django-generate-series Documentation

Release 0.4.5

Jack Linke

Jan 21, 2023

USER GUIDE

CHAPTER

ONE

DJANGO-GENERATE-SERIES

Use Postgres' generate_series to create sequences with Django's ORM

https://django-generate-series.readthedocs.io/

1.1 Goals

When using Postgres, the set-returning functions allow us to easily create sequences of numbers, dates, datetimes, etc. Unfortunately, this functionality is not currently available within the Django ORM.

This project makes it possible to create such sequences, which can then be used with Django QuerySets. For instance, assuming you have an Order model, you can create a set of sequential dates and then annotate each with the number of orders placed on that date. This will ensure you have no date gaps in the resulting QuerySet. To get the same effect without this package, additional post-processing of the QuerySet with Python would be required.

1.2 Terminology

Although this packages is named django-generate-series based on Postgres' `generate_series` set-returning function, mathematically we are creating a sequence rather than a series.

- **sequence**: Formally, "a list of objects (or events) which have been ordered in a sequential fashion; such that each member either comes before, or after, every other member."

  In django-generate-series, we can generate sequences of integers, decimals, dates, datetimes, as well as the equivalent ranges of each of these types.

- **term**: The *n*th item in the sequence, where '*n*th' can be found using the id of the model instance.

  This is the name of the field in the model which contains the term value.

1.3 API

The package includes a `generate_series` function from which you can create your own series-generating QuerySets. The field type passed into the function as `output_field` determines the resulting type of series that can be created. (Thanks, @adamchainz for the format suggestion!)

1.3.1 generate_series arguments

- ***start*** - The value at which the sequence should begin (required)
- ***stop*** - The value at which the sequence should end. For range types, this is the lower value of the final term (required)
- ***step*** - How many values to step from one term to the next. For range types, this is the step from the lower value of one term to the next. (required for non-integer types)
- ***span*** - For range types other than date and datetime, this determines the span of the lower value of a term and its upper value (optional, defaults to 1 if neeeded in the query)
- ***output_field*** - A django model field class, one of BigIntegerField, IntegerField, DecimalField, DateField, DateTimeField, BigIntegerRangeField, IntegerRangeField, DecimalRangeField, DateRangeField, or DateTimeRangeField. (required)
- ***include_id*** - If set to True, an auto-incrementing `id` field will be added to the QuerySet.
- ***max_digits*** - For decimal types, specifies the maximum digits
- ***decimal_places*** - For decimal types, specifies the number of decimal places
- ***default_bounds*** - In Django 4.1+, allows specifying bounds for list and tuple inputs. See Django docs

1.4 Basic Examples

```
# Create a bunch of sequential integers
integer_sequence_queryset = generate_series(
    0, 1000, output_field=models.IntegerField,
)

for item in integer_sequence_queryset:
    print(item.term)
```

Result:

```
term
----
0
1
2
3
4
5
6
7
8
9
10
...
1000
```

```
# Create a sequence of dates from now until a year from now
now = timezone.now().date()
```

(continues on next page)

(continued from previous page)

```
later = (now + timezone.timedelta(days=365))

date_sequence_queryset = generate_series(
    now, later, "1 days", output_field=models.DateField,
)

for item in date_sequence_queryset:
    print(item.term)
```

Result:

```
term
----
2022-04-27
2022-04-28
2022-04-29
2022-04-30
2022-05-01
2022-05-02
2022-05-03
...
2023-04-27
```

Note: See the docs and the example project in the tests directory for further examples of usage.

1.5 Usage with partial

If you often need sequences of a given field type or with certain args, you can use partial.

Example with default `include_id` and `output_field` values:

```
from functools import partial

int_and_id_series = partial(generate_series, include_id=True, output_
↪field=BigIntegerField)

qs = int_and_id_series(1, 100)
```

CHAPTER

TWO

INSTALLATION

2.1 Pip

django-generate-series can be installed with `pip`

```
pip install django-generate-series
```

2.2 Install in Django project

Add the package to `INSTALLED_APPS`

```
INSTALLED_APPS = [
    ...
    "django_generate_series",
    ...
]
```

CHAPTER

THREE

USAGE EXAMPLES

3.1 Basic integer sequence example

Generate a sequence of every third integer from -12 to 12.

```
from django.db import models
from django_generate_series.models import generate_series

integer_sequence = generate_series(-12, 12, 3, output_field=models.IntegerField)

for item in integer_sequence:
    print(item.term)

""" Example:
    -12
    -9
    -6
    -3
     0
     3
     6
     9
     12
"""
```

Resulting SQL

```
SELECT
  "django_generate_series_integerfieldseries"."term"
FROM
  (
    SELECT
      generate_series(-12, 12, 3) term
  ) AS django_generate_series_integerfieldseries;
```

3.2 Basic integer sequence example with id

Generate a sequence of every third integer from -12 to 12, along with an auto-incrementing id field.

To include the `id` field in any sequence, set `include_id=True`. This does add a small increase in overhead.

```
from django.db import models
from django_generate_series.models import generate_series

integer_sequence = generate_series(-12, 12, 3, include_id=True, output_field=models.
↪IntegerField)

for item in integer_sequence:
    print(item.id, item.term)

""" Example:
    1  -12
    2  -9
    3  -6
    4  -3
    5   0
    6   3
    7   6
    8   9
    9   12
"""
```

Resulting SQL

```
SELECT
  "django_generate_series_integerfieldseries"."id",
  "django_generate_series_integerfieldseries"."term"
FROM
  (
    SELECT
      row_number() over () as id,
      "term"
    FROM
      (
        SELECT
          generate_series(-12, 12, 3) term
      ) AS seriesquery
  ) AS django_generate_series_integerfieldseries;
```

3.3 Example with decimals

Generate a sequence of decimal values, starting from 0.000 and increasing by 1.234, until reaching 10.000

```
import decimal

decimal_sequence = generate_series(
    decimal.Decimal("0.000"), decimal.Decimal("10.000"), decimal.Decimal("1.234"),␣
↪output_field=models.DecimalField,
)

for item in decimal_sequence:
    print(item.term)

""" Example:
    0.000
    1.234
    2.468
    3.702
    4.936
    6.170
    7.404
    8.638
    9.872
"""
```

Resulting SQL

```
SELECT
  "django_generate_series_decimalfieldseries"."term"
FROM
  (
    SELECT
      generate_series(0.000, 10.000, 1.234) term
  ) AS django_generate_series_decimalfieldseries;
```

3.4 Get summed costs for orders placed every other day over the past month

Given a model like this (included in tests.example.core.models):

```
class SimpleOrder(models.Model):
    order_date = models.DateField()
    cost = models.IntegerField()
```

In this example, we want to get the summed costs for orders placed on every other day over the past month. Yes, this is a bit nonsensical, but it provides a pretty good example of how to use django-generate-series.

```
import random
from django.db.models import OuterRef, Subquery, Sum
from tests.example.core.random_utils import get_random_date
```

(continues on next page)

(continued from previous page)

```
from tests.example.core.models import SimpleOrder

# Get the current datetime and the datetime 30 days ago
now = timezone.now()
previous = now - timezone.timedelta(days=30)

def random_date_in_past_month():
    # Generate a radom date within the past 30 days
    return get_random_date(min_date=previous, max_timedelta=timezone.timedelta(days=30))

for x in range(0, 30):
    # Create 30 SimpleOrder instances with random date and a cost between $1 and $50
    SimpleOrder.objects.create(
        order_date=random_date_in_past_month(), cost=random.randrange(1, 50)
    )

# Create a Subquery of annotated SimpleOrder objects
simple_order_subquery = (
    SimpleOrder.objects.filter(order_date=OuterRef("term"))
    .order_by()
    .values("order_date")
    .annotate(sum_of_cost=Sum("cost"))
    .values("sum_of_cost")
)

# Our DateTest is expecting date values, so update our variables
previous = previous.date()
now = now.date()

# Annotate the generated DateTest sequence instances with the annotated Subquery
date_sequence_queryset = generate_series(
    previous, now, "2 days", output_field=models.DateField,
).annotate(daily_order_costs=Subquery(simple_order_subquery))

# Print out all of the SimpleOrder objects (these are randomly generated, so your␣
↪results may vary)
for item in SimpleOrder.objects.order_by("order_date"):
    print(item.order_date, item.cost)


""" Example:
    2022-03-28  3
    2022-03-31  26
    2022-04-01  16
    2022-04-01  19
    2022-04-02  19
    2022-04-03  40
    2022-04-05  29
    2022-04-07  26
    2022-04-07  48
    2022-04-09  36
    2022-04-09  24
```

(continues on next page)

(continued from previous page)

```
    2022-04-11  24
    2022-04-12  29
    2022-04-13  25
    2022-04-14  43
    2022-04-15  41
    2022-04-15  30
    2022-04-16  30
    2022-04-18  6
    2022-04-19  17
    2022-04-20  41
    2022-04-21  48
    2022-04-23  19
    2022-04-23  31
    2022-04-23  24
    2022-04-23  36
    2022-04-23  45
    2022-04-24  11
    2022-04-24  20
    2022-04-26  2
"""

# Print out the date_sequence_queryset
#    Remember this is the sum of order costs for every other day over the past month
for item in date_sequence_queryset:
    print(item.term, item.daily_order_costs)

""" Example:
    2022-03-28 00:00:00+00:00  3
    2022-03-30 00:00:00+00:00  None
    2022-04-01 00:00:00+00:00  35
    2022-04-03 00:00:00+00:00  40
    2022-04-05 00:00:00+00:00  29
    2022-04-07 00:00:00+00:00  74
    2022-04-09 00:00:00+00:00  60
    2022-04-11 00:00:00+00:00  24
    2022-04-13 00:00:00+00:00  25
    2022-04-15 00:00:00+00:00  71
    2022-04-17 00:00:00+00:00  None
    2022-04-19 00:00:00+00:00  17
    2022-04-21 00:00:00+00:00  48
    2022-04-23 00:00:00+00:00  155
    2022-04-25 00:00:00+00:00  None
    2022-04-27 00:00:00+00:00  None
"""
```

The resulting SQL would look something like

```
SELECT
  "django_generate_series_datefieldseries"."term",
  (
    SELECT
      SUM(U0."cost") AS "sum_of_cost"
```

(continues on next page)

(continued from previous page)

```
    FROM
      "core_simpleorder" U0
    WHERE
      U0."order_date" = "django_generate_series_datefieldseries"."term"
    GROUP BY
      U0."order_date"
  ) AS "daily_order_costs"
FROM
  (
    SELECT
      generate_series('2022-03-28' :: date, '2022-04-27' :: date, '2 days') :: date term
  ) AS django_generate_series_datefieldseries;
```

3.5 Work with a series of datetime ranges

This example creates a sequence of date ranges, each seven day in length from today to 90 days from now. Then, similar to the previous example, we will sum all of the tickets with an event_datetime which overlaps with a range.

Note the use of `Func` here bypasses Django's default 'group by' functionality, which allows us to select rows that fall within an entire range. Normally, django would try to group by specific matches, but we want to match anything that is contained within each range. (Thanks @niccolomineo for the tip!)

Given a model like this (included in tests.example.core.models).

```python
class Event(models.Model):
    event_datetime = models.DateTimeField()
    ticket_qty = models.IntegerField()
```

3.5.1 Create some random events

```python
import random
from django.contrib.postgres.fields import DateTimeRangeField
from django.db.models import OuterRef, Subquery, Sum
from django.utils import timezone
from tests.example.core.random_utils import get_random_datetime
from tests.example.core.models import Event
from django_generate_series.models import generate_series

# Get the current datetime and the datetime 90 days ago
now = timezone.now()
later = (now + timezone.timedelta(days=90))

def random_datetime_in_past_month():
    # Generate a radom date within the past 90 days
    return get_random_datetime(min_date=now, max_timedelta=timezone.timedelta(days=90))

for x in range(0, 30):
    # Create 30 Event instances with random datetime and a ticket_qty between 1 and 5
    event = Event.objects.create(
```

(continues on next page)

(continued from previous page)

```
        event_datetime=random_datetime_in_past_month(),
        ticket_qty=random.randrange(1, 5),
    )

# Create a Subquery of annotated Event objects
for item in Event.objects.all().order_by("event_datetime"):
    print(item.event_datetime, item.ticket_qty)

""" Example (broken up by 7-day segments for clarity):
    2022-04-28 14:27:42.986299+00:00  3
    2022-04-29 16:58:27.986299+00:00  3

    2022-05-05 11:34:05.986299+00:00  1
    2022-05-06 23:06:52.986299+00:00  2
    2022-05-10 12:08:59.986299+00:00  2

    2022-05-13 23:51:26.986299+00:00  2

    2022-05-18 06:53:05.986299+00:00  3
    2022-05-18 20:22:20.986299+00:00  3
    2022-05-24 21:28:06.986299+00:00  2

    2022-05-26 03:34:56.986299+00:00  1

    2022-06-01 06:15:13.986299+00:00  1
    2022-06-03 15:44:08.986299+00:00  4

    2022-06-08 13:28:02.986299+00:00  3
    2022-06-12 14:09:17.986299+00:00  3

    2022-06-17 14:44:59.986299+00:00  3
    2022-06-19 16:25:02.986299+00:00  1

    2022-06-25 22:49:32.986299+00:00  3
    2022-06-26 11:07:40.986299+00:00  1

    2022-06-30 10:22:05.986299+00:00  3
    2022-06-30 21:38:59.986299+00:00  2
    2022-07-03 15:04:01.986299+00:00  1

    2022-07-07 13:08:58.986299+00:00  1
    2022-07-07 18:41:42.986299+00:00  2
    2022-07-09 18:21:40.986299+00:00  2
    2022-07-11 20:32:52.986299+00:00  1

    2022-07-16 22:46:10.986299+00:00  3
    2022-07-17 05:00:04.986299+00:00  4

    2022-07-20 11:40:06.986299+00:00  1
    2022-07-23 12:53:13.986299+00:00  3
    2022-07-24 21:33:46.986299+00:00  2
"""
```

(continues on next page)

(continued from previous page)

```
event_subquery = (
    Event.objects.filter(event_datetime__contained_by=OuterRef("term"))
    .order_by()
    .annotate(sum_of_tickets=Func(F("ticket_qty"), function="SUM"))
    .values("sum_of_tickets")
)
```

3.5.2 Generate and annotate the datetime ranges

```
datetime_range_sequence = (
    generate_series(now, later, "7 days", output_field=DateTimeRangeField)
    .annotate(ticket_quantities=Subquery(event_subquery))
    .order_by("term")
)

for item in datetime_range_sequence:
    print(item.term, item.ticket_quantities)

""" Example:
    [2022-04-27 01:39:19.986299+00:00, 2022-05-04 01:39:19.986299+00:00)  6
    [2022-05-04 01:39:19.986299+00:00, 2022-05-11 01:39:19.986299+00:00)  5
    [2022-05-11 01:39:19.986299+00:00, 2022-05-18 01:39:19.986299+00:00)  2
    [2022-05-18 01:39:19.986299+00:00, 2022-05-25 01:39:19.986299+00:00)  8
    [2022-05-25 01:39:19.986299+00:00, 2022-06-01 01:39:19.986299+00:00)  1
    [2022-06-01 01:39:19.986299+00:00, 2022-06-08 01:39:19.986299+00:00)  5
    [2022-06-08 01:39:19.986299+00:00, 2022-06-15 01:39:19.986299+00:00)  6
    [2022-06-15 01:39:19.986299+00:00, 2022-06-22 01:39:19.986299+00:00)  4
    [2022-06-22 01:39:19.986299+00:00, 2022-06-29 01:39:19.986299+00:00)  4
    [2022-06-29 01:39:19.986299+00:00, 2022-07-06 01:39:19.986299+00:00)  6
    [2022-07-06 01:39:19.986299+00:00, 2022-07-13 01:39:19.986299+00:00)  6
    [2022-07-13 01:39:19.986299+00:00, 2022-07-20 01:39:19.986299+00:00)  7
"""
```

The resulting SQL would look something like

```
SELECT
  "django_generate_series_datetimerangefieldseries"."term",
  (
    SELECT
      SUM(U0."ticket_qty") AS "sum_of_tickets"
    FROM
      "core_event" U0
    WHERE
      U0."event_datetime" < @ "django_generate_series_datetimerangefieldseries"."term"␣
↪:: tstzrange
  ) AS "ticket_quantities"
FROM
  (
    --- 1
```

(continues on next page)

(continued from previous page)

```
    SELECT
      tstzrange((lag(a) OVER()), a, '[)') AS term
    FROM
      generate_series(
        timestamptz '2022-04-27T01:39:19.986299+00:00' :: timestamptz,
        timestamptz '2022-07-26T01:39:19.986299+00:00' :: timestamptz,
        interval '7 days'
      ) AS a OFFSET 1
  ) AS django_generate_series_datetimerangefieldseries
ORDER BY
  "django_generate_series_datetimerangefieldseries"."term" ASC;
```

3.6 Create a Histogram

This example is a slight modification of the example above, using the same Event model. Here we are creating histogram buckets with a size of 5, and counting how many events have a number of tickets that falls in a given bucket.

3.6.1 Create some random events

```
import random
from django.contrib.postgres.fields import IntegerRangeField
from django.db import models
from django.db.models import OuterRef, Subquery, Count
from django.utils import timezone
from tests.example.core.random_utils import get_random_datetime
from tests.example.core.models import Event
from django_generate_series.models import generate_series


def random_datetime_in_past_month():
    # Generate a radom date within the past 90 days
    return get_random_datetime(min_date=timezone.now(), max_timedelta=timezone.
↪timedelta(days=90))

for x in range(0, 30):
    # Create 30 Event instances with random datetime and a ticket_qty between 1 and 50
    event = Event.objects.create(
        event_datetime=random_datetime_in_past_month(),
        ticket_qty=random.randrange(1, 50),
    )

# Create a Subquery of annotated Event objects
for item in Event.objects.all().order_by("ticket_qty"):
    print(item.event_datetime, item.ticket_qty)

""" Example
    2022-07-15 04:05:55.832641+00:00  9
    2022-05-06 17:45:00.836057+00:00  12
    2022-05-07 16:28:26.833515+00:00  14
```

(continues on next page)

(continued from previous page)

```
    2022-06-20 20:45:21.849374+00:00  15
    2022-05-08 22:15:01.856055+00:00  15
    2022-05-28 12:25:49.852562+00:00  19
    2022-05-14 20:17:00.831192+00:00  19
    2022-07-18 17:48:43.836904+00:00  19
    2022-07-11 01:49:23.843757+00:00  20
    2022-07-08 05:36:27.835197+00:00  21
    2022-05-27 18:56:23.855121+00:00  24
    2022-06-13 00:18:37.837913+00:00  25
    2022-05-17 11:24:04.854219+00:00  27
    2022-06-18 00:45:12.850166+00:00  28
    2022-05-04 16:59:48.842067+00:00  29
    2022-06-14 03:15:12.856889+00:00  31
    2022-06-13 23:03:33.846176+00:00  32
    2022-07-04 23:53:10.848583+00:00  33
    2022-06-09 19:31:38.846988+00:00  36
    2022-06-29 01:25:15.853390+00:00  37
    2022-05-03 13:23:10.845404+00:00  38
    2022-05-16 08:13:33.844641+00:00  39
    2022-05-30 08:54:01.841235+00:00  40
    2022-07-16 22:22:37.838889+00:00  41
    2022-06-12 16:21:05.851736+00:00  42
    2022-04-30 16:39:31.834378+00:00  43
    2022-06-22 03:38:17.842864+00:00  45
    2022-07-02 15:54:05.840007+00:00  45
    2022-07-04 18:12:47.850951+00:00  48
    2022-05-05 11:44:31.847756+00:00  49

"""

# Using Count instead of Sum
event_subquery = (
    Event.objects.filter(ticket_qty__contained_by=OuterRef("term"))
    .order_by()
    .annotate(count_of_tickets=Func(F("ticket_qty"), function="COUNT"))
    .values("count_of_tickets")
)
```

3.6.2 Generate and annotate the datetime ranges

Here we create 10 buckets with a step and span of 5, from 1 to 50.

```
datetime_range_sequence = (
    generate_series(0, 49, 5, 5, output_field=IntegerRangeField)
    .annotate(ticket_quantities=Subquery(event_subquery))
    .order_by("term")
)

for item in datetime_range_sequence:
    print(item.term, item.ticket_quantities)
```

(continues on next page)

(continued from previous page)

```
""" Example:
    [0, 5)    None
    [5, 10)   1
    [10, 15)  2
    [15, 20)  5
    [20, 25)  3
    [25, 30)  4
    [30, 35)  3
    [35, 40)  4
    [40, 45)  4
    [45, 50)  4
"""
```

The resulting SQL would look something like

```
SELECT
  "django_generate_series_integerrangefieldseries"."term",
  (
    SELECT
      COUNT(U0."ticket_qty") AS "count_of_tickets"
    FROM
      "core_event" U0
    WHERE
      U0."ticket_qty" < @ "django_generate_series_integerrangefieldseries"."term" ::␣
↪int4range
  ) AS "ticket_quantities"
FROM
  (
    SELECT
      int4range(a, a + 5) AS term
    FROM
      generate_series(0, 49, 5) a
  ) AS django_generate_series_integerrangefieldseries
ORDER BY
  "django_generate_series_integerrangefieldseries"."term" ASC;
```

CHAPTER

FOUR

CONTRIBUTING

We welcome contributions that meet the goals and standards of this project. Contributions may include bug fixes, feature development, corrections or additional context for the documentation, submission of Issues on GitHub, etc.

For development and testing, you can run your own instance of Postgres (either locally or using a DBaaS), or you can use the provided Docker Compose yaml file to provision a containerized instance and data volume locally.

4.1 Getting up-and-running

4.1.1 Poetry

Install requirements

This installs all packages needed for development and testing.

```
poetry install
```

Note: You may need to run `poetry update` *if there have been minor version updates to required packages.*

Start poetry environment in shell

```
poetry shell
```

4.1.2 Using Your Own postgres Instance

To develop using your own Postgres instance, you may set the following environmental variables on your machine:

- DB_NAME (defaults to "postgres")
- DB_USER (defaults to "docker")
- DB_PASSWORD (defaults to "docker")
- DB_HOST (defaults to "localhost")
- DB_PORT (defaults to "9932")

The process of setting environmental variables varies between different operating systems. Generally, on macOS and Linux, you can use the following convention in the console:

```
export KEY=value
```

4.1.3 Using the Provided Docker Compose Postgres Instance

This guide assumes you already have Docker and Docker Compose installed.

Build & Bring up the Docker Compose container for Postgres services:

Run the following command to build and bring up the postgres service.

```
docker-compose -f dev.yml up -d --no-deps --force-recreate --build postgres
```

These are the database connection details:

```
DB = postgres
USER = docker
PASSWORD = docker
HOST = postgres
PORT = 9932
```

To check the status of the database container:

```
docker ps
```

Once running, you should be able to connect using the test app, psql, or other Postgres management tools if desired.

To completely remove the container and associated data:

```
docker-compose -f dev.yml down --rmi all --remove-orphans -v
```

4.1.4 Once you have a Running Postgres Instance

Install pre-commits:

These ensure code is formatted correctly upon commit. See the pre-commit docs for more information.

```
pre-commit install
```

Run the tests:

```
pytest
```

Run code coverage report:

```
coverage run -m pytest
```

Create html coverage report:

```
coverage html
```

Check the django test project:

```
python manage.py check
```

To run the example project in the python REPL:

```
python manage.py shell_plus
```

4.2 Build the docs

Within the docs directory, run this from the console:

```
make html
```

CHAPTER

FIVE

CREDITS

5.1 Development Lead

- Jack Linke jack@watervize.com

5.2 Contributors

- Adam Johnson (@AdamChainz)
- Niccolò Mineo (@niccolomineo)
- Khemmatat Theanvanichpant (tui95)

CHAPTER

SIX

HISTORY

6.1 0.4.3 (2022-04-27)

- Add ability to specify the span from lower to upper values in each term for ranges.

6.2 0.4.0 (2022-04-27)

Breaking changes

- Modify API taking into consideration recommendations from Adam Johnson.

6.3 0.3.0 (2022-04-25)

Breaking changes

- Changed from using `id` as the sequence value to using the `term` field.

6.4 0.2.1 (2022-04-25)

- Implemented all PRs from adamchainz.
- Improve test matrix.
- Cleanup unused code and comments.

6.5 0.2.0 (2022-04-23)

- Basic package functionality is implemented.
- Tests have been added.
- Initial documentation is added.

6.6 0.1.0 (2022-02-08)

- Built initial readme entry to start documenting project goals.

View this project on Github.