Handling Time Zone in JavaScript

Recently, I worked on a task of adding a time zone feature to the TOAST UI Calendar, the JavaScript calendar library managed by my team. I pretty well knew that the time zone support in JavaScript is quite poor, but hoped that abstracting existing data objects would easily resolve many problems.

However, my hope was false, and I found it really hard to handle time zone in JavaScript as I progressed more. Implementing time zone features beyond simple formatting of time and calculating time data with complex operations (e.g. calendar) was a truly daunting task. For this reason, I had a valuable and thrilling experience of solving a problem leading to cause more problems.

The purpose of this article is to discuss the issues and solutions related to the implementation of time zone features using JavaScript. As I was writing this rather lengthy article, I suddenly realized that the root of my problem lied in my poor understanding of the time zone domain. In this light, I will first discuss the definition and

standards related to time zone in detail, and then talk about JavaScript.

What is Time zone?

A time zone is a region that follows a uniform local time which is legally stated by the country. It's common for many countries to have its unique time zone, and some large countries, such as the USA or Canada, even have multiple time zones. Interestingly, even though China is large enough to have multi time zones, she uses only one time zone. This sometimes results in such an awkward situation where the sun rises around 10:00 AM in the western part of China

GMT, UTC, and Offset

GMT

The Korean local time is normally `GMT +09:00`. GMT is an abbreviation for Greenwich Mean Time, which is the clock time at the Royal Observatory in Greenwich, U.K. located at longitude 0. The GMT system began spreading in Feb. 5, 1925 and became the world time standard until Jan. 1, 1972.

UTC

Many consider GMT and UTC the same thing, and the two are used interchangeably in many cases, but they are actually different. UTC was established in 1972 to compensate for the slowing problem of the Earth's rotation. This time system is based on International Atomic Time, which uses the cesium atomic frequency to set the time standard. In other words, UTC is the more accurate replacement system of GMT. Although the actual time difference between the two is tiny, UTC is whatsoever the more accurate choice for software developers.

When the system was still in development, anglophones wanted to name the system CUT (Coordinated Universal Time) and francophones wanted to name it TUC (Temps Universal Coordonn). However, none of the either side won the fight, so they came to an agreement of using UTC instead, as it contained all the essential letters (C, T, and U).

Offset

`+09:00` in `UTC+09:00` means the local time is 9 hours ahead than the UTC standard time.This means that it's 09:00 PM in Korea when it's 12:00 PM in a UTC region. The difference of time between UTC standard time and the

local time is called “offset”, which is expressed in this way: `+09:00`, `-03:00`, etc.

It’s common that countries name their time zone using their own unique names. For example, the time zone of Korea is called KST (Korea Standard Time), and has a certain offset value which is expressed as `KST = UTC+09:00`. However, the `+09:00` offset is also used by not only Korea but also Japan, Indonesia, and many others, which means the relation between offsets and time zone names are not 1:1 but 1:N. The list of countries in the `+09:00` offset can be found in UTC+09:00.

Some offsets are not strictly on hourly basis. For example, North Korea uses `+08:30` as their standard time while Australia uses `+08:45` or `+09:30` depending on the region.

The entire list of UTC offsets and their names can be found in List of UTC Time offsets.

Time zone !== offset?

As I mentioned earlier, we use the names of time zones (KST, JST) interchangeably with offset without distinguishing them. But it’s not right to treat the time and

offset of a certain region the same for the following reasons:

Summer Time (DST)

Although this term might be unfamiliar to some countries, a lot of countries in the world adopted summer time. “Summer time” is a term mostly used in the U.K. and other European countries. Internationally, it is normally called Daylight Saving Time (DST). It means advancing clocks to one hour ahead of standard time during summer time.

For example, California in the USA uses PST (Pacific Standard Time) during winter time and use PDT (Pacific Daylight Time, `UTC-07:00`) during summer time. The regions that uses the two time zones are collectively called Pacific Time (PT), and this name is adopted by many regions of the USA and Canada.

Then the next question is exactly when the summer begins and ends. In fact, the start and end dates of DST are all different, varying country by country. For example, in the U.S.A and Canada, DST used to be from the first Sunday of April at 02:00 AM to the last Sunday of October at 12:00 AM until 2006, but since 2007, DST has begun on the second Sunday of March at 02:00 AM till the first Sunday of November at 02:00 AM. In Europe, summer time is

uniformly applied across the countries, while DST is applied progressively to each time zone in the states.

Does Time Zone Changes?

As I briefly mentioned earlier, each country has its own right to determine which time zone to use, which means its time zone can be changed due to any political and/or economic reasons. For example, in the states, the period of DST was changed in 2007 because President George Bush signed the energy policy in 2005. Egypt and Russia used to use DST, but they ceased to use it since 2011.

In some cases, a country can change not only its DST but also its standard time. For example, Samoa used to use the `UTC-10:00` offset, but later changed to the `UTC+14:00` offset to reduce the losses in trading caused by the time difference between Samoa and Australia & New Zealand. This decision caused the country to miss the whole day of Dec. 30, 2011 and it made to newspapers all over the world.

Netherlands used to use `+0:19:32.13` offset, which is unnecessarily accurate since 1909, but changed it to `+00:20` offset in 1937, and then changed again to `+01:00` offset in 1940, sticking to it so far.

Time Zone 1 : Offset N

To summarize, a time zone can have one or more offsets. Which offset a country will use as its standard time at a certain moment can vary due to political and/or economic reasons.

This is not a big issue in everyday life, but it is when trying to systematize it based on rules. Let's imagine that you want to set a standard time for your smartphone using an offset. If you live in a DST-applied region, your smartphone time should be adjusted whenever DST starts and ends. In this case, you would need a concept that brings standard time and DST together into one time zone (e.g. Pacific Time).

But this cannot be implemented with just a couple of simple rules. For example, as the states changed the dates DST starts and ends in 2007, May 31, 2006 should use PDT (`-07:00`) as the standard time while Mar 31, 2007 should use PST (`-08:00`) as the standard time. This means that to refer to a specific time zone, you must know all historical data of the standard time zones or the point in time when DST rules were changed.

You can't simply say, "New York's time zone is PST (`-08:00`)." You must be more specific by saying, for instance, "New York's **current** time zone is PST." However, we need a more accurate expression for the sake of the system implementation. Forget the word "time zone". You need to say, "New York is currently using PST as its standard time".

Then what should we use other than offset to designate the time zone of a specific region? The answer is the name of the region. To be more specific, you should group **regions where the changes in DST or standard time zone has been uniformly applied** into one time zone and refer to it as appropriate. You might be able to use names like PT (Pacific Time), but such term only combines the current standard time and its DST, not necessarily all the historical changes. Furthermore, since PT is currently used only in the USA and Canada, you need more well established standards from trusted organizations in order to use software universally.

IANA Time Zone Database

To tell you the truth, time zones are more of a database rather than a collection of rules because they must contain all relevant historical changes. There are several standard

database designed to handle the time zone issues, and the most frequently used one is IANA Time Zone Database. Usually called tz database (or tzdata), IANA Timezone Database contains the historical data of local standard time around the globe and DST changes. This database is organized to contain all historical data currently verifiable to ensure the accuracy of time since the Unix time (`1970.01/01 00:00:00`). Although it also has data before 1970, the accuracy is not guaranteed.

The naming convention follows the Area/Location rule. Area usually refers to the name of a continent or an ocean (Asia, America, Pacific) while Location the name of major cities such as Seoul and New York rather than the name of countries (This is because the lifespan of a country is far shorter than that of a city). For example, the time zone of Korea is `Asia/Seoul` and that of Japan is `Asia/Tokyo`. Although the two countries share the same `UTC+09:00`, both countries have different histories regarding time zone. That is why the two countries are handled using separate time zones.

IANA Time Zone Database is managed by numerous communities of developers and historians. Newly found historical facts and governmental policies are updated

right away to the database, making it the most reliable source. Furthermore, many UNIX-based OSs, including Linux and macOS, and popular programming languages, including Java and PHP, internally use this database.

Note that Windows is not in the above support list. It's because Windows uses its own database called Microsoft Time Zone Database. However, this database does not accurately reflect historical changes and managed only by Microsoft. Therefore, it is less accurate and reliable than IANA.

JavaScript and IANA Time Zone Database

As I briefly mentioned earlier, the time zone feature of JavaScript is quite poor. Since it follows the time zone of the region by default (to be more specific, the time zone selected at the time of the OS installation), there is no way to change it to a new time zone. Also, its specifications for database standard are not even clear, which you will notice if you take a close look at the specification for ES2015. Only a couple of vague declarations are stated regarding local time zone and DST availability. For instance, DST is defined as follows: ECMAScript 2015 — Daylight Saving Time Adjustment

> *An implementation dependent algorithm using best available information on time zones to determine the local daylight saving time adjustment DaylightSavingTA(t), measured in milliseconds. An implementation of ECMAScript is expected to make its best effort to determine the local daylight saving time adjustment.*

It looks like it is simply saying, “Hey, guys, give it a try and do your best to make it work.” This leaves a compatibility problem across browser vendors as well. You might think “That’s sloppy!”, but then you will notice another line right below:

> *NOTE : It is recommended that implementations use the time zone information of the IANA Time Zone Database http://www.iana.org/time-zones/.*

Yes. The ECMA specifications toss the ball to you with this simple recommendation for IANA Time Zone Database, and JavaScript has no specific standard database prepared for you. As a result, different browsers use their own time zone operations for time zone calculation, and they are often not compatible with one another. ECMA specifications later added an option to use IANA time zone in ECMA-402 `Intl.DateTimeFormat` for international API.

However, this option is still far less reliable than that for other programming languages.

Time Zone in Server-Client Environment

We will assume a simple scenario in which time zone must be considered. Let's say we're going to develop a simple calendar app that will handle time information. When a user enters date and time in the field on the register page in the client environment, the data is transferred to the server and stored in the DB. Then the client receives the registered schedule data from the server to displays it on screen.

There is something to consider here though. What if some of the clients accessing the server are in different time zones? A schedule registered for Mar 11, 2017 11:30 AM in Seoul must be displayed as Mar 10, 2017 09:30 PM when the schedule is looked up in New York. For the server to support clients from various time zones, the schedule stored in the server must have absolute values that are not affected by time zones. Each server has a different way to store absolute values, and that is out of the scope of this article since it is all different depending on the server or database environment. However for this to work, the date

and time transferred from the client to the server must be values based on the same offset (usually UTC) or values that also include the time zone data of the client environment.

It's a common practice that this kind of data is transferred in the form of Unix timebased on UTC or ISO-8601 containing the offset information. In the example above, if 11:30 AM on Mar 11, 2017 in Seoul is to be converted into Unix time, it will be an integer type of which value is `1489199400`. Under ISO-8601, it will be a string type of which value is `2017-03-11T11:30:00+09:00`.

If you're working with this using JavaScript in a browser environment, you must convert the entered value as described above and then convert it back to fit the user's time zone. The both of these two tasks have to be considered. In the sense of programming language, the former is called "parsing" and the latter "formatting". Now let's find out how these are handled in JavaScript.

Even when you're working with JavaScript in a server environment using Node.js, you might have to parse the data retrieved from the client depending on the case. However since servers normally have their time zone synced to the database and the task of formatting is

usually left to clients, you have fewer factors to consider than in a browser environment. In this article, my explanation will be based on the browser environment.

Date Object in JavaScript

In JavaScript, tasks involving date or time are handled using a `Date` object. It is a native object defined in ECMAScript, like `Array` or `Function`. which is mostly implemented in native code such as C++. Its API is well described in MDN Documents. It is greatly influenced by Java's java.util.Date class. As a result, it inherits some undesirable traits, such as the characteristics of mutable data and `month` beginning with 0.

JavaScript's `Date` object internally manages time data using absolute values, such as Unix time. However, constructors and methods such as `parse()` function, `getHour()`, `setHour()`, etc. are affected by the client's local time zone (the time zone of the OS running the browser, to be exact). Therefore, if you create a `Date` object directly using user input data, the data will directly reflect the client's local time zone.

As I mentioned earlier, JavaScript does not provide any arbitrary way to change time zone. Therefore, I will

assume a situation here where the time zone setting of the browser can be directly used.

Creating Date Object with User Input

Let's go back to the first example. Assume that a user entered 11:30 AM, Mar 11, 2017 in a device which follows the time zone of Seoul. This data is stored in 5 integers of 2017, 2, 11, 11, and 30 — each representing the year, month, day, hour, and minute, respectively. (Since the month begins with 0, the value must be 3–1=2.) With a constructor, you can easily create a Date object using the numeric values.

```
const d1 = new Date(2017, 2, 11, 11, 30);
d1.toString(); // Sat Mar 11 2017 11:30:00
GMT+0900 (KST)
```

If you look at the value returned by `d1.toString()`, then you will know that the created object's absolute value is 11:30 AM, Mar 11, 2017 based on the offset `+09:00` (KST).

You can also use the constructor together with string data. If you use a string value to the `Date` object, it internally calls `Date.parse()` and calculate the proper value. This

function supports the RFC2888 specifications and the ISO-8601 specifications. However, as described in the MDN's Date.parse() Document, the return value of this method varies from browser to browser, and the format of the string type can affect the prediction of exact value. Thus, it is recommended not to use this method.

For example, a string like `2015-10-12 12:00:00` returns `NaN` on Safari and Internet Explorer while the same string returns the local time zone on Chrome and Firefox. In some cases, it returns the value based on the UTC standard.

Creating Date Object Using Server Data

Let's now assume that you are going to receive data from the server. If the data is of the numerical Unix time value, you can simply use the constructor to create a `Date` object. Although I skipped the explanation earlier, when a `Date` constructor receives a single value as the only parameter, it is recognized as a Unix time value in millisecond. (Caution: JavaScript handles Unix time in milliseconds. This means that the second value must be multiplied by 1,000.) If you see the example below, the resultant value is the same as that of the previous example.

```
const d1 = new Date(1489199400000);
d1.toString(); // Sat Mar 11 2017 11:30:00
GMT+0900 (KST)
```

Then what if a string type such as ISO-8601 is used instead of the Unix time? As I explained in the previous paragraph, the `Date.parse()` method is unreliable and better not be used. However since ECMAScript 5 or later versions specify the support of ISO-8601, you can use strings in the format specified by ISO-8601 for the `Date` constructor on Internet Explorer 9.0 or higher that supports ECMAScript 5 if carefully used.
If you're using a browser of not the latest version, make sure to keep the `Z` letter at the end. Without it, your old browser sometimes interprets it based on your local time instead of UTC. Below is an example of running it on Internet Explorer 10.

```
const d1 = new Date('2017-03-11T11:30:00');
const d2 = new Date('2017-03-11T11:30:00Z');
d1.toString(); // "Sat Mar 11 11:30:00
UTC+0900 2017"
d2.toString(); // "Sat Mar 11 20:30:00
UTC+0900 2017"
```

According to the specifications, the resultant values of both cases should be the same. However, as you can see, the resultant values are different as `d1.toString()` and `d2.toString()`. On the latest browser, these two values will be the same. To prevent this kind of version problem, you should always add `Z` at the end of a string if there is no time zone data.

Creating Data to be Transferred to Server

Now use the `Date` object created earlier, and you can freely add or subtract time based on local time zones. But don't forget to convert your data back to the previous format at the end of the processing before transferring it back to the server.

If it's Unix time, you can simply use the `getTime()` method to perform this. (Note the use of millisecond.)

```
const d1 = new Date(2017, 2, 11, 11, 30);
d1.getTime(); // 1489199400000
```

What about strings of the ISO-8601 format? As explained earlier, Internet Explorer 9.0 or higher that supports

ECMAScript 5 or higher supports the ISO-8601 format. You can create strings of the ISO-8601 format using the `toISOString()` or `toJSON()` method. (`toJSON()` can be used for recursive calls with `JSON.stringify()` or others.) The two methods yield the same results, except for the case in which it handles invalid data.

```
const d1 = new Date(2017, 2, 11, 11, 30);
d1.toISOString(); // "2017-03-
11T02:30:00.000Z"
d1.toJSON();      // "2017-03-
11T02:30:00.000Z"

const d2 = new Date('Hello');
d2.toISOString(); // Error: Invalid Date
d2.toJSON();      // null
```

You can also use the `toGMTString()` or `toUTCString()` method to create strings in UTC. As they return a string that satisfies the RFC-1123 standard, you can leverage this as needed.

Date objects include `toString()`, `toLocaleString()`, and their extension methods. However, since these are mainly used to return a string based on local time zone, and they return varying values depending on your browser and OS used, they are not really useful.

Changing Local Time Zone

You can see now that JavaScript provides a bit of support for time zone. What if you want to change the local time zone setting within your application without following the time zone setting of your OS? Or what if you need to display a variety of time zones at the same time in a single application? Like I said several times, JavaScript does not allow manual change of local time zone. The only solution to this is adding or removing the value of the offset from the date provided that you already know the value of the time zone's offset. Don't get frustrated yet though. Let's see if there is any solution to circumvent this.

Let's continue with the earlier example, assuming that the browser's time zone is set to Seoul. The user enters 11:30 AM, Mar 11, 2017 based on the Seoul time and wants to see it in New York's local time. The server transfers the Unix time data in milliseconds and notifies that New York's offset value is `-05:00`. Then you can convert the data if you only know the offset of the local time zone.

In this scenario, you can use the `getTimeZoneOffset()` method. This method is the only API in JavaScript that can be used to get the local time

zone information. It returns the offset value of the current time zone in minutes.

```
const seoul = new Date(1489199400000);
seoul.getTimeZoneOffset(); // -540
```

The return value of `-540` means that the time zone is 540 minutes ahead of the target. Be warned that the minus sign in front of the value is opposite to Seoul's plus sign (`+09:00`). I don't know why, but this is how it is displayed. If we calculate the offset of New York using this method, we will get `60 * 5 = 300`. Convert the difference of `840` into milliseconds and create a new Date object. Then you can use that object's `getXX` methods to convert the value into a format of your choice. Let's create a simple formatter function to compare the results.

```
function formatDate(date) {
  return date.getFullYear() + '/' +
    (date.getMonth() + 1) + '/' +
    date.getDate() + ' ' +
    date.getHours() + ':' +
    date.getMinutes();
}

const seoul = new Date(1489199400000);
const ny = new Date(1489199400000 - (840 * 60
```

```
  * 1000));

  formatDate(seoul);  // 2017/3/11 11:30
  formatDate(ny);     // 2017/3/10 21:30
```

`formatDate()` shows the correct date and time according to the time zone difference between Seoul and New York. It looks like we found a simple solution. Then can we convert it to the local time zone if we know the region's offset? Unfortunately, the answer is "No." Remember what I said earlier? That time zone data is a kind of database containing the history of all offset changes? To get the correct time zone value, you must know the value of the offset at the time of the date (not of the current date).

Problem of Converting Local Time Zone

If you keep working with the example above a little more, you will soon face with a problem. The user wants to check the time in New York local time and then change the date from 10th to 15th. If you use the `setDate()` method of Date object, you can change the date while leaving other values unchanged.

```
ny.setDate(15);
formatDate(ny);   // 2017/3/15 21:30
```

It looks simple enough, but there is a hidden trap here. What would you do if you have to transfer this data back to the server? Since the data has been changed, you can't use methods such as `getTime()` or `getISOString()`. Therefore, you must revert the conversion before sending it back to the server.

```
const time = ny.getTime() + (840 * 60 * 1000);
// 1489631400000
```

Some of you may wonder why I added using the converted data when I have to convert it back anyway before returning. It looks like I can just process it without conversion and temporarily create a converted Date object only when I'm formatting. However, it is not what it seems. If you change the date of a `Date` *object based on Seoul time from 11th to 15th, 4 days are added (* `24 * 4 * 60 * 60 * 1000`*). However, in New York local time, as the date has been changed from 10th to 15th, resultantly 5 days have been added (* `24* 5 * 60 * 60 * 1000`*). This*

means that you must calculate dates based on the local offset to get the precise result.

The problem doesn't stop here. There is another problem waiting where you won't get wanted value by simply adding or subtracting offsets. Since Mar 12 is the starting date of DST in New York's local time, the offset of Mar 15, 2017 should be `-04:00` not `-05:00`. So when you revert the conversion, you should add 780 minutes, which is 60 minutes less than before.

```
const time = ny.getTime() + (780 * 60 * 1000);
// 1489627800000
```

On the contrary, if the user's local time zone is New York and wants to know the time in Seoul, DST is applied unnecessarily, causing another problem.

Simply put, you can't use the obtained offset alone to perform the precise operations based on the time zone of your choice. If you recollect what we have discussed in the earlier part of this document, you would easily know that there is still a hole in this conversion if you know the summer time rules. To get the exact value, you need a

database that contains the entire history of offset changes, such as IANA timezone Database.

To solve this problem, one must store the entire time zone database and whenever date or time data is retrieved from the `Date` object, find the date and the corresponding offset, and then convert the value using the process above. In theory, this is possible. But in reality, this takes too much effort and testing the converted data's integrity will also be tough. But don't get disappointed yet. Until now, we discussed some problems of JavaScript and how to solve them. Now we're ready to use a well built library.

Moment Timezone

Moment is a well established JavaScript library that is almost the standard for processing date. Providing a variety of date and formatting APIs, it is recognized by so many users recently as stable and reliable. And there is Moment Timezone, an extension module, that solves all the problems discussed above. This extension module contains the data of IANA Time Zone Database to accurately calculate offsets, and provides a variety of APIs that can be used to change and format time zone.

In this article, I won't discuss how to use library or the structure of library in details. I will just show you how simple it is to solve the problems I've discussed earlier. If anyone is interested, see Moment Timezone's Document.

Let's solve the problem shown in the picture by using Moment Timezone.

```
const seoul =
moment(1489199400000).tz('Asia/Seoul');
const ny =
moment(1489199400000).tz('America/New_York');

seoul.format(); // 2017-03-11T11:30:00+09:00
ny.format();    // 2017-03-10T21:30:00-05:00

seoul.date(15).format();  // 2017-03-
15T11:30:00+09:00
ny.date(15).format();     // 2017-03-
15T21:30:00-04:00
```

If you see the result, the offset of `seoul` stays the same while the offset of `ny` has been changed from `-05:00` to `-04:00`. And if you use the `format()` function, you can get a string in the ISO-8601 format that accurately applied the offset. You will see how simple it is compared to what I explained earlier.

Conclusion

So far, we've discussed the time zone APIs supported by JavaScript and their issues. If you don't need to manually change your local time zone, you can implement the necessary features even with basic APIs provided that you're using Internet Explorer 9 or higher. However, if you need to manually change the local time zone, things get very complicated. In a region where there is no summer time and time zone policy hardly changes, you can partially implement it using `getTimezoneOffset()` to convert the data. But if you want full time zone support, do not implement it from scratch. Rather use a library like Moment Timezone.

I tried to implement time zone myself, but I failed, which is not so surprising. The conclusion here after multiple failures is that it is better to "use a library." When I first began writing this article, I didn't know what conclusion I was going to write about, but here we go. As a conclusion, I would say that it's not a recommended approach to blindly use external libraries without knowing what features they support in JavaScript and what kind of issues they have. As always, it's important to choose the right tool for your own situation. I hope this article helped you in determining the right decision of your own.