January 2021

Robust Date-Time Representation

José Edvaldo Saraiva

Robust Date-Time Representation

ABSTRACT

Software application programming interfaces (APIs), frameworks, and databases don't always account for situations in which the rules for time zones and daylight saving time (DST) change. Software programmers frequently find dealing with dates, times, time-zone, and changes to DST rules to be unintuitive and error prone. This disclosure describes simple, intuitive techniques that enable developers to work with dates and times in a consistent manner free of unpredictable time or duration shifts. The disclosure defines primitive, derived, and chained date-times, notions incorporated transparently into the programming model that result in predictable date and time behavior under updates to time-zone rules.

KEYWORDS

- Date-time
- Standard time
- Daylight saving time (DST)
- Time zone
- Local time
- Calendar

BACKGROUND

Software application programming interfaces (APIs), frameworks, and databases don't always account for situations in which the rules for time zones and daylight saving time (DST) change. For example, the Energy Policy Act of 2005 led to data corruption in the form of time shifts. Nearly every year, some jurisdiction around the world makes tweaks to its DST rules.

Dealing with dates, times, time-zone, and DST rules can be unintuitive for software developers and prone to error.

As mentioned earlier, the problem is not only at the application level, but more pervasive, e.g., at the levels of APIs, data formats, frameworks, protocols, etc. Although common cases are handled well, corner cases are often not: Days that are 23 or 25 hours long can show as being 24 hours long. Meetings can get stretched or contracted across daylight saving transitions, even though such transitions are well known. Rule changes that result in a changed UTC (Coordinated Universal Time) offset result in unpredictable program behavior.

Below we consider 3 examples, in increasing levels of complexity. This is not an exhaustive list of possible issues and only touches the tip of the iceberg.

Example 1: The first example involves just a common user interface issue. A person in Los Angeles schedules a meeting with a person in Mumbai, India between 1:30 AM and 3:30 AM Los Angeles local time on March 14, 2021, believing it to last two hours because that is how it is displayed in most applications. Los Angeles observes daylight saving time (DST), such that at 2:00 AM on 21st March 2021, the Los Angeles clocks jump to 3:00 AM. Mumbai does not observe DST. Thus 1:30 AM PST corresponds to 3 PM Indian Standard Time (IST), while 3:30 AM Los Angeles time (after the time change in Los Angeles) corresponds to 4 PM IST. The person from Mumbai, India that attends the meeting therefore believes the meeting is to last only one hour, between 3 PM and 4 PM IST. This is just an user interface issue, but will be confusing to users that are not aware of the DST transition.

Example 2: A more convoluted case involves changes in the time zone rules, which can cause time shifts depending on how the application represents the date-time values. In November of 2020, a person in Los Angeles schedules a meeting between 6 AM and 8 AM local time on the

first of July of the next year. Assume the application stores the values in UTC as 1 PM and 3 PM respectively, since the offset from UTC is -7 hours during the period DST is observed. The application then calculates the local time when displaying the information in the user interface. This is a common pattern. In the interval between the meeting being scheduled and its actual occurrence, the government, federal or local, passes legislation that abolishes DST in Los Angeles, fixing the UTC offset to standard time (-8 hours). The application will then load the UTC values previously stored, converting them to local time based on current rules and show the meeting as occurring between 5 AM and 7 AM, Los Angeles time, instead of between 6 AM and 8 AM. Application developers will need to figure out on their own how to represent date-time values in a way to preserve the intent of the users. A correctly implemented application will have enough data to make the use case work.

Example 3: In this next example, local time is preserved but the duration of the meeting is not. A person in Los Angeles schedules a meeting between 1:30 AM and 4:30 AM in Los Angeles local time on March 14, 2021. Notice the start time is before the DST transition and the end time is after the DST transition based on the rules at the time of creation. The meeting has a duration of 2 hours, since the interval from 2:00 AM to 3:00 AM will be missing. Let's assume that the application is capable of presenting that fact to the user, perhaps by graying out the unavailable interval between 2 AM and 3 AM, and that the intent of the user is clear: a 2 hour long meeting in Los Angeles local time. Let's also assume that the application is storing enough information to preserve the local time and map it correctly to UTC and other time zones, and adjust the values individually under time zone changes. Before the meeting occurs the government makes a change in the time zone rules by adopting either daylight saving time (DST) or standard time all year long. When the meeting was originally scheduled, the local times of 1:30 AM and 4:30 AM

translated to 9:30 AM (1:30 + 8) and to 11:30 AM (4:30 + 7) in UTC respectively. If the government adopts DST all year long and the application preserves the local times, then the adjusted UTC values for the meeting will be 8:30 AM (1:30 + 7) and 11:30 AM (4:30 + 7), extending the duration of the meeting from 2 hours to 3 hours due to an earlier start in the local time. If on the other hand the government decides to abolish DST, the adjusted UTC values will be 9:30 AM (1:30 + 8) and 12:30 PM (4:30 + 8), again extending the duration of the meeting, with the end time occurring one hour later in local time. The reason for the change in the duration is that the date-time representation used doesn't take into account how various date-time values (start and end time) relate to each other.

The complexity increases exponentially with more users, in multiple time zones, using multiple applications that use different formats for communication and data storage, with inconsistent handling of time zone updates.

DESCRIPTION

This disclosure describes simple, intuitive techniques that enable developers to work with dates and times in a consistent manner free of unpredictable time or duration shifts. The disclosure defines primitive, derived, and chained date-times, notions incorporated transparently into the programming model, that result in predictable date and time behavior under updates to time-zone rules.

Per the techniques, applications that rely on time-zone information are freed from having to exchange messages to adjust their times/dates. Rather, a data structure is maintained that tracks enough information to allow for automatic adjustment of date-time values. When the time zone or DST rules change, e.g., due to legislation or fiat, the tracking date and new rules are used

to consistently adjust times throughout applications, APIs, data formats, frameworks, protocols, etc.

The techniques comprise the aforementioned data structure, an adjustment (or rebasing) procedure, a chaining procedure, and a transformation procedure, each explained in greater detail below.

- **Data structure**

```
class DateTimeEx {
  LocalDateTime baseLocalTime;
  ZoneOffset    baseOffset;
  ZoneId        baseZoneId;
  ZoneId        currentZoneId;
  Duration      currentDelta;
}
```

Fig. 1: Data structure

Fig. 1 illustrates an example of the data structure, which includes base fields and extended fields as follows.

Base fields of the data structure

The base fields of the data structure are as follows.

- `baseLocalTime`: the original local time.
- `baseZoneId`: the original time zone id.
- `baseOffset`: the offset from universal coordinated time (UTC).

In general, `baseZoneId` and `baseLocalTime` can determine time accurately in most cases, but not during certain ambiguous durations, e.g., during DST transitions (where the same local time can occur more multiple times, once before and once after the DST transition). The third base

field `baseOffset` can be used to disambiguate time during DST time transitions. The `baseOffset` is still insufficient to preserve the distances between related date-time values.

Since local time, UTC, and UTC offset are related through the formula local = UTC + offset, any two of these three values can be used in the serialized format. For example, the following combinations of values are equivalent: `baseLocalTime` and `baseOffset`; `baseLocalTime` and UTC; UTC and `baseLocalOffset`.

Extended fields of the data structure

The extended fields of the data structure are as follows.

- `currentZoneId`, the time zone of the current user, e.g., a meeting attendee.
- `currentDelta`, a field that enables the programmer to perform adjustments on the `baseLocalTime` properties consistently on all related dates and times (e.g., start and end of a meeting).

Having two time zone ids (`baseZoneId` and `currentZoneId`) enables the programmer to perform time zone conversions based on the location of the current user without losing information about the time zone originally used by the organizer to create the meeting. After making adjustments based on `baseZoneId and baseOffset`, the programmer performs the conversion to currentZoneId and addition of currentDelta to compute the current value of any date-times involved in the user flow. As explained in the section on chaining procedure below, the extended fields `currentZoneId` and `currentDelta` can represent any combination of time-zone conversions and duration increments.

For illustration purposes, let's see how the date-time values in example 3 above can be represented in a textual format. The fields basedLocalTime, baseOffset, baseZoneId, currentZoneId and currentDelta are separated by ";".

```
2021-03-14T01:30;-08:00;America/Los_Angeles;America/Los_Angeles;0
2021-03-14T04:30;-07:00;America/Los_Angeles;America/Los_Angeles;0
```

The described data structure is immutable, e.g., operations on an existing instance of the data structure create a new instance with updated `currentZoneId` and `currentDelta`, while keeping the existing structure unchanged. The new instance preserves the base fields. The fields of the data structure are unchanged when saved to persistent storage or when serialized for transmission. When an instance of the data structure retrieved from persistent storage (or over the wire) is used in applications, the fields of the data structure are adjusted to represent the correct local and absolute time based on the current time zone rules. Although the examples herein refer to calculation of the adjusted value on the fly, the adjusted value can also be made part of the data structure.

- **Adjustment procedure**

The adjustment procedure adjusts the local time and the UTC offset based on current rules for the time zone ID such that the local time stays the same if possible. In some cases, the local time is valid but the offset from UTC changed. In other cases, one local time may represent two distinct points in UTC time. An implementation can pick a default, but ideally all implementations should agree on the same defaults. The adjusted values are part of the in-memory representation, e.g., aren't persisted in storage or transmitted.

Once the base properties are adjusted, the adjusted dates and times are converted to the time zone represented by `currentZoneId` and incremented with the value of the `currentDelta`.

```
// Use base data fields to rebase the original date-time value
// and apply the accumulated transformations.
// ZonedDateTime and its methods (ofLocal, withZoneSameInstant and plus)
are “legacy” implementations. Any implementation with consistent behavior
will suffice.
  public ZonedDateTime toLegacyDateTime() {
    return ZonedDateTime.ofLocal(this.baseLocalTime, this.baseZoneId,
this.baseOffset)
        .withZoneSameInstant(this.currentZoneId)
        .plus(this.currentDelta);
  }
```

Fig. 2: Pseudocode for the adjustment procedure

Fig. 2 illustrates pseudocode for the adjustment procedure. The call to `ofLocal()` does a base adjustment to the base properties. The call to `withZoneSameInstant()` does a time zone conversion. The call to `plus()` does a time increment. The adjusted legacy date and time are returned, which can be used in places where a legacy date and time is expected, e.g., when displaying a date and time to the user.

- **Chaining procedure**

Date-times can be created in two ways.

1. The user calls a function to get the current system time, or the user specifies all the date-time components, including year, month, day of the month, hour, minute, second, and time zone. We call such a date-time primitive time.
2. An existing date-time is transformed by adding a duration (e.g., 30 minutes, 1 hour, etc.) or by converting to a given time zone.

When a date-time is created as the result of a transformation applied to an existing date-time, the existing and the new date-times are said to be chained together through these transformations. Whenever a date-time is created in such a manner, it will preserve the base

fields (`baseLocalTime`, `baseOffset`, `baseZoneId`) and update the transformation fields (`currentZoneId`, `currentDelta`) during transformations.

- **Transformation procedures: convert and add**

```
// Transformation 1. Convert to timezone represented by zoneId.
// Update the current timezone for the new instance and keep the
// rest the same.
public DateTimeEx convert(ZoneId zoneId) {
  return new DateTimeEx(
      this.baseLocalTime, this.baseOffset, this.baseZone,
      zoneId, this.currentDelta);
}
```

```
// Transformation 2. Add a duration to an existing date-time.
// Update the new duration and keep everything else the same.
public DateTimeEx add(Duration duration) {
  return new DateTimeEx(
      this.baseLocalTime,
      this.baseOffset,
      this.baseZone,
      this.currentZoneId,
      this.currentDelta.plus(duration));
}
```

Fig. 3: Illustrating the transformation procedure

Fig. 3 illustrates transformations such as time-zone conversions and the addition of durations. The illustrated operations create a new instance of `DateTimeEx` while changing only one property: `currentZoneId` or `currentDelta`. To access a legacy date-time, the programmer calls `toLegacyDateTime` (illustrated in Fig. 2). Alternatively, that value can be calculated and cached when `DateTimeEx` is instantiated.

- **Pseudo-code example**

```
ZoneId laId = ZoneId.of("America/Los_Angeles");
ZoneId spId = ZoneId.of("America/Sao_Paulo");

DateTimeEx aStart = DateTimeEx.create(2030, 3, 1, 1, 30, 0, 0, laId);
DateTimeEx aEnd = aStart.add(Duration.ofHours(4));
DateTimeEx bStart = aStart.convert(spId);
DateTimeEx bEnd = aEnd.convert(spId);
```

Fig. 4: Example of date-time chaining

In the example of Fig. 4, `aStart` is created directly, while `aEnd`, `bStart`, and `bEnd` are created by applying transformations to `aStart`. Thus, these four values are chained; a change to the America/Los_Angeles time zone rules that causes shifts to any one of them affects all of them in a distance-preserving way.

Observe that from `aStart` to `bEnd` there are two transformations: an add call to `aStart` and a convert call to `aEnd`. In more complex cases, there might be a longer chain of transformations, but no matter how long it is, the final result can be captured by the final `currentZoneId` and `currentDelta`. Together, these two properties effectively summarize any chain of transformation involving any number of time zone conversions and duration increments.

The data structure and corresponding procedures described herein can be used to augment an existing date-time implementation as a wrapper, decorator, or any appropriate design pattern. In addition to the techniques described above, passthrough (and other appropriate modifications) need to be enabled for such a wrapper or decorator.

In this manner, the techniques of this disclosure describe the notions of primitive, derived, and chained (connected) date-times, such that a chain of transformations is represented by two fields in a data structure, e.g., `currentZoneId` (e.g., zone of a meeting participant other than the organizer) and `currentDelta` (a duration). Per the techniques, date-time values can be

(de-)serialized without loss of information. Date-time values are automatically adjusted according to time-zone transition rules in effect while preserving the invariants, e.g., local time or distance between chained date-times. The techniques obviate external tools (or coordination between different processes) to adjust date-times. Temporary inconsistencies due to delay in the update of time zones self-resolve once the time zone data is updated in each system.

CONCLUSION

This disclosure describes simple, intuitive techniques that enable software developers to work with dates and times in a consistent manner free of unpredictable time or duration shifts. The disclosure defines primitive, derived, and chained date-times, notions incorporated transparently into the programming model that result in predictable date and time behavior under updates to time-zone rules.