dmserebr
9 ноя 2023 в 18:41

Почему вам стоит отказаться от использования timestamp в PostgreSQL

10 мин 14K

PostgreSQL*, Программирование*, Базы данных*

Из песочницы

Не секрет, что работа с часовыми поясами — боль, и многие разработчики объяснимо стараются ее избегать. Тем более что в каждом языке программирования / СУБД работа с часовыми поясами реализована по-разному.

Среди тех, кто работает с PostgreSQL, есть очень распространенное заблуждение про типы данных **timestamp** (который также именуется **timestamp without time zone**) и **timestamptz** (или **timestamp with time zone**). Вкратце его можно сформулировать так:

> Мне не нужен тип timestamp with time zone, т.к. у меня все находится в одном часовом поясе — и сервер, и клиенты.

В статье я постараюсь объяснить, почему даже в таком довольно простом сценарии можно запросто напороться на проблемы. А в более сложных (которые на самом деле чаще встречаются на практике, чем может показаться) баги при использовании **timestamp** практически гарантированы.

Статья не претендует на полноту, но надеюсь, что поможет получше разобраться в этой не самой тривиальной теме :)

Про часовые пояса (time zones)

Концепция часовых поясов — на самом деле сравнительно недавнее изобретение человечества. Когда в XIX веке появились железные дороги и телеграф, люди пришли к тому, что настраивать часы в каждом городе на местное солнечное время крайне неудобно. В конечном счете планета оказалась разделена на 24 “полосы”, время в каждой из которых отстоит от ”нулевого пояса” (который обозначается как GMT или UTC) на целое количество часов. Есть, конечно, исключения вроде Индии и Ирана, но про них не в этой статье.

Казалось бы, тогда часовой пояс каждого стационарного места на Земле мог бы быть идентифицирован одним числом (смещением в часах относительно Гринвича). К сожалению, это не работает по двум причинам:

- во многих странах мира (например, в Европе и США) применяется летнее время, и 2 раза в год смещение от GMT меняется;
- из-за законодательных изменений города или страны могут перманентно менять часовой пояс.

Поэтому была разработана база данных **tz database** (https://ru.wikipedia.org/wiki/Tz_database), в которой хранится актуальное соответствие географических идентификаторов часовым поясам. Идентификаторы, например, бывают такими: America/Buenos_Aires, Europe/Paris, Europe/Moscow и т. д. Каждому идентификатору соответствует набор правил, по которым можно вычислить

смещение от GMT на какую-то дату. И когда какой-то город переходит в другой часовой пояс, в базу вносятся изменения.

Как PostgreSQL работает с часовыми поясами?

В PG есть несколько типов данных, использующихся для обозначения времени: https://www.postgresql.org/docs/current/datatype-datetime.html

Здесь мы рассмотрим 2 наиболее часто встречающихся типа — **timestamp** и **timestamptz**.

В целом (по смыслу) **timestamp** соответствует локальному времени без учета часовых поясов, а **timestamptz** — времени с учетом часового пояса. Проще всего понять это, попробовав вывести текущее время с помощью функции **now()** без часового пояса и с ним:

```
public=> select now()::timestamp, now();

            now             |              now
----------------------------+-------------------------------
 2023-06-28 21:49:56.417841 | 2023-06-28 21:49:56.417841+03
(1 row)
```

Объяснить с

Функция **now()** возвращает таймстемп в формате **timestamptz**, с учетом часового пояса (обратите внимание на +03 в конце).

Может сложиться впечатление, что тип **timestamptz** хранит “таймстемп плюс таймзону”. Но это не так — на самом деле для типа данных **timestamptz** хранится только время в UTC (18:49:56.417841), а при отображении вычисляется итоговое время на основании “текущей таймзоны сессии”. Обратимся к документации постгри:

> All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the **TimeZone** configuration parameter before being displayed to the client.

Тут говорится, что текущая таймзона определяется значением системного параметра **TimeZone**: https://www.postgresql.org/docs/current/runtime-config-client.html#GUC-TIMEZONE

> `TimeZone ( string )`
>
> Sets the time zone for displaying and interpreting time stamps. The built-in default is `GMT` , but that is typically overridden in `postgresql.conf` ; initdb will install a setting there corresponding to its system environment. See **Section 8.5.3** for more information.

Вкратце: дефолтное значение параметра **TimeZone** в PG, как правило, проставляется процессом initdb (который инициализирует свежий кластер БД) равным часовому поясу хоста, на котором крутится база. Это можно проверить в файле **postgresql.conf**: `grep '^timezone' $PGDATA/postgresql.conf` . Затем каждый клиент может переопределить параметр **TimeZone** по своему усмотрению в рамках сессии.

Важно, что поскольку часовой пояс никогда не сохраняется вместе со значением таймстемпа, тип данных **timestamptz** занимает в памяти ровно столько же места (8 байт), сколько и **timestamp**. Поэтому аргумент про то, что использование **timestamp** экономит место в памяти, совершенно несостоятелен.

Хьюстон, у нас проблема

Сложности чаще всего случаются тогда, когда клиенты БД могут находиться в разных часовых поясах.

Предположим, у ООО “Рога и Копыта” есть сервер, где в **postgresql.conf** стоит настройка **TimeZone='Europe/Moscow'** (то есть UTC+3). Иными словами, в компании есть договоренность, что все “локальное” время понимается как московское. Пусть на сервере есть таблица **data** с двумя колонками: **created timestamp** и **created_tz timestamptz**.

Однажды к базе подключился клиент Вася из Челябинска, и у него в PG клиенте оказался проставлен **TimeZone=UTC+5** (например, забыл поменять пояс на московский). Пусть в момент времени 2023-10-22 18:47:41.962110 +05:00 он решил выполнить запрос

```
INSERT INTO data(created, created_tz) VALUES (now(), now());
```

Объяснить с

Поскольку **now()** возвращает **timestamptz**, при записи в **created** произойдет конвертация в **timestamp** в текущей таймзоне сессии. Поэтому в него запишется 18:47, а в **created_tz** - время в UTC (13:47).

Когда Вася решит сделать SELECT и получить результат обратно, ему вернется

```
SELECT created, created_tz FROM data;
2023-10-22 18:47:41.962110, 2023-10-22 18:47:41.962110 +05:00
```

Объяснить с

Обратите внимание, что во второй колонке при отображении произошла обратная конвертация из UTC в локальное время.

А теперь предположим, что пришел злой админ Вова, у которого стоит московская таймзона (**SET TimeZone=’Europe/Moscow’**). И ему нужно понять, во сколько Вася добавил строчку в таблицу. Если он сделает аналогичный SELECT, то получит:

```
SELECT created, created_tz FROM data;
2023-10-22 18:47:41.962110, 2023-10-22 16:47:41.962110 +03:00
```

Объяснить с

Для **created_tz** UTC сконвертировалось в локальную таймзону Вовы (московскую), а вот **created** показывается как есть. Глядя исключительно на это поле, невозможно понять, когда же реально была произведена запись! Информация о таймзоне Васи утеряна, вообще ни разу не очевидно, что клиент находился в челябинском часовом поясе.

Хотите еще проблем — не вопрос! Зоопарк из клиентов PostgreSQL

Выше была рассмотрена ситуация, когда клиенты находятся в разных часовых поясах и из-за использования **timestamp without time zone** пропадает возможность определить время события.

Но многие тут резонно заметят: у меня и сервер, и все клиенты в одном часовом поясе, зачем мне думать о каких-то таймзонах?

Если бы все было так просто :) То, с чем я столкнулся при работе с разными клиентами PG, меня поразило.

Сценарий, от которого и смешно, и грустно одновременно

Воспроизвести ситуацию очень просто. Возьмем PostgreSQL, у которого таймзона сервера стоит в **America/Buenos_Aires** (это UTC-3):

```
docker run --name pgdemo -p 5432:5432 -e POSTGRES_USER=pguser -e POSTGRES_PASSWORD=pgpasswd -e
```

Объяснить с

Тут мы запустили чистый инстанс базы Postgres с названием pgdemo, а также проставили в контейнере часовой пояс с помощью переменной TZ (https://www.gnu.org/software/libc/manual/html_node/TZ-Variable.html). Поскольку в файле **postgresql.conf** мы ничего явно не задавали, Postgres использует часовой пояс из docker-контейнера.

Также предположим, что к базе мы подсоединяемся из московского часового пояса (UTC+3). Возьмем 3 распространенных клиента:

- старый добрый **psql** (подключаемся через `psql postgresql://pguser@localhost:5432/postgres`);
- встроенный клиент в IntelliJ IDEA (также используется в других продуктах Jetbrains: PyCharm, DataGrip и так далее);
- DBeaver (популярный свободный десктопный клиент на основе JDBC).

Создадим таблицу (из любого клиента):

```
create table person (
  id integer primary key,
  name text not null,
  created timestamp not null default now(),
  created_tz timestamptz not null default now()
);
```

Объяснить с

Выполним из каждого клиента по запросу:

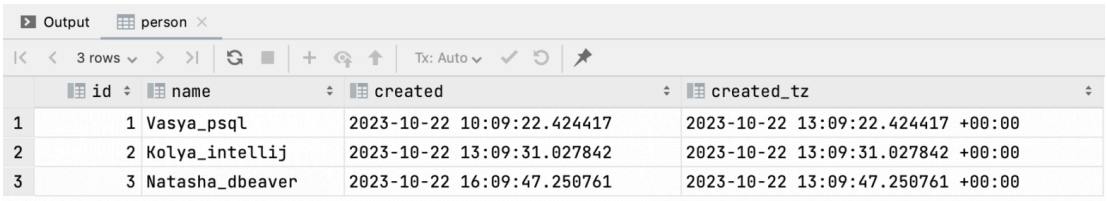
```
(psql)
insert into person (id, name) values (1, 'Vasya_psql');

(intellij)
insert into person (id, name) values (2, 'Kolya_intellij');

(dbeaver)
insert into person (id, name) values (3, 'Natasha_dbeaver');
```

Объяснить с

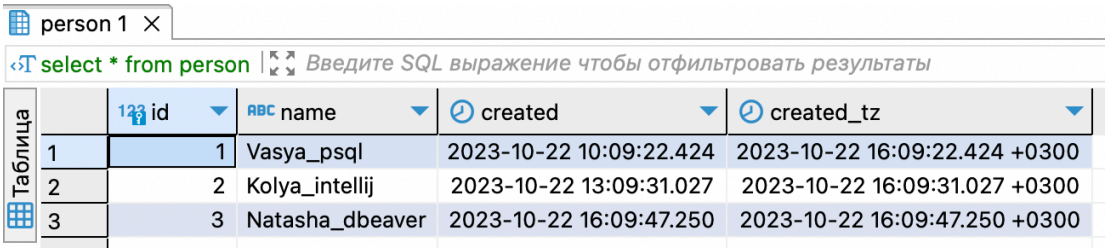
Таймстемпы в таком случае будут заполняться текущим временем. Пусть все клиенты физически находятся в московском часовом поясе (UTC+3). Тогда если они выполнят `select * from person` , то получится:

```
postgres=# select * from person;
 id |      name       |          created           |          created_tz
----+-----------------+----------------------------+-------------------------------
  1 | Vasya_psql      | 2023-10-22 10:09:22.424417 | 2023-10-22 10:09:22.424417-03
  2 | Kolya_intellij  | 2023-10-22 13:09:31.027842 | 2023-10-22 10:09:31.027842-03
  3 | Natasha_dbeaver | 2023-10-22 16:09:47.250761 | 2023-10-22 10:09:47.250761-03
(3 rows)
```

psql



intellij



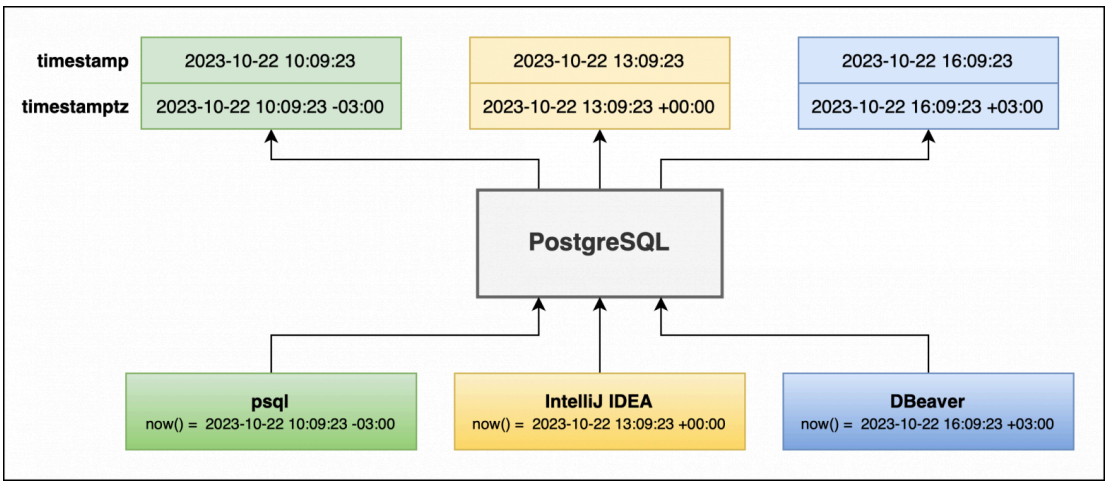
DBeaver

Обратите внимание - в колонку **created** (у которой тип **timestamp**) все клиенты проставили разные значения!

Почему так случилось? Функция **now()** возвращает текущее время в формате **timestamptz**. Затем происходит конвертация в **timestamp**. Результат определяется значением параметра **TimeZone** в каждой клиентской сессии и каждый клиент заполняет его по-своему. Самая боль в том, что в зависимости от реализации клиента дефолтное значение этого параметра может быть практически каким угодно:

- Для **psql** это таймзона сервера (в нашем случае UTC-3)
- Для **intellij / datagrip** — просто UTC (вот тут можно найти объяснение, почему так сделано: https://youtrack.jetbrains.com/issue/DBE-2996)
- Для **dbeaver** — таймзона клиента (UTC+3), как и в целом для большинства JDBC-based клиентов: http://github.com/pgjdbc/pgjdbc/issues/576.

В результате 3 клиента, которые физически находятся на одном компьютере, заполняют поле с типом **timestamp** разными значениями:



3 разных клиента — 3 разных дефолтных поведения

Из этого следует, что даже если сервер находился бы в клиентском часовом поясе (UTC+3), как минимум intellij проигнорировал бы это и записал бы время в UTC.

Вывод: *даже если у вас и сервер, и все клиенты физически находятся в одном часовом поясе, вы не застрахованы от потери данных при использовании **timestamp**!* Запросто могут найтись клиенты, у которых **TimeZone** “неожиданный” и есть риск, что в поле с типом **timestamp** может записаться время в неправильной таймзоне. Конечно, на клиенте дефолтную таймзону можно переопределить на нужную, но это проще простого забыть сделать (да и просто неудобно подобным заниматься).

Напротив, для колонки с типом **timestamptz** все в порядке. Значение **now()** напрямую сохраняется в поле без конвертаций и проблем не возникает. Клиенты отображают значение в соответствии со своими настройками (по времени сервера, клиента или в UTC), но во всех случаях верно и с указанием таймзоны.

Как можно было бы решить проблемы?

Самый правильный подход

В подавляющем большинстве случаев стоит отказаться от использования **timestamp** и перейти на **timestamptz**.

Это позволит (а) устранить неоднозначность в интерпретации таймстемпов, которые уже есть в базе и (б) избавиться от риска, что клиент подключается с “неожиданной” таймзоной и ломает данные (как выше в случае с Вовой). При этом **timestamptz** не занимает больше места в памяти, чем **timestamp**.

Сначала имеет смысл использовать **timestamptz** для всех новых таблиц, затем (в рамках технического долга) мигрировать существующие таблицы с **timestamp** на **timestamptz**.

Альтернативные варианты

Бывает, что в компании исторически сложилось использовать **timestamp** и понимать под ним время в каком-то выделенном часовом поясе (например, в UTC либо в MSK). Также код, который работает с БД, может иметь сложности с поддержкой **timestamptz**.

Какие есть альтернативы, если быстро перейти на **timestamptz** затруднительно?

1. Клиент может генерировать таймстемп сам и передавать его в команде INSERT в явном виде (с часами, минутами и секундами), а не использовать встроенную функции PG **now()** (и аналогичные, которые возвращают **timestamptz**). Тогда не будет производиться неявная конвертация из **timestamptz** в **timestamp**, поэтому таймзона клиента не будет влиять на результат. Если таймстемп генерируется вручную — то можно пользоваться функцией **make_timestamp**: `INSERT INTO data(created) VALUES (make_timestamp(2023, 10, 22, 9, 30, 0));` Также локальный таймстемп может генерироваться в коде приложения явно (например, через `LocalDateTime.now()` в Java).

   Минус подхода - ответственность за корректность добавляемого значения возлагается на клиента, что увеличивает вероятность ошибок. Также нужно дополнительно следить, чтобы все клиенты, которые пишут в базу, имели корректные настройки таймзоны.

2. Если необходимо вычислять текущее время на стороне БД, можно выполнять конвертацию в **timestamp** явно, используя **AT TIME ZONE**: `INSERT INTO data(created) VALUES (now() AT TIME ZONE 'Europe/Moscow');`

   Минус подхода — это требует действий в каждом скрипте, и каждый скриптописатель сам должен следить за тем, какую таймзону указывает (и что она соответствует его актуальной). Забыть указать **AT TIME ZONE** или указать неправильную таймзону — проще простого. А поскольку после вставки в базу информация о таймзоне теряется, раскопать потом что-то может быть невозможно.

Исключение: когда все же стоит использовать тип данных timestamp (without time zone)?

На практике из любого правила бывают исключения. С **timestamp** это тоже так — есть сценарий, когда других вариантов по сути нет: когда вам нужно задать какое-то время в неопределенном часовом поясе. Например, время для напоминания в будущем, для будильника или другого действия по расписанию. Чаще всего вам важно, чтобы будильник сработал, условно, в 9 утра по местному времени в какой-то конкретный день, при этом не важно, какой у вас при этом будет часовой пояс. Вы вообще можете уехать в другое место или часовой пояс в вашем городе может законодательно поменяться. В таких случаях наиболее логично применять именно **timestamp [without time zone]**.

Подведем итоги: почему timestamptz предпочтительнее?

- Тип данных **timestamptz** занимает столько же места в памяти, сколько и **timestamp**, при этом информации фактически содержит больше
- При использовании **now()**, **current_timestamp**, **localtimestamp** и т.п. результат того, что запишется в базу, не зависит от того, в каком часовом поясе клиент
- Даже если сервер и все клиенты в одном часовом поясе — нет проблем с разным дефолтным значением параметра **TimeZone** в разных PG-клиентах (например, из-за дефолтного UTC в intellij)

Исключение по сути одно — время с неизвестным / неопределенным часовым поясом (например, в будущем). В таком случае **timestamp** действительно является оптимальным выбором.

Ссылки для дальнейшего изучения:

https://www.iso.org/iso-8601-date-and-time-format.html
https://www.postgresql.org/docs/current/datatype-datetime.html
https://www.postgresql.org/docs/current/functions-datetime.html
https://wiki.postgresql.org/wiki/Don't_Do_This#Don.27t_use_timestamp_.28without_time_zone.29
https://phili.pe/posts/timestamps-and-time-zones-in-postgresql/

Теги: postgresql, timestamp, timestamptz

Хабы: PostgreSQL, Программирование, Базы данных

22 Карма 0 Общий рейтинг
@dmserebr
Пользователь
Подписаться

Комментарии 136

Virviil
9 ноя 2023 в 19:00

Думаю что большинство читателей этой статьи используют базу для крудошлепства, и в этом плане у нее есть только один клиент - бэк.

Использование timestamp с хранением всех времен в utc полностью беспроблемное решение при таких условиях.

Во вьюшках utc переводится в нужную зону относительно клиента. Внутри бэка нету никаких разночтений и путаницы при операциях с датами и временем.

Ответить

dmserebr
9 ноя 2023 в 19:06

Ну это классика: сегодня может казаться, что клиент только бекенд, а завтра появляется что-то новое и никому даже в голову не приходит, что это может что-то сломать.
Аргумент был бы валиден только в случае, если у timestamptz есть какой-то оверхед, но его нет.

Ответить

Virviil
9 ноя 2023 в 19:25

У timestampz есть когнитивный оверхед когда начинаешь считать «вчера это было или сегодня» на бэке. Или когда у third party api принимается дата в utc. И эти баги случаются гораздо раньше чем «гипотетический другой клиент» который обычно не появляется никогда

Ответить

dmserebr
9 ноя 2023 в 19:43

На бекенде timestamptz, если нужно, без проблем преобразуется в локальный (в системной таймзоне сервера) и там, где надо, работают с локальным. А вообще есть типы, которые напрямую маппятся на timestamptz (например, OffsetDateTime в Java) и в коде бекенда тоже ими можно оперировать.

Про "гипотетический клиент" — я бы сказал, это довольно сильное заявление, особенно в больших проектах на много человек. Даже если вы стараетесь следить, чтобы все insert выполнялись исключительно из кода бекенда, гарантий дать невозможно и кто будет после вас с кодом работать — тоже вопрос. При всем этом есть простое решение, как сделать надежно.

Ответить

n43jl
10 ноя 2023 в 14:42

На моей практике в 90% случаях бекенду вредно знать о timezone, поэтому всегда где-можно должно быть UTC.
Класть utc в timestamp и всегда заменять now() на "now() AS TIME ZONE 'utc'" или использовать timestamptz которое всегда конвертится к utc на стороне приложения - по моему одинаково. Имхо OffsetDateTime в Java в 90% анти-паттерн, зачем-то многие добавляют и процессят информацию о TZ когда она совершенно нерелевантна, и нужно использовать Instant (UTC).

Ответить

TerrorDroid
9 ноя 2023 в 22:46

> сегодня может казаться, что клиент только бекенд, а завтра появляется что-то новое

С куда большей вероятностью завтра никогда не наступит

Ответить

Ivan22
10 ноя 2023 в 12:41

- Завтра никогда не наступит, после нас хоть потоп, живем один раз...

- А можно нам другого архитектора?

Ответить

krabdb
10 ноя 2023 в 13:45

"В Гонконге уже завтра"!

Ответить

M_AJ
10 ноя 2023 в 10:30

> Ну это классика: сегодня может казаться, что клиент только бекенд, а завтра появляется что-то новое

Когда разные клиенты напрямую лезут в базу всегда есть вариант, что "завтра", а точнее при внесении изменений, кто-то о них не узнает, и что-то из-за этого сломается. Чтобы этого избежать, достаточно оставить одного клиента, который будет точно знать как интерпретировать записанное в базе, а другим отдавать через документированное API

Ответить

lwhouse
13 ноя 2023 в 08:56

1. Вероятность появления нового клиента к БД примерно равна вероятности замены Postgres на другую СУБД. Возможно, но в реальности практически никогда не происходит.
2. Если к БД обращается более одного клиента, мы теряем управляемость транзакциями (в бизнес смысле), мы теряем управляемость кэшированием, мы теряем гибкость в управлении миграциями - это как минимум. Потери от перечисленных факторов намного превышают потери от необходимости делать пересчет в таймзоны на бэкенде.

Ответить

Ivan22
13 ноя 2023 в 16:57

нового клиента нет, а вот какая-нить выгрузка напрямую всей БД в DWH может и появиться.

Ответить

Gromilo
10 ноя 2023 в 09:02

Я тот самый крудошлёп!

У меня обычно так:

- Время нужно хранить в UTC, а браузер отобразит правильно, например сообщение чата и время создания поста
- Время нужно хранить в UTC и у пользователя/устройства/города есть часовой пояс и сервер посчитает как правильно. Например, даты в письмах.
- Время нужно хранить ни в чём, т.к. это расписание соревнований в Самаре или в Челябинске и если соревнование начинается в 11, значит везде 11 часов не смотря на часовой пояс клиента.

Для этого я использовал timestamp, но в 6 версии Npgsql решили, что timestamp нужно переводить в локальное время и я пока сижу на
`AppContext.SetSwitch("Npgsql.EnableLegacyTimestampBehavior", true);`

По идее нужно переезжать на `DateTimeOffset`, но я пока не разбирался как жить с тем, что всегда есть `Offset`, который мне не нужен.

Ответить

Heggi
10 ноя 2023 в 14:19

В 7 версии Npgsql стало все веселее. Но, в целом, это все не сложно.

С DateTimeOffset работать просто.

Достаешь из БД данные? В DateTimeOffset время по UTC. Если надо, добавляешь смещение и получаешь время в нужном часовом поясе. Если не нужно (браузер сам умный), то так и отдаешь.

Сохраняешь в БД? Сделай offset = 0, иначе npgsql просто не даст это сохранить в БД (у нас используется gRpc, там Timestamp и так только в UTC передается, так что для нас это вообще не проблема. Только при работе со сторонними API надо быть внимательнее)

Ответить

withkittens
10 ноя 2023 в 14:47

Подумайте насчёт NodaTime, библиотека разделяет "разные времена" по разным типам:

- Время сообщения чата или создания поста - это `Instant`
- Если нужно помнить время в часовом поясе, то это `ZonedDateTime`
- Расписание соревнований - это `LocalDateTime` или даже `LocalTime`

Npgsql/EF Core библиотеку поддерживают.

Ответить

Fedorkov
10 ноя 2023 в 18:31

NodaTime - это самое надёжное решение для работы с часовыми поясами (и особенно с историей их изменений).

Но ещё надёжнее на уровне API отказаться принимать не-UTC время, тогда будет достаточно стандартного DateTimeOffset.

Ответить