DateTime vs DateTimeOffset

Asked 8 years, 9 months ago Active 2 months ago Viewed 179k times

642

253

Currently, we have a standard way of dealing with .net DateTimes in a TimeZone aware way: Whenever we produce a `DateTime` we do it in UTC (e.g. using `DateTime.UtcNow`), and whenever we display one, we convert back from UTC to the user's local time.

That works fine, but I've been reading about `DateTimeOffset` and how it captures the local and UTC time in the object itself. So the question is, what would be the advantages of using `DateTimeOffset` vs what we have already been doing?

c# .net datetime timezone datetimeoffset

edited May 8 '18 at 3:20

ice1000
3,684 4 17 55

asked Dec 2 '10 at 2:39

David Reis
4,479 4 29 41

3 There are some great answers below. But I'm still left wondering what, if anything, could have convinced you to start using DateTimeOffset. – HappyNomad Nov 4 '13 at 18:39

1 Might as well see when-would-you-prefer-datetime-over-datetimeoffset – nawfal Jan 31 '14 at 1:05

When it comes to storage, stackoverflow.com/questions/4715620/… is interesting too. – Dejan Jul 21 '16 at 18:22

9 Answers

1052

`DateTimeOffset` is a representation of *instantaneous time* (also known as *absolute time*). By that, I mean a moment in time that is universal for everyone (not accounting for leap seconds, or the relativistic effects of time dilation). Another way to represent instantaneous time is with a `DateTime` where `.Kind` is `DateTimeKind.Utc`.

This is distinct from *calendar time* (also known as *civil time*), which is a position on someone's calendar, and there are many different calendars all over the globe. We call these calendars *time zones*. Calendar time is represented by a `DateTime` where `.Kind` is `DateTimeKind.Unspecified`, or `DateTimeKind.Local`. And `.Local` is only meaningful in scenarios where you have an implied

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

So then, why `DateTimeOffset` instead of a UTC `DateTime` ? **It's all about perspective.** Let's use an analogy - we'll pretend to be photographers.

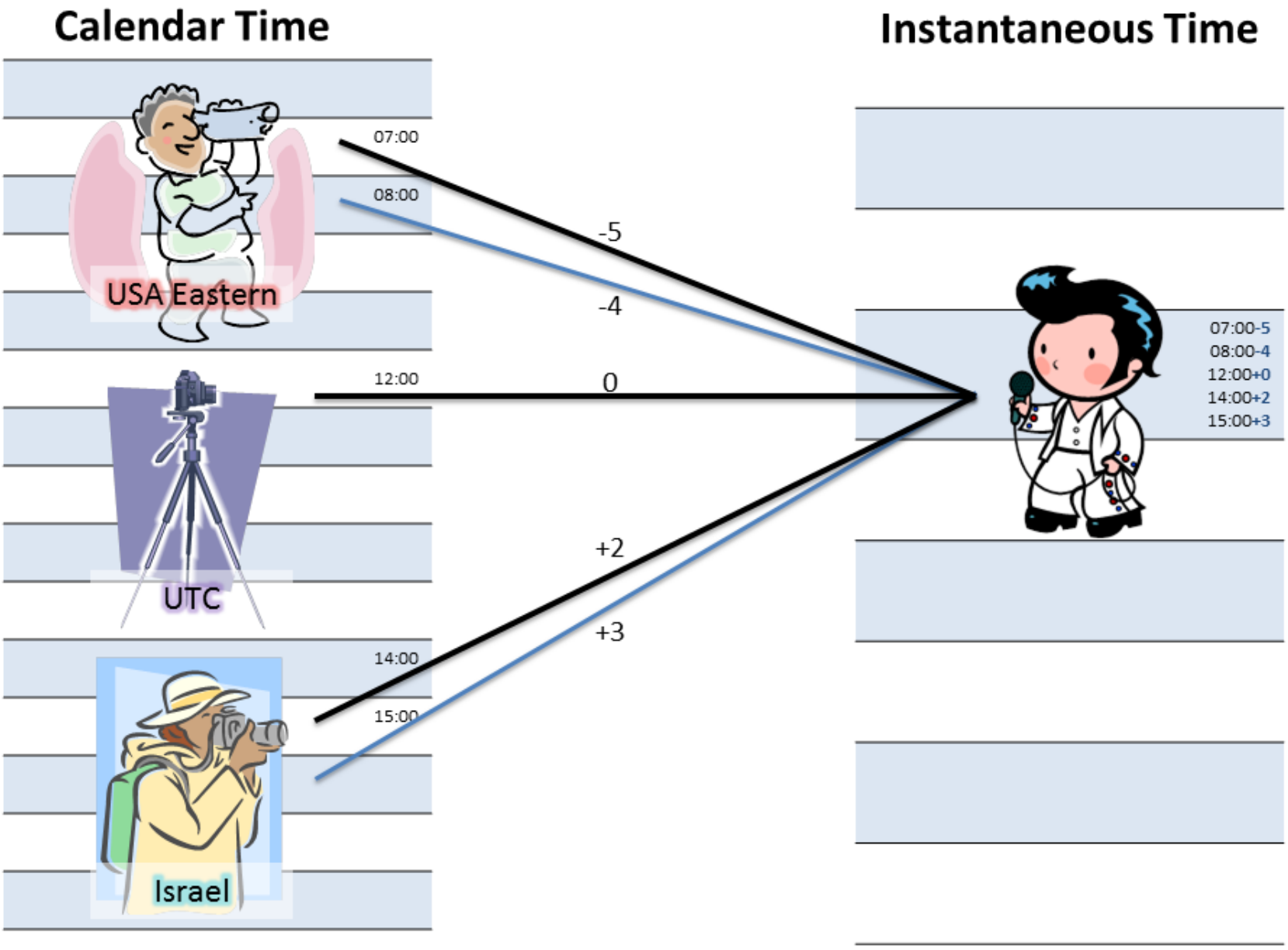
Imagine you are standing on a calendar timeline, pointing a camera at a person on the instantaneous timeline laid out in front of you. You line up your camera according to the rules of your timezone - which change periodically due to daylight saving time, or due to other changes to the legal definition of your time zone. (You don't have a steady hand, so your camera is shaky.)

The person standing in the photo would see the angle at which your camera came from. If others were taking pictures, they could be from different angles. This is what the `Offset` part of the `DateTimeOffset` represents.

So if you label your camera "Eastern Time", sometimes you are pointing from -5, and sometimes you are pointing from -4. There are cameras all over the world, all labeled different things, and all pointing at the same instantaneous timeline from different angles. Some of them are right next to (or on top of) each other, so just knowing the offset isn't enough to determine which timezone the time is related to.

And what about UTC? Well, it's the one camera out there that is guaranteed to have a steady hand. It's on a tripod, firmly anchored into the ground. It's not going anywhere. We call its angle of perspective the zero offset.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.



By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

- If you are representing time relative to some place in particular, represent it in calendar time with a `DateTime`. Just be sure you don't ever confuse one calendar with another. `Unspecified` should be your assumption. `Local` is only useful coming from `DateTime.Now`. For example, I might get `DateTime.Now` and save it in a database - but when I retrieve it, I have to assume that it is `Unspecified`. I can't rely that my local calendar is the same calendar that it was originally taken from.
- If you must always be certain of the moment, make sure you are representing instantaneous time. Use `DateTimeOffset` to enforce it, or use UTC `DateTime` by convention.
- If you need to track a moment of instantaneous time, but you want to also know "What time did the user think it was on their local calendar?" - then you *must* use a `DateTimeOffset`. This is very important for timekeeping systems, for example - both for technical and legal concerns.
- If you ever need to modify a previously recorded `DateTimeOffset` - you don't have enough information in the offset alone to ensure that the new offset is still relevant for the user. You must *also* store a timezone identifier (think - I need the name of that camera so I can take a new picture even if the position has changed).

  It should also be pointed out that Noda Time has a representation called `ZonedDateTime` for this, while the .Net base class library does not have anything similar. You would need to store both a `DateTimeOffset` and a `TimeZoneInfo.Id` value.
- Occasionally, you will want to represent a calendar time that is local to "whomever is looking at it". For example, when defining what *today* means. Today is always midnight to midnight, but these represent a near-infinite number of overlapping ranges on the instantaneous timeline. (In practice we have a finite number of timezones, but you can express offsets down to the tick) So in these situations, make sure you understand how to either limit the "who's asking?" question down to a single time zone, or deal with translating them back to instantaneous time as appropriate.

Here are a few other little bits about `DateTimeOffset` that back up this analogy, and some tips for keeping it straight:

- If you compare two `DateTimeOffset` values, they are first normalized to zero offset before comparing. In other words, `2012-01-01T00:00:00+00:00` and `2012-01-01T02:00:00+02:00` refer to the same instantaneous moment, and are therefore equivalent.
- If you are doing any unit testing and need to be certain of the offset, test *both* the `DateTimeOffset` value, and the `.Offset` property separately.
- There is a one-way implicit conversion built in to the .Net framework that lets you pass a `DateTime` into any `DateTimeOffset` parameter or variable. When doing so, **the `.Kind` matters**. If you pass a UTC kind, it will carry in with a zero offset, but if you pass either `.Local` or `.Unspecified`, it will assume to be **local**. The framework is basically saying, "Well, you asked me to convert calendar time to instantaneous time, but I have no idea where this came from, so I'm just going to use the local calendar." This is a huge gotcha if you load up an unspecified `DateTime` on a computer with a different timezone. (IMHO - that should throw an exception - but it doesn't.)

Shameless Plug:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Many people have shared with me that they find this analogy extremely valuable, so I included it in my Pluralsight course, Date and Time Fundamentals. You'll find a step-by-step walkthrough of the camera analogy in the second module, "Context Matters", in the clip titled "Calendar Time vs. Instantaneous Time".

edited Nov 19 '15 at 0:51

answered Jan 10 '13 at 22:09

Matt Johnson-Pint
152k 47 301 424

4 @ZackJannsen If you have a `DateTimeOffset` in C#, then you should persist that to a `DATETIMEOFFSET` in SQL Server. `DATETIME2` or just `DATETIME` (depending on the range required) are fine for regular `DateTime` values. Yes - you can resolve a local time from any pairing of timezone + dto or utc. The difference is - do you always want to be computing the rules with each resolve, or do you want to precalculate them? In many cases (sometimes for legal concerns) a DTO is a better choice. – Matt Johnson-Pint Mar 29 '13 at 4:15

3 @ZackJannsen For the second part of your question, I would recommend doing as much as possible server-side. Javascript is not that great for timezone calculation. If you must do it, use one of these libraries. But server side is best. If you have other more detailed questions, please start a new S.O. question for them and I will answer if I can. Thanks. – Matt Johnson-Pint Mar 29 '13 at 4:20

4 @JoaoLeme - That depends on where you obtained it from. You are correct that if you say `DateTimeOffset.Now` on the server, you will indeed get the server's offset. The point is that the `DateTimeOffset` type can retain that offset. You could just as easily do that on the client, send it to the server, and then your server would know the client's offset. – Matt Johnson-Pint Aug 9 '13 at 19:21

6 Really love the camera analogy. – Sachin Kainth Feb 7 '14 at 14:13

3 I more confused now after reading about pictures – Mikee Dec 1 '16 at 15:50

272

From Microsoft:

> These uses for DateTimeOffset values are much more common than those for DateTime values. As a result, DateTimeOffset should be considered the default date and time type for application development.

source: "Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo", *MSDN*

We use `DateTimeOffset` for nearly everything as our application deals with particular points in time (e.g. when a record was created/updated). As a side note, we use `DATETIMEOFFSET` in SQL Server 2008 as well.

I see `DateTime` as being useful when you want to deal with dates only, times only, or deal with either in a generic sense. For example, if you have an alarm that you want to go off every day at 7 am, you could store that in a `DateTime` utilizing a `DateTimeKind` of `Unspecified`

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Use caution when using a mix of `DateTimeOffset` and `DateTime` especially when assigning and comparing between the types. Also, only compare `DateTime` instances that are the same `DateTimeKind` because `DateTime` ignores timezone offset when comparing.

edited Aug 9 '17 at 3:54

Cœur
22k 10 127 179

answered Jan 9 '13 at 19:42

Clay
6,105 3 34 32

117 The accepted answer is overly long and the analogy is strained, this is a much better and more concise answer IMO. – nexus says Jul 11 '14 at 19:44

7 I'll just say that I like this answer too, and upvoted. Though in the last part - even ensuring `Kind` are the same, comparison could be in error. If both sides have `DateTimeKind.Unspecified` you don't really know that they came from the same time zone. If both sides are `DateTimeKind.Local` , *most* comparisons are going to be fine, but you could still have errors is one side is ambiguous in the local time zone. Really only `DateTimeKind.Utc` comparisons are foolproof, and yes, `DateTimeOffset` is usually preferred. (Cheers!) – Matt Johnson-Pint Apr 22 '16 at 20:25

+1 I'd add to this: The DataType you choose should reflect your intent. Do not use DateTimeOffset everywhere, just cause. If the Offset matters for your Calculations and Reading-From/Persisting-To the DataBase, then use DateTimeOffset. If it doesn't matter, then use DateTime, so you understand (just by looking at the DataType) that the Offset should have no bearing and Times should remain relative to the Locality of the Server/Machine your C# Code is running on. – MikeTeeVee Jul 22 at 18:50

63

DateTime is capable of storing only two distinct times, the local time and UTC. The *Kind* property indicates which.

DateTimeOffset expands on this by being able to store local times from anywhere in the world. It also stores the *offset* between that local time and UTC. Note how DateTime cannot do this unless you'd add an extra member to your class to store that UTC offset. Or only ever work with UTC. Which in itself is a fine idea btw.

answered Dec 2 '10 at 17:47

Hans Passant
812k 113 1385 2167

33

There's a few places where `DateTimeOffset` makes sense. One is when you're dealing with recurring events and daylight savings time. Let's say I want to set an alarm to go off at 9am every day. If I use the "store as UTC, display as local time" rule, then the alarm will be going off at a *different* time when daylight savings time is in effect.

There are probably others, but the above example is actually one that I've run into in the past (this was before the addition of

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

answered Dec 2 '10 at 2:59

Dean Harding
61.7k 8 125 170

10 DateTimeOffset doesn't fix the DST problem – JarrettV Jan 15 '13 at 18:47

2 Using TimeZoneInfo class does carry rules for DST. if you are on .net 3.5 or later then use either TimeZone or TimeZoneInfo classes to deal with dates that must handle Daylight Savings Time in conjunciton with the timezone offset. – Zack Jannsen Mar 29 '13 at 11:41

1 Yes good example of an exception (the alarm app) but when the time is more important than the date you should really store that separate in your schedule data structure for the application, i.e. occurrence type = Daily and time = 09:00. The point here is the developer needs to be aware of what type of date they are recording, calculating or presenting to users. Especially apps tend to be more global now we have the internet as standard and big app stores to write software for. As a side node I'd also like to see Microsoft add a separate Date and Time structure. – Tony Wall Apr 9 '14 at 9:12

2 Summarizing Jarrett's and Zack's comments: It sounds like DateTimeOffset *alone* will not handle the DST problem but using DateTimeOffset in conjunction with TimeZoneInfo will handle it. This is no different from DateTime where kind is Utc. In both cases I must know the time zone (not just the offset) of the calendar I am projecting the moment to. (I might store that in a user's profile or get it from the client (e.g. Windows) if possible). Sound right? – Jeremy Cook Nov 8 '14 at 5:31

"There's a few places where DateTimeOffset makes sense." --- Arguably, it more often makes sense than not. – Ronnie Overby Dec 5 '18 at 20:28

20

The most important distinction is that DateTime does not store time zone information, while DateTimeOffset does.

Although DateTime distinguishes between UTC and Local, there is absolutely no explicit time zone offset associated with it. If you do any kind of serialization or conversion, the server's time zone is going to be used. Even if you manually create a local time by adding minutes to offset a UTC time, you can still get bit in the serialization step, because (due to lack of any explicit offset in DateTime) it will use the server's time zone offset.

For example, if you serialize a DateTime value with Kind=Local using Json.Net and an ISO date format, you'll get a string like `2015-08-05T07:00:00-04` . Notice that last part (-04) had nothing to do with your DateTime or any offset you used to calculate it... it's just purely the server's time zone offset.

Meanwhile, DateTimeOffset explicitly includes the offset. It may not include the name of the time zone, but at least it includes the offset, and if you serialize it, you're going to get the explicitly included offset in your value instead of whatever the server's local time happens to be.

answered Aug 5 '15 at 23:00

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

11 with all the above answers, I wonder why no one bothered to write your single sentence that sums it all up `The most important distinction is that DateTime does not store time zone information, while DateTimeOffset does.` – Korayem May 15 '16 at 22:38

5 DateTimeOffset does NOT store time zone info. MS doc titled "Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo" specifies this stating: "A DateTimeOffset value is not tied to a particular time zone, but can originate from any of a variety of time zones". That said, DateTimeOffset IS time zone AWARE, containing the offset from UTC, which makes all the difference and is why it's MS recommended default class when dealing with app development that deals with date info. If you truly care about which specific timezone the data came from, you must preserve that separately – stonedauwg Jun 14 '17 at 20:35

Yes, but as has been shown in many places, + or - hours says nothing about what timezone you were in and is ultimately useless. Depending on what you need to do, you can just as well store a datetime as Kind.Unspecified and then store the id of its timezone and I think you are actually better off. – Arwin Apr 6 '18 at 13:14

9

Most of the answers are good , but i thought of adding some more links of MSDN for more information

- A brief History of DateTime - By Anthony Moore by BCL team
- Choosing between Datetime and DateTime Offset - by MSDN
- Do not forget SQL server 2008 onwards has a new Datatype as DateTimeOffset
- The .NET Framework includes the **DateTime**, **DateTimeOffset**, and **TimeZoneInfo** types, all of which can be used to build applications that work with dates and times.
- Performing Arithmetic Operations with Dates and Times-MSDN

edited Apr 29 '13 at 0:47

answered Apr 29 '13 at 0:41

dekdev

3,410 1 20 29

Also - blogs.msdn.microsoft.com/davidrickard/2012/04/06/… – Adrian K Dec 2 '16 at 20:22

7

A major difference is that `DateTimeOffset` can be used in conjunction with `TimeZoneInfo` to convert to local times in timezones other than the current one.

This is useful on a server application (e.g. ASP.NET) that is accessed by users in different timezones.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

104k 24 158 290

8 TimeZoneInfo work also with DateTime – Marco Staffoli Jun 29 '11 at 12:54

3 @Bugeo Bugeo is true, but there is a risk. You can compare two DateTimes by first calling "ToUniversalTime" on each. If you have exactly one value in the comparison that is DateTimeKind = Unspecified your strategy will fail. This potential for a failure is a reason to consider DateTimeOffset over DateTime when conversions to local time are required. – Zack Jannsen Mar 29 '13 at 11:49

As above, I think in this scenario you are better off iwth storing the TimeZoneId than using DateTimeOffset, which ultimately means nothing. – Arwin Apr 6 '18 at 13:15

7

This piece of code from Microsoft explains everything:

```
// Find difference between Date.Now and Date.UtcNow
  date1 = DateTime.Now;
  date2 = DateTime.UtcNow;
  difference = date1 - date2;
  Console.WriteLine("{0} - {1} = {2}", date1, date2, difference);

  // Find difference between Now and UtcNow using DateTimeOffset
  dateOffset1 = DateTimeOffset.Now;
  dateOffset2 = DateTimeOffset.UtcNow;
  difference = dateOffset1 - dateOffset2;
  Console.WriteLine("{0} - {1} = {2}",
                    dateOffset1, dateOffset2, difference);
  // If run in the Pacific Standard time zone on 4/2/2007, the example
  // displays the following output to the console:
  //    4/2/2007 7:23:57 PM - 4/3/2007 2:23:57 AM = -07:00:00
  //    4/2/2007 7:23:57 PM -07:00 - 4/3/2007 2:23:57 AM +00:00 = 00:00:00
```

answered Oct 19 '18 at 9:32

Mojtaba

433 5 13

The only negative side of DateTimeOffset I see is that Microsoft "forgot" (by design) to support it in their XmlSerializer class. But it has since been added to the XmlConvert utility class.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

XmlConvert.ToString

I say go ahead and use DateTimeOffset and TimeZoneInfo because of all the benefits, just beware when creating entities which will or may be serialized to or from XML (all business objects then).

edited Aug 27 '18 at 19:41

answered Apr 9 '14 at 11:11

Tony Wall

1,314 19 16

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.