Moment.JS

Introductory Concepts

The guides area is designed to help developers learn to better interact with the date and time problem domain, and the Moment.js library. We address our most frequently seen support requests here, so it is a great place to check for solutions to any issues you may have.

The guides section is new and still under construction. If you have a request for a guide that you would like to see here, or would like to add a guide please create an issue or make a pull request in the momentjs.com repository.

Check out this scrimba moment guide if you're just starting.

Mutability

The moment object in Moment.js is mutable. This means that operations like add, subtract, or set change the original moment object. When first using Moment.js many developers are confused by scenarios like this:

```
var a = moment('2016-01-01');
var b = a.add(1, 'week');
a.format();
"2016-01-08T00:00:00-06:00"
```

As you can see, adding one week mutated `a`. To avoid situations like that, clone the moment before performing date math:

```
var a = moment('2016-01-01');
var b = a.clone().add(1, 'week');
a.format();
"2016-01-01T00:00:00-06:00"
```

Date Math vs Time Math

There is a logical difference between time math and date math.

In Moment.js time math assumes a linear time scale, just incrementing or decrementing the UTC-based timestamp by the amount of the time units provided.

Date math does not use a linear time scale, but rather increments or decrements the dates on the calendar. This is because the amount of time in a day, month, or year is variable. For example, due to daylight saving time transition, a day may be anywhere between 23 and 25 hours long. Months of course vary in number of days, and due to leap year, years vary in length as well. Date math can cause some interesting scenarios.

Due to daylight saving time, one day may not equal 24 hours:

```
//date math
moment('2016-03-12 13:00:00').add(1, 'day').format('LLL')
"March 13, 2016 1:00 PM"
//time math
moment('2016-03-12 13:00:00').add(24, 'hours').format('LLL')
"March 13, 2016 2:00 PM"
```

Due to leap years, one year may not equal 365 days:

```
moment('2016-01-01').add(1, 'year').format('LL')
"January 1, 2017"
moment('2016-01-01').add(365, 'day').format('LL')
"December 31, 2016"
```

Because of the variability of duration in day math, Moment's API does not officially support adding or subtracting decimal values for days and larger. Moment.js will accept decimal values and do its best to handle them by rounding to the nearest whole number.

As of **2.12.0** decimal day and month values use absolute value/round to convert to integers. This means that 1.5 rounds to 2, and -1.5 rounds to -2.

```
moment().add(1.5, 'days') == moment().add(2, 'days')
moment().add(-1.5, 'days') == moment().add(-2, 'days') == moment().subtract(1.5, 'days') ==
moment().add(2.3, 'months') == moment().add(2, 'months')
moment().add(-2.3, 'months') == moment().add(-2, 'months') == moment().subtract(2.3, 'months
```

Quarters and years are converted to months, and then absolute value/rounded.

```
moment().add(1.5, 'years') == moment().add(18, 'months')
moment().add(.8, 'years') == moment().add(9.6, 'months') == moment().add(10, 'months')
moment().add(1.5, 'quarters') == moment().add(4.5, 'months') == moment().add(5, 'months')
```

Time Zone vs Offset

Frequently, people are confused about the difference between time zones and UTC offsets.

A UTC offset is a value that represents how far a particular date and time is from UTC. It is expressed in the format HH:mm most of the time.

A time zone is a geographical region where all people observe a legally mandated standard time.

A time zone usually has more than one offset from UTC due to daylight saving time. Several time zones may have the same offset at some point during the year. For example, the time zones America/Chicago, America/Denver, and America/Belize all have an offset of -06:00 at varying times. For this reason, it is impossible to infer a time zone from just an offset value.

The Moment.js core library provides functionality related to adjusting times based on an offset value. It does not provide support for adjusting dates based on time zone data - this is provided by the Moment TimeZone library.

For an in depth description of this issue, see the Stack Overflow tag.

JavaScript Date

Moment.js provides a wrapper for the native JavaScript date object. In doing this, Moment.js extends the functionality and also accounts for several deficiencies in the object.

Parsing is notably unpredictable with native date. For instance, suppose I am using a computer in the United States, but I have a date in DD/MM/YYYY format.

```
var a = new Date('01/12/2016'); //December 1 2016 in DD/MM/YYYY format
//"Tue Jan 12 2016 00:00:00 GMT-0600 (Central Standard Time)"
```

There is no good work-around for this behavior with the native Date object. Moment's parser handles it just fine though:

```
moment('01/12/2016', 'DD/MM/YYYY', true).format()
"2016-12-01T00:00:00-06:00"
```

In addition, the ECMA Script 5 Specification makes an unusual assertion about the offset of ISO 8601 dates:

> The value of an absent time zone offset is "Z"

Effectively what this means is that ISO 8601 dates without an offset are to be treated as UTC values, creating the following oddity:

```
//US local format
var a = new Date('1/1/2016');
//"Fri Jan 01 2016 00:00:00 GMT-0600 (Central Standard Time)"

//ISO 8601
var a = new Date('2016-01-01');
//"Thu Dec 31 2015 18:00:00 GMT-0600 (Central Standard Time)"
```

The ES2015 spec fixes this mistake, bringing it in line with the ISO8601 specification, which specifies local time absent of offset. This is in it's own way bad as it has numerous negative back compatibility implications.

With Moment, the date is always interpreted as local time, unless you specify otherwise. This is not something that will change with the adoption of ES2015.

```
moment('2016-01-01')
//"2016-01-01T00:00:00-06:00"
```

Arithmetic is another area where the native Date object is lacking. The Date object actually provides no API for this. Instead, it relies on overflowing date values. Suppose you wanted to add 1 day to April 30, 2016. With the date object you would do the following:

```
var a = new Date('4/30/2016');
a.setDate(a.getDate() + 1);
```

This does the trick, but is somewhat unintuitive. Moment provides an API to add/subtract:

```
moment('4/30/2016', 'MM/DD/YYYY').add(1, 'day')
//"2016-05-01T00:00:00-05:00"
```

Internal Properties

Moment objects have several internal properties that are prefixed with `_`.

The most commonly viewed internal property is the `_d` property that holds the JavaScript Date that Moment wrappers. Frequently, developers are confused by console output of the value of `_d`. Moment uses a technique called epoch shifting that causes this property to sometimes differ from the actual date value that the Moment reflects. In particular if Moment TimeZone is in use, this property will almost never be the same as the actual value that Moment will output from its public `.format()` function. As such, the values of `_d` and any other properties prefixed with `_` should not be used for any purpose.

To print out the value of a Moment, use `.format()`, `.toString()` or `.toISOString()`.

To retrieve a native Date object from Moment, use `.toDate()`. This function returns a properly shifted date for interaction with third party APIs.

Parsing Guide

Moment.js has a very flexible and advanced parser that allows a huge range of functionality. The flexibility of the parser also makes it one of the most frequently misused tools of Moment.js.

This section lays out some guidelines about how to correctly use the parser in your situation.

Local vs UTC vs Offset

Moment offers three functions for parsing dates, the basic moment function, moment.utc, and moment.parseZone.

If you wish to interact with a date in the context of the user's local time, use the moment function.

```
moment('2016-01-01T23:35:01');
```

This results in a date with a UTC offset that is the same as the local computer:

"2016-01-01T23:35:01-06:00"

If you wish to interact with the date as a UTC date, use moment.utc:

```
moment.utc('2016-01-01T23:35:01');
```

This results in a date with a utc offset of +0:00:

"2016-01-01T23:35:01+00:00"

If your date format has a fixed timezone offset, use moment.parseZone:

```
moment.parseZone("2013-01-01T00:00:00-13:00");
```

This results in a date with a fixed offset:

"2013-01-01T00:00:00-13:00"

Note that if you use moment() or moment.utc() to parse a date with a specified offset, the date will be converted from that offset to either local or UTC:

This date is shifted by 8 hours, moving from +2 to -6 (the offset of the local machine)

```
moment('2016-01-01T00:00:00+02:00').format()
"2015-12-31T16:00:00-06:00"
```

This date is shifted by 2 hours, moving from +2 to UTC

```
moment.utc('2016-01-01T00:00:00+02:00').format()
"2015-12-31T22:00:00+00:00"
```

Known Date Formats

If you know the format of the date string that you will be parsing, it is always the best choice to explicitly specify that format.

Examples:

```
moment('01/01/2016', 'MM/DD/YYYY')
moment('2016-01-01 11:31:23 PM', 'YYYY-MM-DD hh:mm:ss a')
```

If your dates are in an ISO 8601 format, you can use a constant built into moment to indicate that:

```
moment('2016-01-01 12:25:32', moment.ISO_8601)
```

ISO 8601 formats include, but are not limited to:

```
2013-02-08              # A calendar date part
2013-W06-5              # A week date part
2013-02-08T09           # An hour time part separated by a T
2013-02-08 09           # An hour time part separated by a space
2013-02-08 09:30:26     # An hour, minute, and second time part
```

```
2013-02-08 09+07:00      # +-HH:mm
```

Strict Mode

Strict mode is the recommended mode for parsing dates. You should always use strict mode if your code base will allow it. More than half of the parser issues seen on GitHub and Stack Overflow can be fixed by strict mode.

In a later release, the parser will default to using strict mode.

Strict mode requires the input to the moment to exactly match the specified format, including separators. Strict mode is set by passing true as the third parameter to the moment function.

```
moment('01/01/2016', 'MM/DD/YYYY', true).format()
"2016-01-01T00:00:00-06:00"
moment('01/01/2016 some text', 'MM/DD/YYYY', true).format()
"Invalid date"
```

Separator matching:

```
//forgiving mode
moment('01-01-2016', 'MM/DD/YYYY', false).format()
"2016-01-01T00:00:00-06:00"
//strict mode
moment('01-01-2016', 'MM/DD/YYYY', true).format()
"Invalid date"
```

Scenarios fixed by strict mode:

```
//UUID matches YYYYDDD because it starts with 7 digits
moment('5917238b-33ff-f849-cd63-80f4c9b37d0c', moment.ISO_8601).format()
"5917-08-26T00:00:00-05:00"
//strict mode fails because trailing data exists
moment('5917238b-33ff-f849-cd63-80f4c9b37d0c', moment.ISO_8601, true).format()
"Invalid date"
//date has out of range value but is parsed anyways
moment('100110/09/2015', 'MM/DD/YYYY').format()
"2015-10-09T00:00:00-05:00"
//strict mode catches out of range issue
moment('100110/09/2015', 'MM/DD/YYYY', true).format()
"Invalid date"
//wrong date is parsed because non-strict mode ignores data after format
moment('2016-12-31 11:32 PM').format('LT')
"11:32 AM"
//trailing data is noticed
moment('2016-12-31 11:32 PM', moment.ISO_8601, true).format('LT')
"Invalid date"
```

Forgiving Mode

While strict mode works better in most situations, forgiving mode can be very useful when the format of the string being passed to moment may vary.

A common scenario where forgiving mode is useful is in situations where a third party API is providing the date, and the date format for that API could change. Suppose that an API starts by sending dates in 'YYYY-MM-DD' format, and then later changes to 'MM/DD/YYYY' format.

In strict mode, the following code results in 'Invalid Date' being displayed:

```
moment('01/12/2016', 'YYYY-MM-DD', true).format()
"Invalid date"
```

In forgiving mode using a format string, you get a wrong date:

```
moment('01/12/2016', 'YYYY-MM-DD').format()
"2001-12-20T00:00:00-06:00"
```

The wrong date scenario in forgiving mode is certainly less obvious to the user, but by that token could go unnoticed for a long time.

When choosing between strict and forgiving mode, it is important to consider whether it is more important that dates be accurate, or that dates never display as "Invalid Date".

Multiple Formats

Moment's parser supports specifying multiple possible formats for a date string. This can be extremely useful for situations where a date may be coming from multiple data sources. Just pass the formats as an array:

```
moment('12 March, 2016', ['DDMMMMY', 'MMMMDDY']).format()
"2016-03-12T00:00:00-06:00"
moment('March 12, 2016', ['DDMMMMY', 'MMMMDDY']).format()
"2016-03-12T00:00:00-06:00"
```

In order for this functionality to work properly, moment must parse every format provided. Because of this, the more formats that are used, the longer that parsing takes. Moment's heuristic for determining which format to use is as follows:

- Prefer formats resulting in valid dates over invalid ones.
- Prefer formats that parse more of the string than less and use more of the format than less, i.e. prefer stricter parsing.
- Prefer formats earlier in the array than later.

Warnings and Errors

There are several places where Moment.js displays deprecation warnings about functionality that will be removed in the future. Work-arounds are outlined

here.

JS Date Construction

```
Moment construction falls back to js Date.
This is discouraged and will be removed in an upcoming major release.
```

This deprecation warning is thrown when no known format is found for a date passed into the string constructor. To work around this issue, specify a format for the string being passed to `moment()`.

Define Locale Override

```
Use moment.updateLocale(localeName, config) to change an existing locale.
moment.defineLocale(localeName, config) should only be used for creating a new locale
```

This deprecation warning is thrown when you attempt to change an existing locale using the defineLocale function. Doing this will result in unexpected behavior related to property inheritance. moment.updateLocale will properly replace properties on an existing locale.

Parent Locale Undefined

Warning removed since **2.16.0**.

A locale can be defined with a parent before the parent itself is defined or loaded. If the parent doesn't exist or can't be lazy loaded when the moment is created, the parent will default to the global locale.

Locale Not Found

```
Locale <key> not found. Did you forget to load it?
```

This warning is displayed when a global locale is set but Moment cannot find it. Perhaps this locale is not bundled in your copy.

Add/Subtract

```
moment().add(period, number) is deprecated. Please use moment().add(number, period)
moment().subtract(period, number) is deprecated. Please use moment().subtract(number, perioc
```

Moment deprecated ordering the parameters of add and subtract as (period, number). Invert your parameters.

Bad:

```
moment().add('hours', 3);
```

Good:

```
moment().add(3, 'hours');
```

Min/Max

```
moment().min is deprecated, use moment.max
moment().max is deprecated, use moment.min
```

This warning is not a typo, but it is confusing.

Previous to version 2.7.0, moment supported moment().min and moment().max functions. These functions were unintuitive.

Min would return the greater of the two moments in question, and max would return the lesser.

Due to this inverted behavior, the suggestion provided in the deprecation warning is correct.

```
moment('2016-01-01').min('2016-02-01').format()
"2016-02-01T00:00:00-06:00"
//is equivalent to
moment.max(moment('2016-01-01'), moment('2016-02-01')).format()
"2016-02-01T00:00:00-06:00"
moment('2016-01-01').max('2016-02-01').format()
"2016-01-01T00:00:00-06:00"
//is equivalent to
moment.min(moment('2016-01-01'), moment('2016-02-01')).format()
"2016-01-01T00:00:00-06:00"
```

Zone

```
moment().zone is deprecated,
use moment().utcOffset instead.
```

This deprecation was made for purposes of clarity.

The result of `moment().zone()` is an integer that indicates the number of minutes that a given moment is offset from UTC, with the sign inverted (US moments result in a positive value).

Using `moment().zone(number)` to set the offset will set the offset on the date, also using an inverted sign.

Because a time zone is not the same thing as an offset, the name was changed to utcOffset. At that time the sign was corrected to reflect the actual direction of the UTC offset.

```
moment().zone()
360
//is replaced by
moment().utcOffset()
-360
```

```
moment().zone(420)
//is replaced by
moment().utcOffset(-420)
```