Time Expressions

Time expressions are simple, compact, textual representations of time- and date-based schedules. Time expressions are used in several places throughout Flux (including engine configuration options as well as trigger and action properties) to specify when and how often particular tasks will execute on the Flux engine.

Time expressions make it easy to instruct workflows and events to run at specific times, on a recurring basis, or even relative to other times (or relative to the last run of the same event).

In Flux, there are two different types of time expressions:

- *Cron-style* time expressions typically model schedules that occur at a specific time, date, or frequency, according to a set of constraints. For example, a typical use case for a Cron-style time expression might be an event that needs to occur every 30 minutes from 08:00 to 17:00. Cron-style time expressions are modeled after Unix Cron (Flux's Cron-style time expressions are, however, much more powerful and expression than Unix Cron).
- *Relative* time expressions model schedules where each event is scheduled relative to a certain time or date, or relative to the time and date of the previous event. Relative time expressions are especially useful for schedules that Cron-style expressions cannot express: for example, use cases for a Relative time expression might be an event that needs to run on the first Monday after the previous event, or an event that must run on election day in the United States (the first Tuesday after the first Monday in November).

This sections describes the general syntax for using time expressions in Flux, and provides some specific examples to demonstrate how time expressions can be used in your scheduling.

Cron-style time expressions

Fundamentally, a Cron-style time expression specifies dates and times according to a set of constraints. These constraints determine when an event (like a trigger fire) may execute.

The Cron-style time expression itself is a series of eleven columns. Each column represents one of the possible constraints on the expression. The columns are separated by spaces (no other spaces are allowed in the expression – a space can only be used to separate one column from the next). A column may contain a particular value or range (as appropriate for the constraint it represents), or it may contain the '*' character indicating that the column is completely unconstrained. A totally unconstrained column, therefore, would look like:

```
* * * * * * * * * * *
```

In order (from left to right), each column represents the following corresponding constraint:

milliseconds	seconds	minutes	hours	days-of-month	months	days-of-week	day-of-year	week-of-month	week-of-year	year

If a column is unconstrained (using the '*' character) then it will fire at every possible instant represented by the column. A completely unconstrained Cron-style expression, therefore, would fire at every millisecond continuously, so naturally it is not practical to use a Cron-style expression without applying some constraints.

Each column has its own range and type of accepted values. The table below shows the valid range for each column:

Column	Values
Milliseconds	0-999
Seconds	0-59
Minutes	0-59
Hours	0-23
Days-of-month	1-31 *or* <position><weekday of month> (see Special Values for the Days of Month and Day of Year Columns below for more information).
Months	0-11 or jan-dec

Days-of-week	1-7 or sun-sat
Day-of-year	1-366 *or* <position><weekday of year> (see Special Values for the Days of Month and Day of Year Columns below for more information).
Week-of-month	*minimum*-6, where *minimum* is either 0 or 1 (depending on your locale). In the United Sates, the minimum value is 1.
Week-of-year	*minimum*-53, where *minimum* is either 0 or 1 (depending on your locale). In the United Sates, the minimum value is 1.
Year	1970-3000

For a practical example, consider a workflow that needs to run at 12:00 (noon) every weekday. In this case, the Cron-style time expression must be constrained, to provide some limitation on when the expression can fire.

The milliseconds, seconds, and minutes will be constrained to 0 (so the expression will only fire when each of those reaches the "0" value, which would occur at the top of every hour). Because the expression must fire at noon, the hours column is constrained to 12. Days-of-month and months can be totally unconstrained as the expression is not concerned with the month or the position in the month. Days-of-week, however, must be constrained to weekdays: Monday through Friday. A simple constraint of "mon-fri" will accomplish that goal. The day-of-year, week-of-month, week-of-year and year can also be unconstrained as they are not required for this expression.

After applying all these constrains, you will end up with an expression that looks like:

```
0 0 0 12 * * mon-fri * * * *
```

As you can see, the format of a Cron-style time expression is a series of columns, beginning with the milliseconds column and continuing through the rest of the columns. Note that it is not necessary to specify values for all of the columns if you are not using them all – you only need to specify values, left-to-right, up to the last column that you need to use. If you do need a value in a certain column, however, it is required to specified some value (or the unconstrained value '*') for each column to the left of that column.

For example, consider a task that needs to run at 17:00 every day. You could use the following expression for this, and take advantage of the fact that only the columns up to the last column you use are required:

```
0 0 0 17
```

Because the hours column is the furthest-right column that is used, only the columns to the left of it must be set.

Similarly, if you wanted an expression to fire only during the 17th hour of the day, but you did not need to constrain the minutes column, you could use an expression like:

```
0 0 * 17
```

As noted, the value **'' in any column will include the entire range. Every value in the range will be included in the expression (keep in mind, however, that this is still restricted by the other constraints on the expression – so although a ''** in the minutes column will fire for every possible minute value, if the hours are restricted then it will only fire for minute values on the included hours).

Each column can use specific values as appropriate for the particular column. You can also extend the range of values included in each column by applying the following concepts:

- Specify multiple specific values for a column by separating each item with a comma. For each, "5,10,15" in the minutes column would include the 5th, 10th, and 15th minute.
- Use the '-' character to specify a range. For example, "5-15" in the minutes column would include all of the minutes beginning with the 5th and ending with the 15th. Ranges are inclusive, so the first and last values from the range are also picked up (meaning that 5-15 in the minutes column would pick up the 5th minute, the 15th minute, and all minutes in-between).
- Combine the comma and range concepts to specify multiple ranges. Setting "5-7,10-12" in the minutes column would pick up the range of minutes from 5th to 7th *and* the range from 10th to 12th.
- Ranges and commas will also work for non-numeric values. For example, "tue-sat" in the days-of-week column would include all days beginning on Tuesday and ending on Saturday.
- Use "step" values to specify values for a column that occur at a specific frequency. The step is apply by ending the value in */n*, where the column will accept every *nth* possible value. For example, setting "*/4" in the minutes column will include every fourth possible minute. This would include the minutes 0, 4, 8, 12, etc. You can also combine the step with a range – setting "3-9/3", in the minutes column, for example, would include the 3rd, 6th, and 9th minute.
- Use the '+' character to specify a relative value for a column. For example, setting "+30" in the minutes column will include every 30th minute.

As listed above, some columns can use textual values in addition to numeric values. Acceptable values for each column are listed below:

- **Months**: jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec. Each months corresponds to its numeric value, beginning with jan (month 0) and ending with dec (month 11).
- **Days of week**: sun, mon, tue, wed, thu, sat. Each day of week corresponds to its numeric value, beginning with sun (day of week 1) and ending with sat (day of week 7).

Special Values for the Days of Month and Day of Year Columns

In addition to specifying numeric values for the days of month and day of year columns, you can set special weekday values. This allows to specify items that occur in a specific position in the month or year, where the specific numeric value could change from month to month or year to year. Specific examples might include "the second Monday of each month" or "the last Friday of the year".

These special values have the syntax <position><weekday abbreviation>. The possible position values are:

Value or Range	Meaning
1-6 (days of month) or 1-53 (day of year)	Run on the specific number value for the given weekday (see table below). For example, setting "1MO" in the days of month column will include the first Monday of the month. Likewise, "44FR" in the day of year column will include the 44th Friday of the year.
^	Run on the first possible instance of the given weekday. For example, "^TU" in the days of month column will include the first Tuesday of the month. Similarly, "^WE" in the day of year column will include the first Wednesday of the year.
$	Run on the last possible instance of the given weekday. For example, "$TH" in the days of month column will include the last Thursday of the month. Similarly, "$SA" in the day of year column will include the last Saturday of the year.

The possible weekday abbreviations values are:

Abbreviation	Day of Week
SU	Sunday
MO	Monday
TU	Tuesday
WE	Wednesday
TH	Thursday
FR	Friday
SA	Saturday

Note that these abbreviations are different than those used for the days of week column. Take care to ensure that you are using the appropriate abbreviations in each column.

Applying Business Intervals in Cron-style Expressions

Many properties that use time expressions on triggers and actions (like the time expression setting on a timer trigger, or the timeout property available on all triggers and actions) can be combined with Business Intervals to express more complicated schedules.

When you define a business interval, you create a calendar of business dates and hours. To actually use the business interval, though, you must combine it with a time expression that specifies how the interval is applied.

To help with this, Cron-style time expressions accept two special symbols:

Symbol	Explanation
b	A business unit of time – that is, a unit that is *included* in the business interval.
h	A non-business unit of time – that is, a unit that *excluded* from the business interval.

Using one of these special characters in a column of the Cron-style time expression indicates to Flux, "use only the values in this column that are included (or excluded) from the business interval).

These special characters must be preceded by an identifier, to describe how the character should be applied (i.e., "the first unit of time for this column if it is allowed", "the 5th unit of time if it is not allowed"). The identifiers that can be used are:

Value	Affect
*	Include all business units of time (using the 'b' character) or non-business units of time (using the 'h' character) for this column. For example, setting "*b" in the days of month column will include all days of the month that are included in the business interval. Likewise, "*h" in the days of month column will include only days of the month that are excluded from the interval.
Numeric value *n* (where *n* is 1, 2, 3, etc.)	Include the *nth* business unit of time (using the 'b' character) or non-business until (using the 'h' character). For example, setting "2b" in the days of month column would include the second business day of the month. Likewise, setting "3h" in the days of month column would include the 3rd non-business day of the month.

^	Include the first business (or non-business) unit of time for this column. A value of "^b" in the days of week column would include the first day of the week allowed by the business interval. Similarly, "^h" in the days of week column would include the first day of week *not* allowed by the business interval.
$	Include the last business (or non-business) unit of time for this column. A value of "$b" in the days of week column would include the last day of the week allowed by the interval, and a value of "$h" would include the last day of the week *not* allowed.

These can be combined with other concepts for Cron-style time expressions, including:

- **Ranges**: Similar to a normal range, but allows to limit the included values to only those included or excluded by your interval. For example, you might use "5b-10b" in the days of month column to include the 5th business day through the 10th business day of the month (skipping non-business days), or "3h-5h" to include the 3rd non-business day through the 5th non-business day.
- **Steps**: Business and non-business notation can be used on both sides of the step. For example, you could use "5b-9b/2" in the days of month column to say "fire every second calendar day between the 5th and 9th business days of the month", or "5-20/2b" to say "fire every second business day between the 5th and 20th calendar days of the month". These can also be combined: for example, you might use "5b-15b/2b" to say "fire every second business day from the 5th to the 15th business days of the month".

Note that the 'b' and 'h' characters are ignored if you do not have a business interval defined.

Shift and Rollover Operators

Cron-style time expressions also support "shift" symbols. Shifting can shift backwards or forwards. This functionality allows tasks to run at moments relative to other moments. For example, to run tasks on the second-to-last day of the month, use the expression "$<1" in the Days-of-month column, which means one day before the last day of the month.

Shifting can be employed for other useful purposes. For instance, to run a task on the first business day on or after the 10th calendar day of the month, use "10>1b" in the Days-of-month column. If the 10th calendar day of the month falls on a holiday, then the ">1b" portion will "push" the expression ahead to the next business day. On the other hand, if the 10th calendar day of the month is already a business day, then the ">1b" portion does nothing, because the 10th calendar day of the month is already a business day.

To run a task on the second business day on or after the 10th calendar day of the month, simply use "10>2b".

To run tasks on the second-to-last *business* day of the month, use a slightly different expression, "$<2b" or "$b<2b", both of which achieve the same result. The expression "$b<1b" is equivalent to "$b". To run tasks on the second-to-last *calendar* day of the month, use "$<1". This expression means "the last calendar day of the month, less one day". In summary, "<1b" and ">1b" are no-ops if the current day is already a business day, but "<2b" and ">2b" are guaranteed to shift by at least one day. This behavior makes it possible run tasks *on or after* certain days of the month.

You can also use the shift operators to run that are a few calendar days after a certain day. For example, to run a task 5 calendar days after the 6th business day of the month, use "6b>5" in the Days-of-month Cron column. As another example, to run a task on election day in the United States, which is the first Tuesday after the first Monday in November, use "1MO>1TU" in the day-of-month Cron column. This expression, "1MO>1TU", says to go to the first Monday of the month, then advance to the following Tuesday. In general, you can shift left or right by a number of days of the week. For example, you can shift right to the third Friday after a certain moment by using the expression ">3FR".

In general, all of these techniques and symbols can be used in any of the Cron-style time expression columns, not just Days-of-month and Day-of-year columns.

The rollover operator (">>") provides similar functionality to the right shift operator (">"). The difference between rollover and right shift is that when the rollover operator attempts to satisfy the constraints of the Cron-style time expression, it can "rollover" into the next Cron column on the right in search of a match, unlike the right shift operator.

For example, suppose you need to fire a task on the first Monday, at midnight, on or after the 28th calendar day of the month. Using a right shift operator, the Cron-style time expression is shown as follows.

```
0 0 0 0 28>1MO
```

Using the above time expression, this task fires at midnight on every month that has a Monday on or after the 28th calendar day of the month, such as the 28th, 29th, 30th, or 31st calendar day of the month. This time expression can be satisfied in some months, such as September 2003, when the 28th calendar day of the month falls on Sunday.

However, suppose that the first Monday of the month is not until the 3rd calendar day of the following month, such as the case in October 2003. In October 2003, the 28th calendar day of the month is a Tuesday. The first Monday after the 28th does not occur until Monday, 3 November 2003.

In October 2003, this task will not fire using the above Cron-style time expression and the right shift operator. The reason is simple. When the Cron-style time expression searches for a matching date and time, the day-of-month column constraint requires that the month column constraint stays fixed. Consequently, only month values that are explicitly specified are considered.

However, if you want this task to always fire on the first Monday on or after the 28th calendar day of the month, even if the time expression needs to "rollover" into the next month, the rollover operator (">>") performs this function.

For example, the following Cron-style time expression always fires on the first Monday on or after the 28th calendar day of the month.

```
0 0 0 0 28>>1MO
```

In October 2003, the 28th calendar day of the month is a Tuesday. The first Monday after the 28th does not occur until Monday, 3 November 2003. Therefore this task fires on Monday, 3 November 2003. The task next fires on Monday, 1 December 2003. The task next fires on Monday, 29 December 2003, because in December 2003, the 28th calendar day of the month falls on Sunday, so the first Monday is simply the next day.

The right side of the rollover operator accepts numbers, followed by an optional "b" symbol, "h" symbol, or a two letter day-of-week abbreviation.

Using the rollover operator, you can specify firing times that contain both a "fixed" component, followed by a "variable" component, which searches as far as necessary into other Cron columns in order to satisfy the constraints of the Cron-style time expression.

The "h" symbol is the opposite of "b". It means holidays or non-business days. It can be used anywhere "b" can be used and has the opposite meaning of "b".

Each column normally accepts a number, a range of numbers, or a comma-separated list of numbers and ranges of numbers. However, relative increments such as "+5" can be specified also. For example, "+15" in the Minutes column means that the task should fire every 15 minutes. Note that this requirement is subtly different than "*/15", which means to fire when the minute hand is on the 0, 15, 30, and 45 number on the clock.

As a further example of using relative increments such as "+5", you can specify that a task should run on every 10th Wednesday. To specify this requirement, place "WED" in the days-of-week column and "+10" in the week-of-month or week-of-year column. In this case, the task will fire on a Wednesday, every 10 weeks.

Note that these Cron-style time expressions are slightly different than traditional Unix-style Cron specifications. In Unix, the time range goes down to the minute. Here, the time range goes down to the millisecond.

Furthermore, in Unix, the months range from 1-12. Here, the months range from 0-11. Similarly, in Unix, the days-of-week range from 0-7, starting with Sunday. Sunday is associated with the numbers 0 and 7. Here, the days of the week range from 1-7, starting with Sunday. Sunday is associated with the number 1 and Saturday with the number 7. The reason for these differences is that the scheduler is consistent with Java. In Java, the months range from 0-11 and the days-of-the-week range from 1-7.

"Or" Constructs

Cron-style time expressions also permit running tasks at two or more times that are unrelated to each other. For example, to run a task at 10:15 am and 3:35 pm, you can use the "Or" construct in Cron-style time expressions. For example:

```
0 0 (15 10 | 35 15) * * *
```

Notice how the third and fourth Cron columns, Minutes and Hours, are grouped. This group contains two elements, "15 10" and "35 15", which means that the task can fire at 10:15 am or 3:35 pm. These "Or" constructs can accept two or more columns in each group. In fact, you can take an indefinite number of fully-specified Cron-style time expressions and group them together using multiple "|" symbols, also known as "Or" constructs.

For example, the following three Cron-style time expression, which are completely unrelated to each other, can be grouped together to form a new Cron-style time expression that fires when any of the three individual Cron expressions would fire.

- 0 0 30 8-16 * * mon-fri // Fires on the half hour.
- 0 0 0 * * * sat,sun // Fires at the top of the hour.
- 0 0 15 * * * * // Fires on the first quarter hour.

The result of "Or"ing the above three Cron expressions is shown below. This Cron expression fires on the half hour, at the top of the hour, and on the first quarter hour.

```
(0 0 30 8-16 * * mon-fri | 0 0 0 * * * sat,sun | 0 0 15 * * * *)
```

Note: The "Or" construct cannot contain nested "for" loops or "Or" constructs. Flux does not support "for" loops or additional "Or"s within the "Or"ed Cron expressions.

For Loops

Furthermore, Cron-style time expressions permit running tasks from a beginning point in time, followed by task firings at regular intervals, up to an ending point in time. Analogous to the "for" loop language construct in the Java programming language, Cron-style time expressions have a "For Loop" construct.

For example, suppose you need to fire tasks from 9:15 am through 4:30 pm, every 15 minutes, Monday through Friday. The following Cron-style Time Expressions uses a "For Loop" to describe this desired firing pattern:

```
0 0 (15 9; 30 16; */15 *) * * *
```

The "For Loop" contains "(15 9; 30 16; */15 *)". Like the Java programming language "For Loop", the loop starts at 9:15 am, continues until 4:30 pm, and fires when the minute hand is on 0, 15, 30, or 45 and when the hour hand is on any number of the clock. In general, the first component of the "For Loop" is called the "Start Constraint". The second component is called the "End Constraint", and the third component is called the "Increment Constraint". Notice how these names and this behavior is very similar to Java "For Loops".

There are some variations on the "For Loop" that you can use. Instead of firing when the hand is on 0, 15, 30, and 45, you can fire every 15 minutes, which is subtly different:

```
0 0 (15 9; 30 16; +15 *) * * *
```

Finally, you can omit the End Constraint and insert a fourth constraint at the end, called the count:

```
0 0 (15 9; ; */15 *; 5) * * *
```

The above Cron-style time expression fires at 9:15 am, 9:30 am, 9:45 am, 10:00 am, and 10:15 am. After the 5th firing, the "For Loop" breaks out and the Cron-style time expression seeks out the next time in the future that can be satisfied.

Using For-Loops to Condense Multiple Timer Triggers

This section describes how you can condense several timer triggers into one using the for-loop construct in Cron-style time expressions.

First of all, here is a typical Cron expression for running a task every day from 10:00 through 11:45, on the quarter hours:

```
0 0 */15 10-11 * * *
```

Reading left to right, the milliseconds and seconds columns are on 0, and the minutes are 0, 15, 30, 45. The hours are 10 and 11. For the days-of-month, months, and days-of-week, a wildcard is shown, meaning the task can fire on any day-of-the-month, month, or day-of-the-week.

This Cron expression means that your TimerTrigger will fire at 10:00, 10:15, 10:30, ..., 11:30, and 11:45.

The for-loop constructs add to this functionality. First consider this typical use case: "Fire from 10:15 through 12:30."

By using the technique above, you would observe extra firings at 10:00 and 12:45. Without "for loops", you would have to break the Cron expression into three different tasks:

```
0 0 15,30,45 10 * * *
0 0 */15 11 * * *
0 0 15,30 12 * * *
```

This technique of using three different tasks will work, of course, but it is cumbersome, because you need three different Cron expressions, three different TimerTriggers, and three different tasks.

The for-loop construct solves this problem. It is patterned after the Java "for loop", which is well understood. Consider this "for loop" syntax:

```
0 0 (15 10; 30 12; */15 *) * * *
```

The first two columns are the same as the first two columns in each of the three Cron expressions listed above: 0 0. Next, notice the parenthesized part. This part is very similar to a Java "for loop". A Cron for-loop specifies a repeating pattern of timing points.

The first part, "15 10", says to start the for-loop firing at 10:15. The "15 10" part represents the minutes and hours columns in a Cron expression.

The second part, "30 12", represents the ending point. This for-loop firing will stop at 12:30.

Finally, the third part, "**/15 *", is the "increment" -- just like in a Java for-loop. In Java, the increment is usually something like "i++". Here, it is "**/15 *". This increment says to fire between the start and end dates, whenever the minute hand is on the 0, 15, 30, and 45 and when the hour hand is on any legal hour.

The effect of the Cron for-loop is you observe the following firing pattern:

10:15, 10:30, 10:45, 11:00, 11:15, 11:30, 11:45, 12:00, 12:15, and 12:30

The last three columns of the Cron expression, "* * *", have the usual meaning.

The great part about the Cron for-loop is that you can condense multiple tasks into just a single Cron expression. There is no need for three Cron-expressions, three TimerTriggers, and three tasks. Just one of each will do.

Here is another example. In this case, we're scheduling a task to execute every 15 minutes from 20:15 to 23:15, Monday to Friday.

```
(0 0 15,30,45 20-22 * * MON-FRI | 0 0 0 21-23 * * MON-FRI | 0 0 15 21-23 * * MON-FRI)
```

If you don't want to use the string Cron syntax directly, you can use the Cron helper object to generate your Cron expression. The Cron object API can generate any legal Cron expression. It doesn't matter whether you prefer to write out your Cron expressions using strings or whether you use the Cron API. At runtime, it's all the same.

Relative time expressions

Relative time expressions specify an offset, relative to a particular point in time.

As with Cron-style time expressions, there is a helper object, *Relative*, that can be used to create Relative time expressions, if desired. A Relative object is obtained by calling EngineHelper.makeRelative(). By using the Relative helper object, the exact syntax of Relative time expressions does not need to be learned or memorized. However, it is useful to read and understand the syntax.

Examples include:

- +90s (90 seconds in the future)
- -45s (45 seconds in the past)
- +2H+30m+15s (2 hours, 30 minutes, 15 seconds in the future)
- >thu (forward to next Thursday)
- >2sat (forward 2 Saturdays)
- >oct (advance to October)
- ^M (go to the beginning of the current month)
- @fri (go to the first Friday of the current month)
- @2fri (go to the second Friday of the current month)
- ^M>3fri (go to the third Friday of the current month)
- +M^M>2tue (go to the second Tuesday of next month)
- @fri (go to the first Friday of the current month)
- @!fri (go to the last Friday of the current month)
- @22d (go to the 22nd day of the current month)
- @3H (go to 3 AM on the current day)
- >b (advance to the next business day)
- +M^M>b@9H (advance to 9 AM on the first business day of next month)
- <b (back up to the previous business day)
- ?sat{-d}?sun{+d} (if today is Saturday, back up to Friday but if today is Sunday, advance to Monday}

Relative time expressions consist of the following syntax. They may be combined to form more complex time expressions.
Relative time expression Syntax| + <number> <unit> | Move forward in time by *number* years, months, weeks, days, hours, minutes, seconds, or milliseconds. If *number* is omitted, it defaults to 1. |

- <number> <unit>	Move backward in time by *number* years, months, weeks, days, hours, minutes, seconds, or milliseconds. If *number* is omitted, it defaults to 1.
> <number> <unit>	Move forward in time by *number* absolute days-of-the-week, absolute months, holidays, non-holidays, weekdays, weekend days, or business days. If *number* is omitted, it defaults to 1. If date is already on the given day-of-week or month, then *number* is first decremented by 1.
< <number> <unit>	Move backward in time by *number* absolute days-of-the-week, absolute months, holidays, non-holidays, weekdays, weekend days, or business days. If *number* is omitted, it defaults to 1. If date is already on the given day-of-week or month, then *number* is first decremented by 1.
^ <unit>	Go to the beginning of the current year, month, week, day, hour, minute, second, or millisecond. Going to the beginning of one time unit (such as hour) also causes any sub-units (such as minute, second, and millisecond) to go to the beginning.
$ <unit>	Go to the end of the current year, month, week, day, hour, minute, second, or millisecond. Going to the end of one time unit (such as hour) also causes any sub-units (such as minute, second, and millisecond) to go to the end.
@ <n> <unit>	Go to the *nth* year, month, day-of-month, hour, minute, second, or millisecond. Recognized units are *y, M, d, H, m, s, and S.*
@ <n> <day>	Go to the *nth* day-of-week in the current month. *n* is 1-based. If *n* is omitted, go to the first day-of-week in the current month. Recognized day-of-week values are *sun, mon*, *tue*, *wed*, *thu*, *fri*, and *sat.*
@ ! <day>	Go to the last day-of-week in the current month.
? <unit>{<then-time-expression>} {<else-time-expression>}	If the current time occurs on the given unit (day-of-week, month-of-year, business day, etc), apply the *then-time-expression*. Otherwise, apply the optional *else-time-expression*. These conditional time expressions may be nested.

The following units are recognized.
Recognized Units| **Unit** | **Description** |

y	Year
M	Month
w	Week

d	Day
H	Hour
m	Minute
s	Second
S	Millisecond
h	Holiday
n	Non-holiday
D	Weekday: Monday through Friday
e	Weekend day: Saturday through Sunday
b	Business day: A day specified using a business interval to define an acceptable business day. For instance, a day that is not a weekend or holiday.
mon	Monday
tue	Tuesday
wed	Wednesday
thu	Thursday
fri	Friday
sat	Saturday
sun	Sunday
jan	January
feb	February
mar	March
apr	April
may	May
jun	June
jul	July
aug	August
sep	September
oct	October
nov	November
dec	December

More Examples

Relative time expression Examples| +30d-8H | Move forward 30 days, then backup 8 hours. |

-12H+30s	Move backward 12 hours, then forward 30 seconds.
+23y-23M+23S-8m	Move forward 23 years, then backup 23 months, then move forward 23 milliseconds, then backup 8 minutes.
>sat	Move forward to Saturday.
>3sat>nov	Move forward 3 Saturdays, then skip ahead to November.

<5may>fri	Move backward 5 Mays, then advance to Friday.
@2005y+2y<sun	Go to the year 2005, then move forward 2 years, then backup to Sunday.
(2M)22d (8H)30m	Go to March 22nd, 8:30 AM of the current year.
^M>4mon	Go to the beginning of the month, then advance 4 Mondays.
+M^M>4mon	Go to the beginning of next month, then advance 4 Mondays.
^d+7H	Go to the beginning of today, then advance 7 hours.
>3D	Move forward three weekdays.
<4e	Move backward 4 weekend days.
>n	Move forward to the next non-holiday.
?D{+d}	If today is a weekday, advance a day.
?b{}{+2d}	If today is not a business day, advance two days.

Designing and Testing Time Expressions

Designing and testing time expressions can quickly become complicated as you incorporate more and more of the concepts that Flux time expressions offer. To assist you with this, the Web-based Designer in Flux includes a time expression editor, which can help guide you through both the creation and testing process for your time expressions.

The time expression editor is available on any trigger or action property that uses a time expression. To access the time expression editor:

1. Open the Web-based Designer.
2. Add a trigger or action that uses a time expression (such as the timer trigger) to the workspace.
3. Open the trigger or action's properties.
4. Find the property that accepts a time expression (for example, the time expression property on the timer trigger) and click the editor icon next to the property field.

This will display the time expression editor, which will guide you through the process of designing your time expression. When you're finished designing the expression, you can use the "test" button on the editor to make sure your expression covers the times and dates you are expecting.

If you have a running workflow that is waiting on a timer or delay trigger, you can edit the workflow and adjust the time expression to automatically update the schedule in-place, without affecting any existing or runtime data in the workflow.

Changing Time Settings on the System

If you change time settings on the machine where the engine is running (including changing the system clock or the locale setting of the JVM) you will need to restart the engine to pick up the new settings. Flux will continue to use the previous settings until the engine is restarted.

Leap Years

Flux will strictly honor schedules that include the leap day. These schedules will fire only when the actual scheduled date (leap day) occurs and will be ignored for any non-leap years.

Using Time Expressions within Process and Other Actions

Not all actions support time expressions (i.e., there is no time expression dialog available for the action), and runtime substitution does not support time expressions. So you need something to compute the business date – using your desired calendar - and using a time expression, like the following prescript that does the computation:

```
import flux.repository.*;

import fluximpl.repository.*;

RepositoryAdministrator repoAdmin = engine.getRepositoryAdministrator();

RepositoryIterator iterator = repoAdmin.get("/Weekdays-Excluding-Federal-Holidays");

    while(iterator.hasNext()){

        Date date = EngineHelper.applyTimeExpression("<3b", ((RepositoryElementImpl) iterator.next()).
getBusinessInterval(), null, 1);

        flowContext.put("businessDate", date);
```

```
    };
iterator.close();
```

“businessDate” is now in the flowContext, and you can reference it, and format it, like so :

```
${businessDate?string["MM-dd-yyyy"]}
```