Current Time (from your computer)

Current Date/Time: (JavaScript)
1353920263 (Epoch)
Mon Nov 26 10:57:43 UTC+0200 2012 (Standard)

Current Date/Time: (PerlScript)
(Sorry, it doesn't appear that you have PerlScript installed.)

This section shows the output of two client-side scripting languages: JavaScript and PerlScript. Why JavaScript? (It's arguably the de facto standard for client-side scripting on the web.) Why PerlScript? (I just thought it was cool that there is a PERL interpreter for browsers, rare though it is.) I've left out VBScript, ...sorry. How the output for this section is generated is really secondary to the core date-to-epoch and epoch-to-date conversion topic. It's relevance in this example is that DHTML techniques using a client-side scripting language is what updates this webpage with your input without a reload from the server.

What is Epoch

Epoch has a few meanings (see also http://dictionary.reference.com/search?q=epoch). The definition that we'll use is "0" in computer time. While there are folks who will argue this, for our purposes, this "0" time on our calendar was January 1, 1970 00:00:00 GMT. Epoch is useful in the programming world because it allows us to mathematically compare dates with other dates or some other measure of time. Said another way, it allows us to use a programming algorithm to make decisions regarding a human readable date or time. See the examples of usage section below for more detail.

How to Convert between Date and Epoch

I've had a number of inquiries into how to do the conversion. I figured folks interested in that level of detail would use the source code for this page to ascertain the approach used here. However, I suppose assuming everyone enjoys deciphering the idiosyncrasies of JavaScript for fun was a bit naive... my bad. I'll take a stab at explaining what the code does.

First off, the easiest way to get the current time in epoch (using JavaScript), is to call getTime() method of the JavaScript Date object and divide the return value by 1000. getTime returns the number of milliseconds elapsed, in your computer's timezone, since 1/1/1970 GMT. Because epoch is measured in seconds, you then want to divide the return value by 1000 to change milliseconds into seconds. Keep in mind that the JavaScript Date object will get it's time from your computer's clock. Here's a really simple example doing this.

Note: We're subtracting the milliseconds in order to get a whole number instead of a fraction. You could probably use parseInt similarly, in fact, here's another mini-example demonstrating the differences between (a) getTime()/1000 alone -- sometimes a fraction is returned, (b) method above, (getTime - milliseconds)/1000, and (c) parseInt(getTime/1000).

"Tools are all well and good, but I like pain, how do you do this manually"

Er ok... if you have a human friendly date/time that you want to convert, but you are shall we say "tool averse" or you just enjoy mental gymnastics (good for you, btw), I'll take a stab at explaining a process for doing this manually.

Keep in mind that an accurate description of what the JavaScript Date Object methods are doing isn't in scope--rather my effort to explain will be conceptual. (So easy even a Geico customer could do it, my caveman friend likes to say ;-)

You could do a rough calculation as follows:

For kicks and giggles let's use for our example a day that many folks will recognize, April 15, 2013. And, we'll say it's 08:30 in the morning just to add some spice to this effort.

Years

Start with the year: 2013. Subtract out 1970 from the year. 2013 - 1970 leaves us with 43 years.

Convert the years to days. So 43 years multiplied by 365 days in each of those years. This gives us 15695 days.

Now the first relatively difficult issue--dealing with leap years. Like me, you may have had it explained to you that roughly every four years can be a leap year...with some caveats. Actually, (as a helpful individual has pointed out to me) the real rule is as follows: Years that are evenly divisible by 100 are not leap years, unless they are also evenly divisible by 400, in which case they are leap years, bringing the total number of days in that year to 366.

We need to examine each of the 43 years between our target and the 1970 starting point. (This is one reason a tool is so great. You don't have to brute force the effort.) Writing a quick script to determine the calculation...

The number of extra days due to leap years between 1970 and 2013 is [11] days. Leap years in the range are: 2012, 2008, 2004, 2000, 1996, 1992, 1988, 1984, 1980, 1976, 1972

Want to sanity check whether a specific year is a leap year? (Uses algorithm in isLeapYear function code example above to calculate leap year.)

So, we need to add 11 to our original 15695 which brings us to 15706 total days so far.

Months in Days

Now we need to factor in the months in the current year and convert to days. In our example, we're using April 15, so we have 15 days for the current month. January has 31 days, February has 28--unless this year is a leap year in which case it has 29. If this is a leap year, we caught it in our leap year check above. March has 31 days. Adding these up we get 15 + 31 + 28 + 31 = 105 days. Adding that back in to our running total of days we get 15706 + 105 = 15811

Now that we've got all of the days, let's convert those to seconds. It turns out that there are roughly 86,400 seconds in each day. (See the chart below for more useful conversions.) So we take 15811 and multiply it by 86400 to get 1366070400 seconds which have elapsed up until the beginning of April 15.

Hours, Minutes, ...Seconds?

Now we need to factor in the hours and minutes that have passed in the current hypothetical day. It's 08:30. In each minute there are 60 seconds, in each hour there are 60 minutes, so we convert the 8 o'clock hour as 8 * 60 * 60. This gives us 28800. Further, we'll convert the 30 minutes since the 8 o'clock hour into seconds, 30 * 60 which gives us 1800. And we add those all back into the seconds so far to get 1366101000

Add a dash of craziness...

Lastly, we need to calculate our offset from GMT. Er, and then factor in daylight savings (ugh). In my case, PST, we're 8 hours different--and we participate in daylight savings. So I add (8 * 60 * 60) + (1 * 60 * 60) which gives 32400, back into my running seconds to get my local time. 1366133400

This should be the epoch time we're looking for. (See sanity checking your results below.) So, we're saying 1366133400 have elapsed since 1/1/1970--"0" in computer time. It's not very friendly from a human standpoint, but it's pure gold from a computer programming standpoint.

Sanity checking your results

Plug 1366133400 back into the converter and see how far off March 15, 2006 you are. It's likely that you are off by some amount, probably the offset for your own local timezone. In general, you should always sanity check your results. Two good tests are, checking your results against today (using the current epoch from getTime/1000 for example), and "0" time. If your mechanism passes both tests, that's a good sign.

Another interesting note: I've heard folks suggest using UTC rather than local time. UTC is certainly an appropriate "global" approach to time based applications, particularly when your audience spans the globe; In that case, having a standard time to fallback on is excellent. In order to apply it to your specific locale, you would calculate the target timezone offset from UTC. There are some strange issues that come up. Calculating an offset accurately can be really tough in the real world. For example, Arizona time while correct in Windows, isn't correct in JavaScript. And that's just one example. I'm not an expert on timezones, so I don't know how prolific timezone offset problems are. I'd appreciate feedback from anyone in the know.

Examples of Usage

Of course, the most relevant question in all of this is "why would I want to do any of this?" I've had the need for this arise many times. One notable example which occurred while working for a web hosting company was that we wanted to turn messaging chunks on/off 3 days after a new signup for services. We had a marketing manager that wanted to play some messaging to our audience of webmasters three days into their experience. Because there is a steady inflow of customers, the exercise is an ongoing one, and relative to each specific account. We needed a way to identify a member of the 3+ day audience and then some logic to trigger the messaging. In turning that need into programming routines, we did the following steps.

1. **First, we stored the epoch value of the date of each signup (for example, Jan 1, 2003 is 1041459471 in epoch). Now that the date value is an integer, we can do math with it.**
2. **Second, we calculate 3 days in epoch measures (seconds) as follows... (86400 * 3)**
3. **Third, compare the current time with the signup time, plus the 3 days epoch value.**

By making signup date a variable, this rule can apply to anyone for whom we want to know 3 days into the future of their signup.
ex: $signup_date = 1041459471 (or whatever the epoch is for any given site)

```
if ($current_time > ($signup_date + (86400 * 3))) {
  /*  ...then this site was signed up more than 3 days ago
        show them the relevant message
  */
}
```

Common Time Measures in Epoch

1 Millisecond = 1/1000 in Epoch (used frequently with JavaScript conversions)
1 Second = 1 in Epoch
1 Minute = 60 in Epoch
10 Minutes = 600 in Epoch
1 Hour = 3600 in Epoch
6 Hours = 21600 in Epoch
12 Hours = 43200 in Epoch
1 Day = 86400 in Epoch (24 Hours)
1 Week = 604800 in Epoch (7 Days / 168 Hours)
1 Month = (see below... number of days in the month)
28 Days = 2419200 in Epoch (672 Hours)
29 Days = 2505600 in Epoch (696 Hours)
30 Days = 2592000 in Epoch (720 Hours)
31 Days = 2678400 in Epoch (744 Hours)
AVERAGE MONTHS IN YEAR = 2628000 in Epoch
(365 / 12 = 30.41666666666666666666666666666667) (average months)
(30.41666666666666666666666666666667 * 86400 = 2628000) (average months in Epoch)
1 Year = 31536000 in Epoch (365 days * 86400 seconds)