstart=parse("06/22/2023"))

r.__str__()
'DTSTART:20230622T000000\nRRULE:FREQ=WEEKLY;INTERVAL=2'
```

Insert string form of rrule into database.

```
INSERT INTO public.rrule_example OVERRIDING SYSTEM VALUE VALUES (1, 'Every two week

select * from rrule_example;
-[ RECORD 1 ]----------------------------------------
task_id    | 1
task_title | Every two weeks
task_desc  | Task occurrs every two weeks on Thursday
task_rrule | DTSTART:20230622T000000                 +
           | RRULE:FREQ=WEEKLY;INTERVAL=2
```

```
start_date | 06/22/2023
until_date | NULL
```

Create function to find next rule occurrence using plpython3u procedural language.

```
CREATE OR REPLACE FUNCTION public.rrule_next_occurrence(t_rrule character
varying, start_dt timestamp with time zone)
RETURNS timestamp with time zone
LANGUAGE plpython3u
SECURITY DEFINER
AS $function$
from datetime import datetime
from dateutil.parser import parse
from dateutil.rrule import rrulestr

rule = rrulestr(t_rrule, ignoretz=True)
next_occ = rule.after(parse(start_dt, ignoretz=True), inc=True)

return next_occ

$function$
;
```

The function uses dateutil.rrulestr to parse the string version of the rrule. Then the after() method to find first occurrence of rule after specified date.

```
select rrule_next_occurrence(task_rrule, '2023-06-21') from rrule_example where tas

rrule_next_occurrence
-------------------------
 06/22/2023 00:00:00 PDT
```

Create function to find previous rule occurrence.

```
CREATE OR REPLACE FUNCTION public.rrule_prior_occurrence(t_rrule character
varying, start_dt timestamp with time zone)
RETURNS timestamp with time zone
LANGUAGE plpython3u
SECURITY DEFINER
AS $function$
from datetime import datetime
from dateutil.parser import parse
from dateutil.rrule import rrulestr

rule = rrulestr(t_rrule, ignoretz=True)
prior_occ = rule.before(parse(start_dt, ignoretz=True), inc=True)

return prior_occ

$function$
;
```

Use rrulestr to parse string rrule. Then before() to find last occurrence of rule before specified date.

```
select rrule_prior_occurrence(task_rrule, '2023-06-23') from rrule_example where ta

 rrule_prior_occurrence
-------------------------
 06/22/2023 00:00:00 PDT
```

Using this information in a Web page.

Using Flask set up FullCalendar(https://fullcalendar.io/) calendar to display recurring events using rrule.js(https://github.com/jakubroztocil/rrule).

Need to include rrule-tz.js first then the FullCalendar rrule plugin.

```
<!--rrule.js with timezone support-->
<script type=""text/javascript" src="{{ url_for('static',
filename='js/external/rrule/rrule-tz.js') }}"></script>
<script type=""text/javascript" src="{{ url_for('static',
filename='js/external/full_calendar/main.js') }}"></script>
<!--FullCalendar rrule plugin-->
<script type=""text/javascript" src="{{ url_for('static',
filename='js/external/rrule/main.global.js') }}"></script>
```

In calendar constructor eventSources is where the calendar gets the information to fill in the calendar.

```
<script>

      document.addEventListener('DOMContentLoaded', function() {
        var calendarEl = document.getElementById('calendar');
        var calendar = new FullCalendar.Calendar(calendarEl, {
            timeZone: "US/Pacific",
            slotMinTime: "07:00",
            slotMaxTime: "19:00",
            slotDuration: "00:15:00",
            forceEventDuration: true,
            defaultTimedEventDuration: "00:15",
            initialView: "dayGridMonth",
            headerToolbar: {
                left: "prev,next today, prevYear,nextYear",
                center: "title",
                right: "dayGridMonth,timeGridWeek,timeGridDay"
                },
            stickyHeaderDates: true,
            eventSources: [
                {
                   url: "/task_calendar_data",
                },
                {events:
                    [{
                        title: 'Weekly Mon/Fri',
                        rrule: {
                            freq: 'weekly',
                            interval: 1,
                            byweekday: [ 'mo', 'fr' ],
                            dtstart: '2023-06-01T10:30:00',
                            until: '2023-10-31'
                        }
                    }],
                    id: "fixed_event"


                }
            ]
        });
        calendar.render();
      });

    </script>
```

In this case there are two sources url which fetches from a view in Flask and events which is a fixed event that uses the rrule.js syntax to build an event.

The view is:

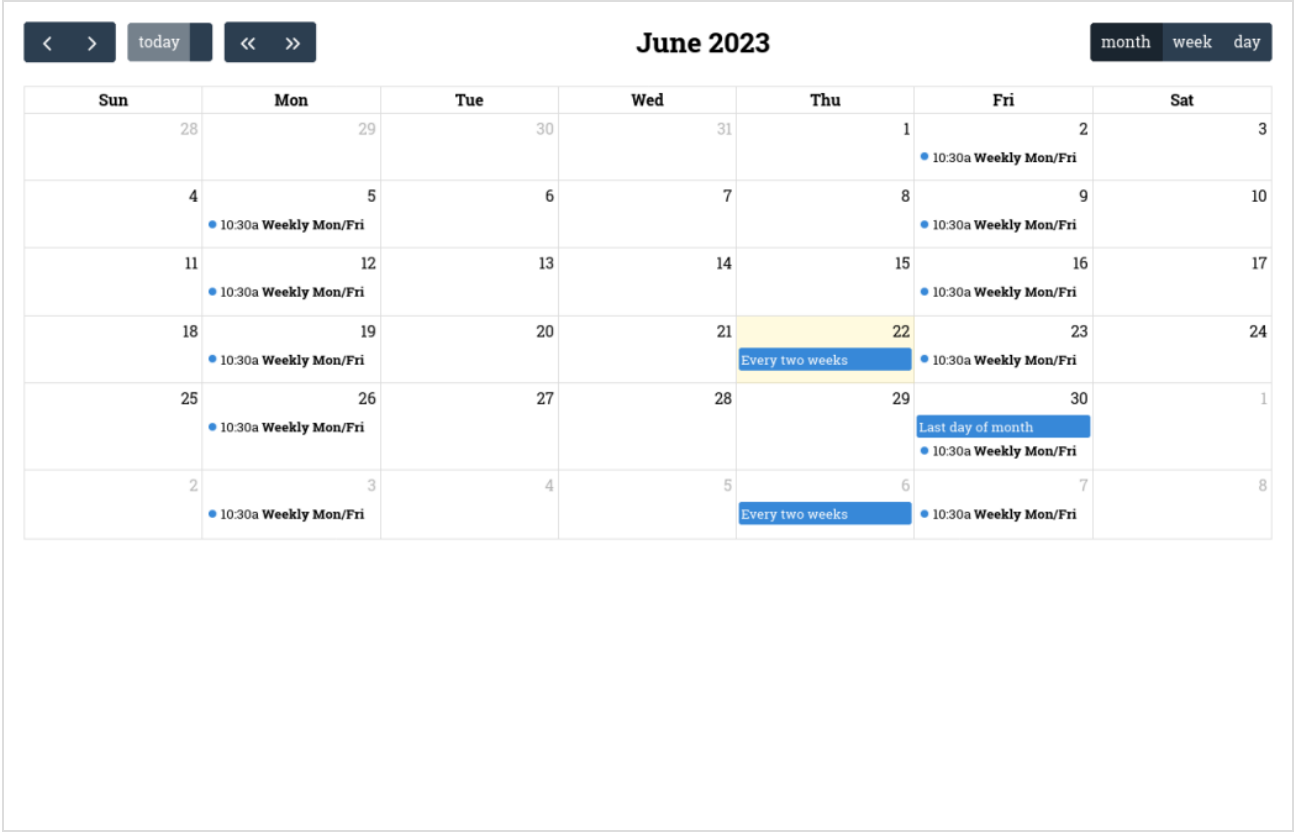
```
@calendar_bp.route("/task_calendar_data")
def taskCalendarData():
    today_dt = date.today()
    start_dt = request.args.get("start", today_dt.strftime("%m/%d/%Y"))
    end_dt = request.args.get("end",
                              (today_dt
                               + timedelta(days=1)).strftime("%m/%d/%Y"))
    # The connection(con) returned from get_db() uses cursor_factory=RealDictCursor
    # so results are returned as dictionaries.
    con = db.get_db()
    cur = con.cursor()
    cur.execute("select * from rrule_example")
    rs = cur.fetchall()
    tasks = []
    if rs:
        for task in rs:
            tasks.append({"id": task["task_id"], "title": task["task_title"],
                          "rrule": task["task_rrule"], "allDay": True})
    response = current_app.response_class(
                response=json.dumps(tasks),
                mimetype='application/json'
            )
    return response
```

allDay is set True to pin the task to 00:00.

Insert a rrule that shows an occurrence on last day of month.

```
INSERT INTO
    public.rrule_example OVERRIDING SYSTEM VALUE
VALUES
(2, 'Last day of month', 'Task occurrs last day of each month',
E'DTSTART:20230622T000000\nRRULE:FREQ=MONTHLY;BYMONTHDAY=-1',
'2023-06-22', NULL);
```

The calendar display for the rrules inserted into the database and from the eventSources in the calendar constructor. The current month and October 2023 when the rrule in the calendar constructor ends.



This entry was posted in **Postgres** by **aklaver**. Bookmark the **permalink [https://aklaver.org/wordpress/2023/06/22/using-icalendar-rrule-in-postgres/]** .