DjangoSchedule Documentation

Release 1.0

Tony Hauber, Yann Malet, Rock Howard

Oct 06, 2017

Contents

DjangoSchedule is an open-source calendaring application.

CHAPTER 1

Install

```
pip install django-scheduler
```

add package to INSTALLED_APPS in `settings.py`:

```
'schedule',
```

make sure that you have "`django.template.context_processors.request`" in TEMPLATE_CONTEXT_PROCESSORS.

CHAPTER 2

A Quick Overview

What is an Event?

An event doesn't have any date or time associated with it, just a rule for how it recurs. In a way it designates a set of occurrences. A weekly staff meeting is a perfect example. A weekly staff meeting is an event, it says what it is and how often it recurs. Now if we were to say Tuesday's staff meeting, that's an occurrence. That is, a specific element in the set of occurrences designated by weekly staff meeting.

There is an exception, and that is the "one-time" event. If your boss calls and sets up a meeting today at 3. That's a one-time event. It's only going to happen this one time. That doesn't mean it's an occurrence. It just means that it's an event which represents a set of occurrences that only has one occurrence in it.

What is an Occurrence?

An occurrence is an instance of an event. If we have an event and it is Weekly staff meetings which occur every Tuesday, then next Tuesday's staff meeting is an occurrence.

What does persisted Occurrences mean?

Occurrences are generated programmatically. This is because we cannot store all of the occurrences in the database, because there could be infinite occurrences. But we still want to be able to persist data about occurrences. Like cancelling an occurrence, moving an occurrence or storing a list of attendees with the occurrence. This is done lazily. An occurrence is generated programmatically until it needs to be saved to the database. When you use any function to get an occurrence, it will be completely transparent whether it was generated programatically or whether it is persisted (expect that persisted ones will have a `pk`). Just treat them like they are persisted and you shouldn't run into any trouble.

What is a Rule?

A rule defines how an event will recur. As of right now, there are no rules included with the app, so you will have to create your own. Doing this is somewhat straight forward.

Accessing Occurrences with an Event

Because some Event can recur indefinitely, you cannot have a function like, `event.get_all_occurrences()`, because that would be an infinite list. So, there are two ways of getting occurrences given an event.

`get_occurrences(start, end)`

This gives you a list of all occurrences that occur inclusively after start and exclusively before end. When we say occur, that means that they exist at all between start and end. If occurrence ends 10 seconds after `start` then it will be in the list, and if an occurrence starts 10 seconds before `end` then it will also be in the list.

`occurrences_after(after)`

This method produces a generator that generates events inclusively after the given datetime `after`. If no date is given then it uses now.

Accessing Occurrences from lists of Events

You are often going to have a list of events and want to get occurrences from them. To do this you can use Periods, and EventListManagers.

CHAPTER 3

Periods

One of the goals of DjangoSchedule is to make occurrence generation and persistence easy. To do this it creates simple classes for accessing these occurrences. These are Periods. Period is an object that is initiated with an iterable object of events, a start datetime, and an end datetime.

It is common to subclass Period for common periods of time. Some of these already exist in the project. Year, Month, Week, Day

Expect more in the future: Hour, HalfHour

Period Base Class

This is the base class from which all other periods inherit. It contains all of the base functionality for retrieving occurrences. It is instantiated with a list of events, a start date, and an end date. *The start date is inclusive, the end date is exclusive.* i.e. [start, end)

```
>>> p = Period(datetime.datetime(2008,4,1,0,0))
```

get_occurrences()

This method is for getting the occurrences from the list of passed in events. It returns the occurrences that exist in the period for every event. If I have a list of events `my_events`, and I want to know all of the occurrences from today to next week I simply create a Period object and then call get_occurrences. It will return a sorted list of Occurrences.

```
import datetime

today = datetime.datetime.now()
this_week = Period(my_events, today, today+datetime.timedelta(days=7))
this_week.get_occurrences()
```

classify_occurrence(occurrence)

You use this method to determine how the Occurrence `occurrence` relates to the period. This method returns a dictionary. The keys are `"class"` and `"occurrence"`. The class key returns a number from 0 to 3 and the occurrence key returns the occurrence.

Classes:

0: Only started during this period.

1: Started and ended during this period.

2: Didn't start or end in this period, but does exist during this period.

3: Only ended during this period.

get_occurrence_partials()

This method is used for getting all the occurrences, but getting them as classified occurrences. Simply it runs classify_occurrence on each occurrence in get_occurrence and returns that list.

```
import datetime

today = datetime.datetime.now()
this_week = Period(my_events, today, today+datetime.timedelta(days=7))
this_week.get_occurrences() == [classified_occurrence['occurrence'] for classified_
↪occurrence in this_week.get_occurrence_partials()]
```

has_occurrence()

This method returns whether there are any occurrences in this period

Year

The year period is instantiated with a list of events and a date or datetime object. It will resemble the year in which that date exists.

```
>>> p = Year(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 1, 1, 0, 0)
>>> p.end
datetime.datetime(2009, 1, 1, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

get_months()

This function returns 12 Month objects which resemble the 12 months in the Year period.

Month

The Month period is instantiated with a list of events and a date or datetime object. It resembles the month that contains the date or datetime object that was passed in.

```
>>> p = Month(events, datetime.datetime(2008,4,4))
>>> p.start
datetime.datetime(2008, 4, 1, 0, 0)
>>> p.end
datetime.datetime(2008, 5, 1, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

get_weeks()

This method returns a list of Week objects that occur at all during that month. The week does not have to be fully encapsulated in the month, just has to exist in the month at all.

get_days()

This method returns a list of Day objects that occur during the month.

get_day(day_number)

This method returns a specific day in a year given its day number.

Week

The Week period is instantiated with a list of events and a date or datetime object. It resembles the week that contains the date or datetime object that was passed in.

```
>>> p = Week(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 3, 30, 0, 0)
>>> p.end
datetime.datetime(2008, 4, 6, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

get_days()

This method returns the 7 Day objects that represent the days in a Week period.

Day

The Day period is instantiated with a list of events and a date or datetime object. It resembles the day that contains the date or datetime object that was passed in.

```
>>> p = Day(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 4, 1, 0, 0)
>>> p.end
datetime.datetime(2008, 4, 2, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

CHAPTER 4

Utilities

There are some utility classes found in the utils module that help with certain tasks.

EventListManager

EventListManager objects are instantiated with a list of events. That list of events dictates the following methods

occurrences_after(after)

Creates a generator that produces the next occurrence inclusively after the datetime `after`.

OccurrenceReplacer

If you get more into the internals of django-schedule, and decide to create your own method for producing occurrences, instead of using one of the public facing methods for this, you are going to want to replace the occurrence you produce with a persisted one, if a persisted one exists. To facilitate this in a standardized way you have the OccurrenceReplacer class.

To instantiate it you give it the pool of persisted occurrences you would like to check in.

```
>>> persisted_occurrences = my_event.occurrence_set.all()
>>> occ_replacer = OccurrenceReplacer(persisted_occurrences)
```

Now you have two convenient methods: get_occurrence and has_occurrence.

get_occurrence(occurrence)

This method returns either the passed-in occurrence or the equivalent persisted occurrences from the pool of persisted occurrences this OccurrenceReplacer was instantiated with.

```
>>> # my_generated_occurrence is an occurrence that was programatically
>>> # generated from an event
>>> occurrence = occ_replacer.get_occurrence(my_generated_occurrence)
```

has_occurrence(occurrence)

This method returns a boolean. It returns True of the OccurrenceReplacer has an occurrence it would like to replace with the give occurrence, and false if it does not.

```
>>> hasattr(my_generated_occurrence, 'pk')
False
>>> occ_replacer.has_occurrence(my_generated_occurrence)
True
>>> occurrence = occ_replacer.get_occurrence(my_generated_occurrence)
>>> hasattr(occurrence, 'pk')
True
>>> # Now with my_other_occurrence which does not have a persisted counterpart
>>> hasattr(my_other_occurrence, 'pk')
False
>>> occ_replacer.has_occurrence(my_other_occurrence)
False
>>> occurrence = occ_replacer.get_occurrence(my_other_occurrence)
>>> hasattr(occurrence, 'pk')
False
```

CHAPTER 5

Useful Template Tags

All of the templatetags are located in templatetags/scheduletags.py. You can look at more of them there. I am only going to talk about a few here.

To load the template tags this must be in your template

```
{% load scheduletags %}
```

querystring_for_date

Usage `{% querystring_for_date <date>[ <num>] %}`

This template tag produces a querystring that describes `date`. It turns date into a dictionary and then turns that dictionary into a querystring, in this fashion:

```
>>> date = datetime.datetime(2009,4,1)
>>> querystring_for_date(date)
'?year=2009&month=4&day=1&hour=0&minute=0&second=0'
```

This is useful when creating links as the calendar_by_period view uses this to display any date besides `datetime.datetime.now()`. The `num` argument can be used to say how specific you want to be about the date. If you were displaying a yearly calendar you only care about the year so `num` would only have to be `1`. See the examples below

```
>>> querystring_for_date(date, num=1)
'?year=2009'
>>> # Now if we only need the month
>>> querystring_for_date(date, num=2)
'?year=2009&month=4'
>>> # Now everything except the seconds
>>> querystring_for_date(date, num=5)
'?year=2009&month=4&day=1&hour=0&minute=0'
```

CHAPTER 6

Views

calendar

This view is for displaying meta_data about calendars. Upcoming events, Name, description and so on and so forth. It should be noted that this is probably not the best view for displaying a calendar in a traditional sense, i.e. displaying a month calendar or a year calendar, as it does not equip the context with any period objects. If you would like to do this you should use calendar_by_period.

Required Arguments

`request` As always the request object.

`calendar_slug` The slug of the calendar to be displayed.

Optional Arguments

`template_name`

default 'schedule/calendar.html'.

This is the template that will be rendered.

Context Variables

`calendar` The Calendar object designated by the `calendar_slug`.

calendar_by_period

This view is for getting a calendar, but also getting periods with that calendar. Which periods you get, is designated with the list periods. You can designate which date you want the periods to be initialized to by passing a date in

request.GET. See the template tag `query_string_for_date`.

Required Arguments

`request` As always the request object.

`calendar_slug` The slug of the calendar to be displayed.

Optional Arguments

`template_name`

default 'schedule/calendar_by_period.html'

This is the template that will be rendered.

`periods`

default `[]`

This is a list of Period Subclasses that designates which periods you would like to instantiate and put in the context.

Context Variables

`date` This was the date that was generated from the query string.

`periods` this is a dictionary that returns the periods from the list you passed in. If you passed in Month and Day, then your dictionary would look like this

```
{
    'month': <schedule.periods.Month object>
    'day':   <schedule.periods.Day object>
}
```

So in the template to access the Day period in the context you simply use `periods.day`.

`calendar` This is the Calendar that is designated by the `calendar_slug`.

`weekday_names` This is for convenience. It returns the local names of weekdays for internationalization.

event

This view is for showing an event. It is important to remember that an event is not an occurrence. Events define a set of recurring occurrences. If you would like to display an occurrence (a single instance of a recurring event) use `occurrence`.

Required Arguments

`request` As always the request object

`event_id` the id of the event to be displayed

Optional Arguments

template_name

default 'schedule/calendar_by_period.html'

This is the template that will be rendered.

Context Variables

event This is the event designated by the event_id.

back_url this is the url that referred to this view.

occurrence

This view is used to display an occurrence. There are two methods of displaying an occurrence.

Required Arguments

request As always the request object.

event_id the id of the event that produces the occurrence.

From here you need a way to distinguish the occurrence and that involves

occurrence_id if its persisted

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using `get_absolute_url` from the Occurrence model will help you standardize this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

Optional Arguments

template_name

default 'schedule/calendar_by_period.html'

This is the template that will be rendered

Context Variables

event the event that produces the occurrence

occurrence the occurrence to be displayed

back_url the url from which this request was referred

edit_occurrence

This view is used to edit an occurrence.

Required Arguments

request As always the request object

event_id the id for the event

From here you need a way to distinguish the occurrence and that involves

occurrence_id the id of the occurrence if its persisted

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using `get_edit_url` from the Occurrence model will help you standardize this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

Optional Arguments

template_name

> **default** 'schedule/calendar_by_period.html'
>
> This is the template that will be rendered.

Context Variables

form an instance of OccurrenceForm to be displayed.

occurrence an instance of the occurrence being modified.

cancel_occurrence

This view is used to cancel an occurrence. It is worth noting that cancelling an occurrence doesn't stop it from being in occurrence lists or being persisted, it just changes the `cancelled` flag on the instance. It is important to check this flag when listing occurrences.

Also if this view is requested via POST, it will cancel the event and redirect. If this view is accessed via a GET request it will display a confirmation page.

Required Arguments

`request` As always the request object.

From here you need a way to distinguish the occurrence and that involves

`occurrence_id` if its persisted

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using get_cancel_url from the Occurrence model will help you standardize this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

Optional Arguments

`template_name`

default 'schedule/calendar_by_period.html'

This is the template that will be rendered, if this view is accessed via GET.

`next`

default the event detail page of `occurrence.event`

This is the url you wish to be redirected to after a successful cancelation.

Context Variables

`occurrence` An instance of the occurrence being modified.

create_or_edit_event

This view is used for creating or editing events. If it receives a GET request or if given an invalid form in a POST request it will render the template, or else it will redirect.

Required Arguments

request As always the request object.

calendar_id This is the calendar id of the event being created or edited.

Optional Arguments

template_name

default 'schedule/calendar_by_period.html'

This is the template that will be rendered.

event_id If you are editing an event, you need to pass in the id of the event, so that the form can be pre-propagated with the correct information and so also save works correctly.

next The url to redirect to upon successful completion or edition.

Context Variables

form An instance of EventForm to be displayed.

calendar A Calendar with id=calendar_id.

delete_event

This view is for deleting events. If the view is accessed via a POST request it will delete the event. If it is accessed via a GET request it will render a template to ask for confirmation.

Required Arguments

request As always the request object.

event_id The id of the event to be deleted.

Optional Arguments

template_name

default 'schedule/calendar_by_period.html'

This is the template that will be rendered.

next The url to redirect to after successful deletion.

login_required

default `True`

If you want to require a login before deletion happens you can set that here.

Context Variables

object The event object to be deleted.

CHAPTER 7

Models

Not Documented yet

CHAPTER 8

Settings

OCCURRENCE_CANCEL_REDIRECT

This setting controls the behavior of `Views.get_next_url()`. If set, all calendar modifications will redirect here (unless there is a *next* set in the request.)

SHOW_CANCELLED_OCCURRENCES

This setting controls the behavior of `Period.classify_occurence()`. If True, then occurrences that have been cancelled will be displayed with a css class of canceled, otherwise they won't appear at all.

Defaults to False

CHECK_EVENT_PERM_FUNC

This setting controls the callable used to determine if a user has permission to edit an event or occurrence. The callable must take the object (event) and the user and return a boolean.

example:

```
check_edit_permission(ob, user):
    return user.is_authenticated()
```

If ob is None, then the function is checking for permission to add new events.

CHECK_CALENDAR_PERM_FUNC

This setting controls the callable used to determine if a user has permission to add, update or delete events in a specific calendar. The callable must take the object (calendar) and the user and return a boolean.

example:

```
check_edit_permission(ob, user):
    return user.is_authenticated()
```

GET_EVENTS_FUNC

This setting controls the callable that gets all events for calendar display. The callable must take the request and the calendar and return a *QuerySet* of events. Modifying this setting allows you to pull events from multiple calendars or to filter events based on permissions.

example:

```
get_events(request, calendar):
    return calendar.event_set.all()
```

SCHEDULER_BASE_CLASSES

This setting allows for custom base classes to be used on specific models. Useful for adding fields, managers, or other elements.

Defaults to django.db.models.Model

Extend all the models using a list:

SCHEDULER_BASE_CLASSES = ['my_app.models.BaseAbstract1', 'my_app.models.BaseAbstract1']

Extend specific models using a dict, where the key is the model class name:

SCHEDULER_BASE_CLASSES = { 'Event': ['my_app.models.EventAbstract1', 'my_app..models.EventAbstract2'] 'Calendar': [my_app.models.CalendarAbstract']

}

SCHEDULER_ADMIN_FIELDS

This complements the SCHEDULER_BASE_CLASSES feature, by allowing fields added via a base class to be shown in the admin form for that model.

Example - EventBase adds a 'cost' field, and now the field can be shown in the admin form too.

‘‘‘ SCHEDULER_BASE_CLASSES = {

'Event': ['main.models.EventBase']

}

SCHEDULER_ADMIN_FIELDS = { 'Event': [('cost',)]

}

CHAPTER 9

Indices and tables

- genindex
- modindex
- search