Time zones

Overview

When support for time zones is enabled, Django stores datetime information in UTC in the database, uses time-zone-aware datetime objects internally, and translates them to the end user's time zone in templates and forms.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if your website is available in only one time zone, it's still good practice to store data in UTC in your database. The main reason is daylight saving time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. This probably doesn't matter for your blog, but it's a problem if you over bill or under bill your customers by one hour, twice a year, every year. The solution to this problem is to use UTC in the code and use local time only when interacting with end users.

Time zone support is enabled by default. To disable it, set `USE_TZ = False` in your settings file.

Changed in Django 5.0:
In older version, time zone support was disabled by default.

Time zone support uses `zoneinfo`, which is part of the Python standard library from Python 3.9.

If you're wrestling with a particular problem, start with the time zone FAQ.

Concepts

Naive and aware datetime objects

Python's `datetime.datetime` objects have a `tzinfo` attribute that can be used to store time zone information, represented as an instance of a subclass of `datetime.tzinfo`. When this attribute is set and describes an offset, a datetime object is **aware**. Otherwise, it's **naive**.

You can use `is_aware()` and `is_naive()` to determine whether datetimes are aware or naive.

When time zone support is disabled, Django uses naive datetime objects in local time. This is sufficient for many use cases. In this mode, to obtain the current time, you would write:

```
import datetime

now = datetime.datetime.now()
```

When time zone support is enabled (`USE_TZ=True`), Django uses time-zone-aware datetime objects. If your code creates datetime objects, they should be aware too. In this mode, the example above becomes:

```
from django.utils import timezone

now = timezone.now()
```

Warning: Dealing with aware datetime objects isn't always intuitive. For instance, the `tzinfo` argument of the standard datetime constructor doesn't work reliably for time zones with DST. Using UTC is generally safe; if you're using other time zones, you should review the `zoneinfo` documentation carefully.

Note: Python's `datetime.time` objects also feature a `tzinfo` attribute, and PostgreSQL has a matching `time with time zone` type. However, as PostgreSQL's docs put it, this type "exhibits properties which lead to questionable usefulness".

Django only supports naive time objects and will raise an exception if you attempt to save an aware time object, as a timezone for a time with no associated date does not make sense.

Interpretation of naive datetime objects

When `USE_TZ` is `True`, Django still accepts naive datetime objects, in order to preserve backwards-compatibility. When the database layer receives one, it attempts to make it aware by interpreting it in the default time zone and raises a warning.

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. That's why you should always create aware datetime objects when time zone support is enabled. (See the `Using ZoneInfo section of the zoneinfo docs` for examples using the `fold` attribute to specify the offset that should apply to a datetime during a DST transition.)

In practice, this is rarely an issue. Django gives you aware datetime objects in the models and forms, and most often, new datetime objects are created from existing ones through `timedelta` arithmetic. The only datetime that's often created in application code is the current time, and `timezone.now()` automatically does the right thing.

Default time zone and current time zone

The **default time zone** is the time zone defined by the `TIME_ZONE` setting.

The **current time zone** is the time zone that's used for rendering.

You should set the current time zone to the end user's actual time zone with `activate()`. Otherwise, the default time zone is used.

> **Note:** As explained in the documentation of `TIME_ZONE`, Django sets environment variables so that its process runs in the default time zone. This happens regardless of the value of `USE_TZ` and of the current time zone.
>
> When `USE_TZ` is `True`, this is useful to preserve backwards-compatibility with applications that still rely on local time. However, as explained above, this isn't entirely reliable, and you should always work with aware datetimes in UTC in your own code. For instance, use `fromtimestamp()` and set the `tz` parameter to `utc`.

Selecting the current time zone

The current time zone is the equivalent of the current locale for translations. However, there's no equivalent of the `Accept-Language` HTTP header that Django could use to determine the user's time zone automatically. Instead, Django provides time zone selection functions. Use them to build the time zone selection logic that makes sense for you.

Most websites that care about time zones ask users in which time zone they live and store this information in the user's profile. For anonymous users, they use the time zone of their primary audience or UTC. `zoneinfo.available_timezones()` provides a set of available timezones that you can use to build a map from likely locations to time zones.

Here's an example that stores the current timezone in the session. (It skips error handling entirely for the sake of simplicity.)

Add the following middleware to `MIDDLEWARE`:

```
import zoneinfo

from django.utils import timezone


class TimezoneMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        tzname = request.session.get("django_timezone")
        if tzname:
            timezone.activate(zoneinfo.ZoneInfo(tzname))
        else:
            timezone.deactivate()
        return self.get_response(request)
```

Create a view that can set the current timezone:

```
from django.shortcuts import redirect, render

# Prepare a map of common locations to timezone choices you wish to offer.
common_timezones = {
    "London": "Europe/London",
    "Paris": "Europe/Paris",
    "New York": "America/New_York",
}


def set_timezone(request):
    if request.method == "POST":
        request.session["django_timezone"] = request.POST["timezone"]
        return redirect("/")
    else:
        return render(request, "template.html", {"timezones": common_timezones})
```

Include a form in `template.html` that will `POST` to this view:

```
{% load tz %}
{% get_current_timezone as TIME_ZONE %}
<form action="{% url 'set_timezone' %}" method="POST">
    {% csrf_token %}
    <label for="timezone">Time zone:</label>
    <select name="timezone">
        {% for city, tz in timezones %}
        <option value="{{ tz }}"{% if tz == TIME_ZONE %} selected{% endif %}>{{ city }}</option>
        {% endfor %}
    </select>
    <input type="submit" value="Set">
</form>
```

Time zone aware input in forms

When you enable time zone support, Django interprets datetimes entered in forms in the current time zone and returns aware datetime objects in `cleaned_data`.

Converted datetimes that don't exist or are ambiguous because they fall in a DST transition will be reported as invalid values.

Time zone aware output in templates

When you enable time zone support, Django converts aware datetime objects to the current time zone when they're rendered in templates. This behaves very much like format localization.

> **Warning:** Django doesn't convert naive datetime objects, because they could be ambiguous, and because your code should never produce naive datetimes when time zone support is enabled. However, you can force conversion with the template filters described below.

Conversion to local time isn't always appropriate – you may be generating output for computers rather than for humans. The following filters and tags, provided by the `tz` template tag library, allow you to control the time zone conversions.

Template tags

`localtime`

Enables or disables conversion of aware datetime objects to the current time zone in the contained block.

This tag has exactly the same effects as the `USE_TZ` setting as far as the template engine is concerned. It allows a more fine grained control of conversion.

To activate or deactivate conversion for a template block, use:

```
{% load tz %}

{% localtime on %}
    {{ value }}
{% endlocaltime %}

{% localtime off %}
    {{ value }}
{% endlocaltime %}
```

> **Note:** The value of `USE_TZ` isn't respected inside of a `{% localtime %}` block.

`timezone`

Sets or unsets the current time zone in the contained block. When the current time zone is unset, the default time zone applies.

```
{% load tz %}

{% timezone "Europe/Paris" %}
    Paris time: {{ value }}
{% endtimezone %}

{% timezone None %}
    Server time: {{ value }}
{% endtimezone %}
```

`get_current_timezone`

You can get the name of the current time zone using the `get_current_timezone` tag:

```
{% get_current_timezone as TIME_ZONE %}
```

Alternatively, you can activate the `tz()` context processor and use the `TIME_ZONE` context variable.

Template filters

These filters accept both aware and naive datetimes. For conversion purposes, they assume that naive datetimes are in the default time zone. They always return aware datetimes.

`localtime`

Forces conversion of a single value to the current time zone.

For example:

```
{% load tz %}

{{ value|localtime }}
```

`utc`

Forces conversion of a single value to UTC.

For example:

```
{% load tz %}

{{ value|utc }}
```

`timezone`

Forces conversion of a single value to an arbitrary timezone.

The argument must be an instance of a `tzinfo` subclass or a time zone name.

For example:

```
{% load tz %}

{{ value|timezone:"Europe/Paris" }}
```

Migration guide

Here's how to migrate a project that was started before Django supported time zones.

Database

PostgreSQL

The PostgreSQL backend stores datetimes as `timestamp with time zone`. In practice, this means it converts datetimes from the connection's time zone to UTC on storage, and from UTC to the connection's time zone on retrieval.

As a consequence, if you're using PostgreSQL, you can switch between `USE_TZ = False` and `USE_TZ = True` freely. The database connection's time zone will be set to `DATABASE-TIME_ZONE` or `UTC` respectively, so that Django obtains correct datetimes in all cases. You don't need to perform any data conversions.

Time zone settings: The `time_zone` configured for the connection in the `DATABASES` setting is distinct from the general `TIME_ZONE` setting.

Other databases

Other backends store datetimes without time zone information. If you switch from `USE_TZ = False` to `USE_TZ = True`, you must convert your data from local time to UTC – which isn't deterministic if your local time has DST.

Code

The first step is to add `USE_TZ = True` to your settings file. At this point, things should mostly work. If you create naive datetime objects in your code, Django makes them aware when necessary.

However, these conversions may fail around DST transitions, which means you aren't getting the full benefits of time zone support yet. Also, you're likely to run into a few problems because it's impossible to compare a naive datetime with an aware datetime. Since Django now gives you aware datetimes, you'll get exceptions wherever you compare a datetime that comes from a model or a form with a naive datetime that you've created in your code.

So the second step is to refactor your code wherever you instantiate datetime objects to make them aware. This can be done incrementally. `django.utils.timezone` defines some handy helpers for compatibility code: `now()`, `is_aware()`, `is_naive()`, `make_aware()`, and `make_naive()`.

Finally, in order to help you locate code that needs upgrading, Django raises a warning when you attempt to save a naive datetime to the database:

```
RuntimeWarning: DateTimeField ModelName.field_name received a naive
datetime (2012-01-01 00:00:00) while time zone support is active.
```

During development, you can turn such warnings into exceptions and get a traceback by adding the following to your settings file:

```
import warnings

warnings.filterwarnings(
    "error",
    r"DateTimeField .* received a naive datetime",
    RuntimeWarning,
    r"django\.db\.models\.fields",
)
```

Fixtures

When serializing an aware datetime, the UTC offset is included, like this:

```
"2011-09-01T13:20:30+03:00"
```

While for a naive datetime, it isn't:

```
"2011-09-01T13:20:30"
```

For models with `DateTimeField`s, this difference makes it impossible to write a fixture that works both with and without time zone support.

Fixtures generated with `USE_TZ = False`, or before Django 1.4, use the "naive" format. If your project contains such fixtures, after you enable time zone support, you'll see `RuntimeWarning`s when you load them. To get rid of the warnings, you must convert your fixtures to the "aware" format.

You can regenerate fixtures with `loaddata` then `dumpdata`. Or, if they're small enough, you can edit them to add the UTC offset that matches your `TIME_ZONE` to each serialized datetime.

FAQ

Setup

1. **I don't need multiple time zones. Should I enable time zone support?**

   Yes. When time zone support is enabled, Django uses a more accurate model of local time. This shields you from subtle and unreproducible bugs around daylight saving time (DST) transitions.

   When you enable time zone support, you'll encounter some errors because you're using naive datetimes where Django expects aware datetimes. Such errors show up when running tests. You'll quickly learn how to avoid invalid operations.

   On the other hand, bugs caused by the lack of time zone support are much harder to prevent, diagnose and fix. Anything that involves scheduled tasks or datetime arithmetic is a candidate for subtle bugs that will bite you only once or twice a year.

   For these reasons, time zone support is enabled by default in new projects, and you should keep it unless you have a very good reason not to.

2. **I've enabled time zone support. Am I safe?**

   Maybe. You're better protected from DST-related bugs, but you can still shoot yourself in the foot by carelessly turning naive datetimes into aware datetimes, and vice-versa.

   If your application connects to other systems – for instance, if it queries a web service – make sure datetimes are properly specified. To transmit datetimes safely, their representation should include the UTC offset, or their values should be in UTC (or both!).

Finally, our calendar system contains interesting edge cases. For example, you can't always subtract one year directly from a given date:

```
>>> import datetime
>>> def one_year_before(value):  # Wrong example.
...     return value.replace(year=value.year - 1)
...
>>> one_year_before(datetime.datetime(2012, 3, 1, 10, 0))
datetime.datetime(2011, 3, 1, 10, 0)
>>> one_year_before(datetime.datetime(2012, 2, 29, 10, 0))
Traceback (most recent call last):
...
ValueError: day is out of range for month
```

To implement such a function correctly, you must decide whether 2012-02-29 minus one year is 2011-02-28 or 2011-03-01, which depends on your business requirements.

3. **How do I interact with a database that stores datetimes in local time?**

   Set the `TIME_ZONE` option to the appropriate time zone for this database in the `DATABASES` setting.

   This is useful for connecting to a database that doesn't support time zones and that isn't managed by Django when `USE_TZ` is `True`.

Troubleshooting

1. **My application crashes with `TypeError: can't compare offset-naive and offset-aware datetimes` – what's wrong?**

   Let's reproduce this error by comparing a naive and an aware datetime:

   ```
   >>> from django.utils import timezone
   >>> aware = timezone.now()
   >>> naive = timezone.make_naive(aware)
   >>> naive == aware
   Traceback (most recent call last):
   ...
   TypeError: can't compare offset-naive and offset-aware datetimes
   ```

   If you encounter this error, most likely your code is comparing these two things:

   - a datetime provided by Django – for instance, a value read from a form or a model field. Since you enabled time zone support, it's aware.
   - a datetime generated by your code, which is naive (or you wouldn't be reading this).

   Generally, the correct solution is to change your code to use an aware datetime instead.

   If you're writing a pluggable application that's expected to work independently of the value of `USE_TZ`, you may find `django.utils.timezone.now()` useful. This function returns the current date and time as a naive datetime when `USE_TZ = False` and as an aware datetime when `USE_TZ = True`. You can add or subtract `datetime.timedelta` as needed.

2. **I see lots of `RuntimeWarning: DateTimeField received a naive datetime (YYYY-MM-DD HH:MM:SS) while time zone support is active` – is that bad?**

   When time zone support is enabled, the database layer expects to receive only aware datetimes from your code. This warning occurs when it receives a naive datetime. This indicates that you haven't finished porting your code for time zone support. Please refer to the migration guide for tips on this process.

   In the meantime, for backwards compatibility, the datetime is considered to be in the default time zone, which is generally what you expect.

3. **`now.date()` is yesterday! (or tomorrow)**

   If you've always used naive datetimes, you probably believe that you can convert a datetime to a date by calling its `date()` method. You also consider that a `date` is a lot like a `datetime`, except that it's less accurate.

   None of this is true in a time zone aware environment:

   ```
   >>> import datetime
   >>> import zoneinfo
   >>> paris_tz = zoneinfo.ZoneInfo("Europe/Paris")
   >>> new_york_tz = zoneinfo.ZoneInfo("America/New_York")
   >>> paris = datetime.datetime(2012, 3, 3, 1, 30, tzinfo=paris_tz)
   # This is the correct way to convert between time zones.
   >>> new_york = paris.astimezone(new_york_tz)
   >>> paris == new_york, paris.date() == new_york.date()
   (True, False)
   >>> paris - new_york, paris.date() - new_york.date()
   (datetime.timedelta(0), datetime.timedelta(1))
   >>> paris
   datetime.datetime(2012, 3, 3, 1, 30, tzinfo=zoneinfo.ZoneInfo(key='Europe/Paris'))
   >>> new_york
   datetime.datetime(2012, 3, 2, 19, 30, tzinfo=zoneinfo.ZoneInfo(key='America/New_York'))
   ```

   As this example shows, the same datetime has a different date, depending on the time zone in which it is represented. But the real problem is more fundamental.

A datetime represents a **point in time**. It's absolute: it doesn't depend on anything. On the contrary, a date is a **calendaring concept**. It's a period of time whose bounds depend on the time zone in which the date is considered. As you can see, these two concepts are fundamentally different, and converting a datetime to a date isn't a deterministic operation.

What does this mean in practice?

Generally, you should avoid converting a `datetime` to `date`. For instance, you can use the `date` template filter to only show the date part of a datetime. This filter will convert the datetime into the current time zone before formatting it, ensuring the results appear correctly.

If you really need to do the conversion yourself, you must ensure the datetime is converted to the appropriate time zone first. Usually, this will be the current timezone:

```
>>> from django.utils import timezone
>>> timezone.activate(zoneinfo.ZoneInfo("Asia/Singapore"))
# For this example, we set the time zone to Singapore, but here's how
# you would obtain the current time zone in the general case.
>>> current_tz = timezone.get_current_timezone()
>>> local = paris.astimezone(current_tz)
>>> local
datetime.datetime(2012, 3, 3, 8, 30, tzinfo=zoneinfo.ZoneInfo(key='Asia/Singapore'))
>>> local.date()
datetime.date(2012, 3, 3)
```

4. **I get an error** `" Are time zone definitions for your database installed? "`

   If you are using MySQL, see the Time zone definitions section of the MySQL notes for instructions on loading time zone definitions.

Usage

1. **I have a string `"2012-02-21 10:28:45"` and I know it's in the `"Europe/Helsinki"` time zone. How do I turn that into an aware datetime?**

   Here you need to create the required `ZoneInfo` instance and attach it to the naïve datetime:

   ```
   >>> import zoneinfo
   >>> from django.utils.dateparse import parse_datetime
   >>> naive = parse_datetime("2012-02-21 10:28:45")
   >>> naive.replace(tzinfo=zoneinfo.ZoneInfo("Europe/Helsinki"))
   datetime.datetime(2012, 2, 21, 10, 28, 45, tzinfo=zoneinfo.ZoneInfo(key='Europe/Helsinki'))
   ```

2. **How can I obtain the local time in the current time zone?**

   Well, the first question is, do you really need to?

   You should only use local time when you're interacting with humans, and the template layer provides filters and tags to convert datetimes to the time zone of your choice.

   Furthermore, Python knows how to compare aware datetimes, taking into account UTC offsets when necessary. It's much easier (and possibly faster) to write all your model and view code in UTC. So, in most circumstances, the datetime in UTC returned by `django.utils.timezone.now()` will be sufficient.

   For the sake of completeness, though, if you really want the local time in the current time zone, here's how you can obtain it:

   ```
   >>> from django.utils import timezone
   >>> timezone.localtime(timezone.now())
   datetime.datetime(2012, 3, 3, 20, 10, 53, 873365, tzinfo=zoneinfo.ZoneInfo(key='Europe/Paris'))
   ```

   In this example, the current time zone is `"Europe/Paris"`.

3. **How can I see all available time zones?**

   `zoneinfo.available_timezones()` provides the set of all valid keys for IANA time zones available to your system. See the docs for usage considerations.