Working With KPIs

Metrics and KPIs

Depending on who you ask the terms “metric” and “KPI” can mean the same thing. They are often used interchangeably to mean: a measurable value for some property at some specified time. However, within ServiceClarity these terms have two very distinct meanings:

ServiceClarity Metric:
Numeric values, plus optional breakdown metadata, collected direct from source in a format usually dictated by the capabilities of the source data or API. Metrics are collected regularly throughout the day with a resolution of up to every 10 minutes, with every 3 or 4 hours being typical. ***On any given day there can be multiple time stamped values collected for each metric****.*

ServiceClarity KPI:
Numeric values, plus optional breakdown metadata, calculated daily from one or more metrics or KPIs. A KPI can be a direct daily aggregate of a single metric or it can be calculated from a user supplied formula that combines several metrics or it can be a higher level KPI that builds on other existing KPIs. KPI values are daily but can be recalculated throughout the day as fresh metric data is collected. ***There is only ever one KPI value per day****.*

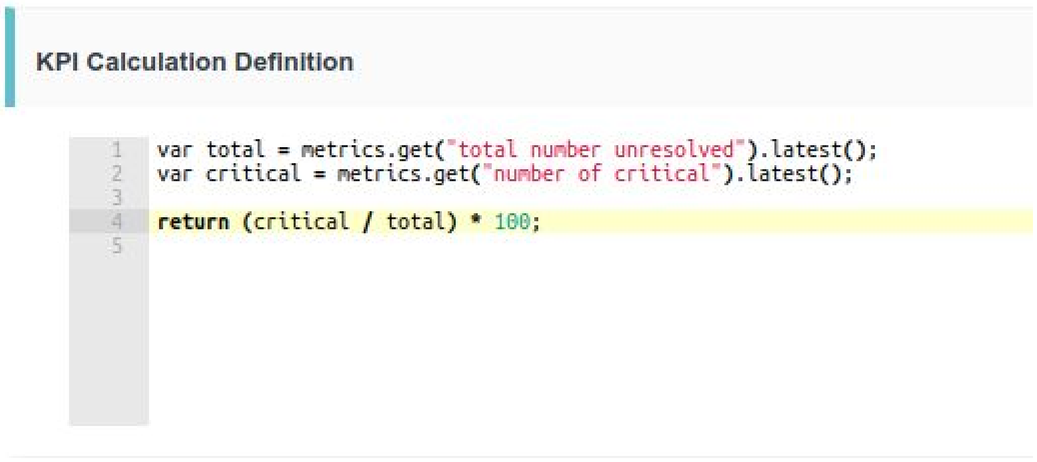
ServiceClarity KPIs are calculated by combining one or more metrics - or other KPIs - using standard javascript. The simplest KPI calculation is of course to take a single metric and return the last known value for the day and this is the default behaviour which you will see if you create a KPI without adding a formula. This document is intended as both a getting started guide that goes beyond this simple scenario and as a reference for the ServiceClarity KPI formula API.

A simple KPI example

To illustrate how KPI formula work imagine a scenario where you are collecting the number of unresolved critical issues from a Service Desk application but you are more interested in what percentage of all unresolved issues are critical. This is a simple example of how ServiceClarity can combine two metrics into a KPI using the formula:

```
% critical = 100 * (number of critical / total number unresolved)
```

Within ServiceClarity the javascript for the above formula might look like this:



KPI Calculation Definition

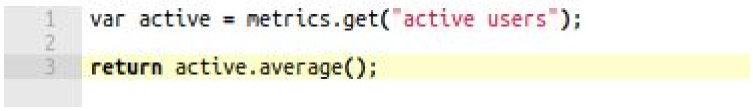
```
1 var total = metrics.get("total number unresolved").latest();
2 var critical = metrics.get("number of critical").latest();
3
4 return (critical / total) * 100;
5
```

- On line 1 the special ServiceClarity variable ***metrics*** is used to get the last known value for the metric: total number unresolved.
- On line 2 the same ***metrics*** variable is used to get the last known value for the metric: number of critical.
- On line 4 the two values for critical and and total issues are combined and returned as the calculated value for the KPI.

Aggregate KPIs

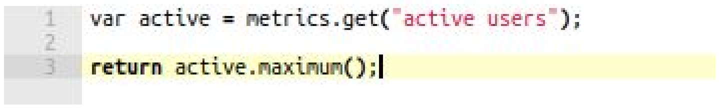
In the previous example we used the ***latest*** aggregation to get the metric data that was of interest; the latest value for a metric on a given day is typically the most useful value but there are other possibilities.

Imagine that we are running an online gaming service and we have a data source that can tell us the number of players actively using our service at this moment. We can create an *active users* metric to track this value every ten minutes. Two KPIs that might be if interest are the *average number of users* and the *maximum number of users*.



```
1 var active = metrics.get("active users");
2
3 return active.average();
```

- On line 1 the ServiceClarity variable ***metrics*** is used to get the metric: active users.
- On line 3 the aggregate function ***average*** is applied and the value of the KPI is returned as the average number of active users.



```
1 var active = metrics.get("active users");
2
3 return active.maximum();
```

- On line 1 the ServiceClarity variable ***metrics*** is used to get the metric: active users.
- On line 3 the aggregate function ***maximum*** is applied and the value of the KPI is returned as the average number of active users.

ServiceClarity provides a range of aggregation functions that can be applied to metrics and other KPIs:

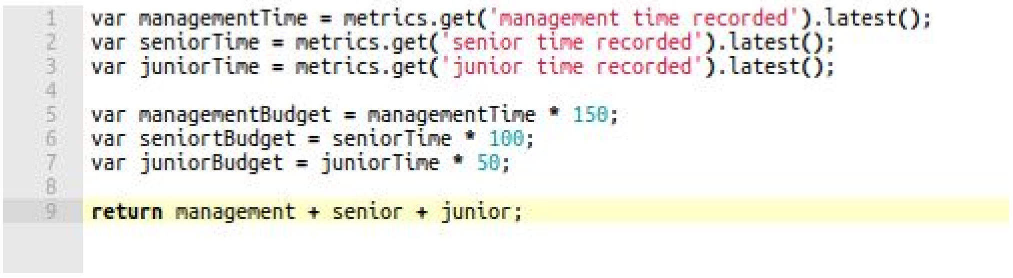
Aggregate	**Example**
Latest	`var latest = metrics.get('active users').latest()`
Average	`var average = metrics.get('active users').average()`
Maximum	`var max = metrics.get('active users').maximum()`
Minium	`var min = metrics.get('active users').minimum()`
Earliest	`var first = metrics.get('active users').earliest()`
Median	`var median = metrics.get('active users').median()`
Sum	`var total = metrics.get('active users').sum()`

Building KPIs from KPIs

A KPI can be built from one or more metrics and from one or more KPIs. Calculating a KPI from another KPI offers an opportunity to reuse existing calculations and to access the history of a KPI value, which opens up interesting possibilities for broader aggregations, statistical functions and even prediction.

For example, perhaps you are tracking a KPI for the *daily budget spent*. This KPI has been calculated from a time reporting metric or inferred from a workflow management tool such as JIRA. We can reuse this KPI for *daily budget spent* to calculate a new KPI for the *total budget spent* and we can reuse this new KPI to build yet another KPI that predicts the total budget spent at the end of the month.

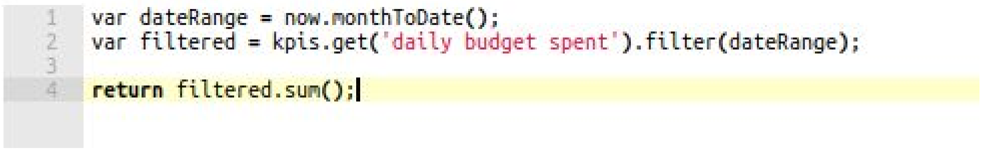
Daily budget spent:



```
1  var managementTime = metrics.get('management time recorded').latest();
2  var seniorTime = metrics.get('senior time recorded').latest();
3  var juniorTime = metrics.get('junior time recorded').latest();
4
5  var managementBudget = managementTime * 150;
6  var seniortBudget = seniorTime * 100;
7  var juniorBudget = juniorTime * 50;
8
9  return management + senior + junior;
```

- On lines 1-3 get the latest values for three different time metrics.
- On lines 5-7 convert time to money using different rates for each metric value
- On line 9 return the total budget spent by summing all three components.

Total budget spent:

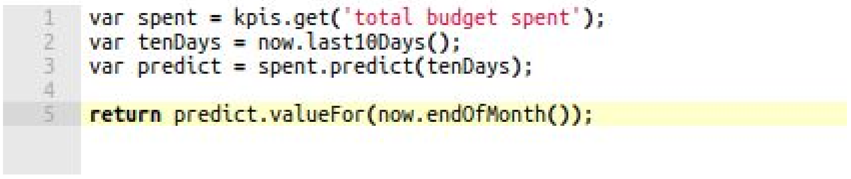


```
1  var dateRange = now.monthToDate();
2  var filtered = kpis.get('daily budget spent').filter(dateRange);
3
4  return filtered.sum();
```

- On line 1 the special ServiceClarity variable ***now*** is used to define a range of dates, from the start of the month until the current date.

- On line 2 the ServiceClarity variable ***kpis*** is used to get the existing KPI: daily budget spent and then ***filter*** the to data to include only dates in our date range.
- On line 3 the aggregate function ***sum*** is applied and the value of the KPI is returned as the total budget spent in the month to date

Predicted end of month spend:



```
1  var spent = kpis.get('total budget spent');
2  var tenDays = now.last10Days();
3  var predict = spent.predict(tenDays);
4
5  return predict.valueFor(now.endOfMonth());
```

- On line 1 get the kpi for total budget spent calculated in the previous steps.
- On line 2 use the ServiceClarity variable ***now*** to define a date range that covers the last 10 days.
- On line 3 use the last 10 days and the KPI budget values to predict how the budget will evolve over the coming days
- On line 5 use the prediction to calculate the expected budget on the last day of the month and return this as the value for our new *predicted end of month spend* KPI

Defensive coding

KPIs are data driven, often calculated from raw metrics collected from data sources that do not always contain data for a specific date or time or may occasionally not respond. This unpredictability of real world data needs to be taken into account when writing KPI formula. As with all forms of programming there is value in a certain amount of *defensive coding*.

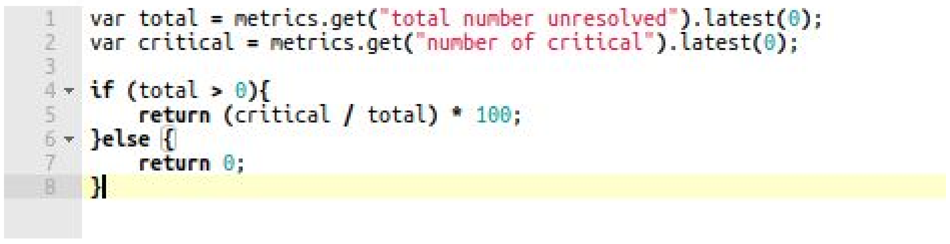
Take our first example KPI calculation for the *percentage critical issue*. This is a KPI built from two metrics, a metric that collects the number of critical issues opened in a day and one that collects the total number of issues raised on the day. On a typical day the hope would be that there are zero critical issue to it likely that the value for the critical issues metric will regularly be zero. If we review our formula we can see that this doesn't present much of an issue:

```
% critical = 100 * (number of critical / total number unresolved)
```

A zero value for *number of critical* will return zero which is the correct answer.

On a really good day we can hope that the *total number unresolved* is also zero. In this case our original formula fails because with a zero value for *total number unresolved* we have a divide by zero error. To cope with this possibility we need to add some defensive coding.

While we consider possible error scenarios we should also protect against issue with collecting data from the underly data source. Not all sources of data are available 100% of the time and although the likelihood of failure is sometimes vanishingly small ServiceClarity provides simple ways to protect against this when writing KPI formula.



```
1  var total = metrics.get("total number unresolved").latest(0);
2  var critical = metrics.get("number of critical").latest(0);
3
4  if (total > 0){
5      return (critical / total) * 100;
6  }else {
7      return 0;
8  }
```

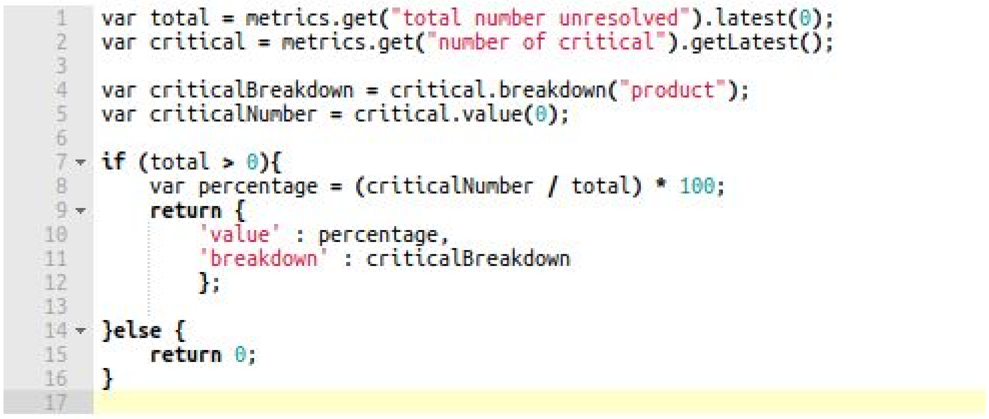
- On line 1 as before we use the variable ***metrics*** to get the last known value for the metric: total number unresolved only this time we set a default value of 0 so that we can be sure that even unexpected failures don't interfere with our calculation.
- On line 2 the same ***metrics*** variable is used to get the last known value for the metric: number of critical again with a default value of 0
- On line 4 we protect against the possibility that the total is zero before we calculate the percentage KPI value, on line 5.
- On line 7 we provide the correct alternative KPI value of zero critical issues

Working with Breakdowns

Although in essence a metric is a numeric value of a quantity at a given time ServiceClarity provides the capability to calculate metadata about that value at that point in time. This is called the breakdown and more than one breakdown can be collected about a metric, depending on the capabilities of the underlying data source.

If we return to our first example the *% critical issues* we can imagine that it would be useful to know how that KPI breaks down by product line, service or component. Collecting this kind of metadata from a metric source such as JIRA is relatively simple. Lets rework the *% critical issues* formula to also return the breakdown by ***product*** - assuming our metric has been configure to collect this metadata.

This requires a slightly different approach to the KPI formula as this formula needs to work with the value for the metric *and* the metadata about that value.

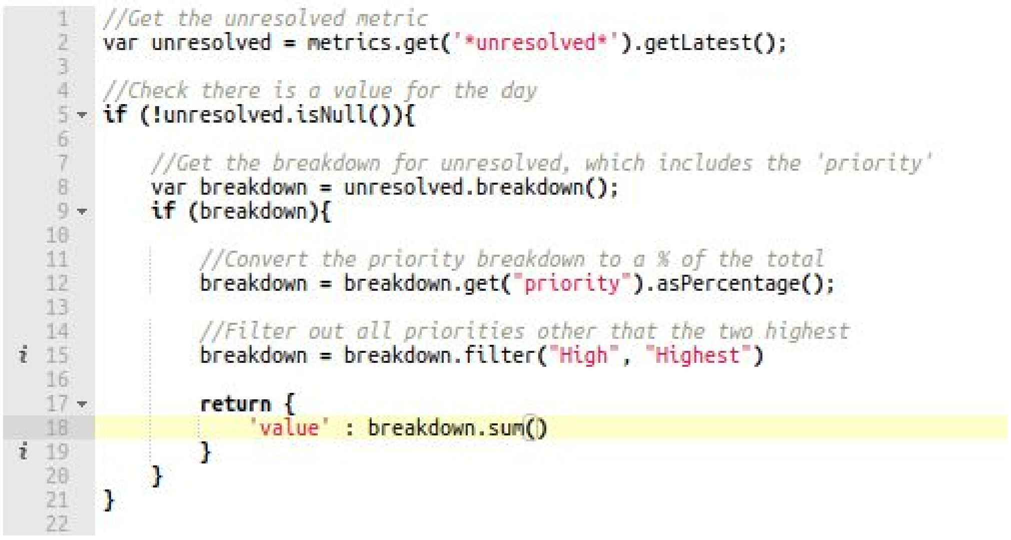


```
 1  var total = metrics.get("total number unresolved").latest(0);
 2  var critical = metrics.get("number of critical").getLatest();
 3
 4  var criticalBreakdown = critical.breakdown("product");
 5  var criticalNumber = critical.value(0);
 6
 7  if (total > 0){
 8      var percentage = (criticalNumber / total) * 100;
 9      return {
10          'value' : percentage,
11          'breakdown' : criticalBreakdown
12          };
13
14  }else {
15      return 0;
16  }
17
```

- On line 1 as before we use the variable ***metrics*** to get the last known value for the metric: total number unresolved and as before we set a default value of 0
- On line 2 the same ***metrics*** variable is used to get the the metric: number of critical only this time we have used the ***getLatest()*** function to retrieved the last known object rather than the numeric value.
- On line 4 because the object also contains the metadata for the metric as well as the value we can extract the breakdown metadata for ***product***
- On line 5 we also still need the numeric value for the critical number of issues, again we set a default value of 0
- On line 8 we calculate the percentage KPI value using the two numeric values.
- On lines 9-12 we return the numeric value for our metric *and* the breakdown metadata for product
- On line 15 we provide the correct alternative KPI value of zero critical issues

Convert a Breakdown into a KPI

Sometimes the most interesting KPIs that you can created come from the metadata rather than the numeric metric or KPI value. This is especially true when it is important to compare the proportion of one breakdown value to another. For example, if we collect a metric for the *number of unresolved issues* why might choose to a breakdown for this metric of ***Priority***. A useful KPI that can be calculated from this metric is the *% of unresolved high priority issues*.



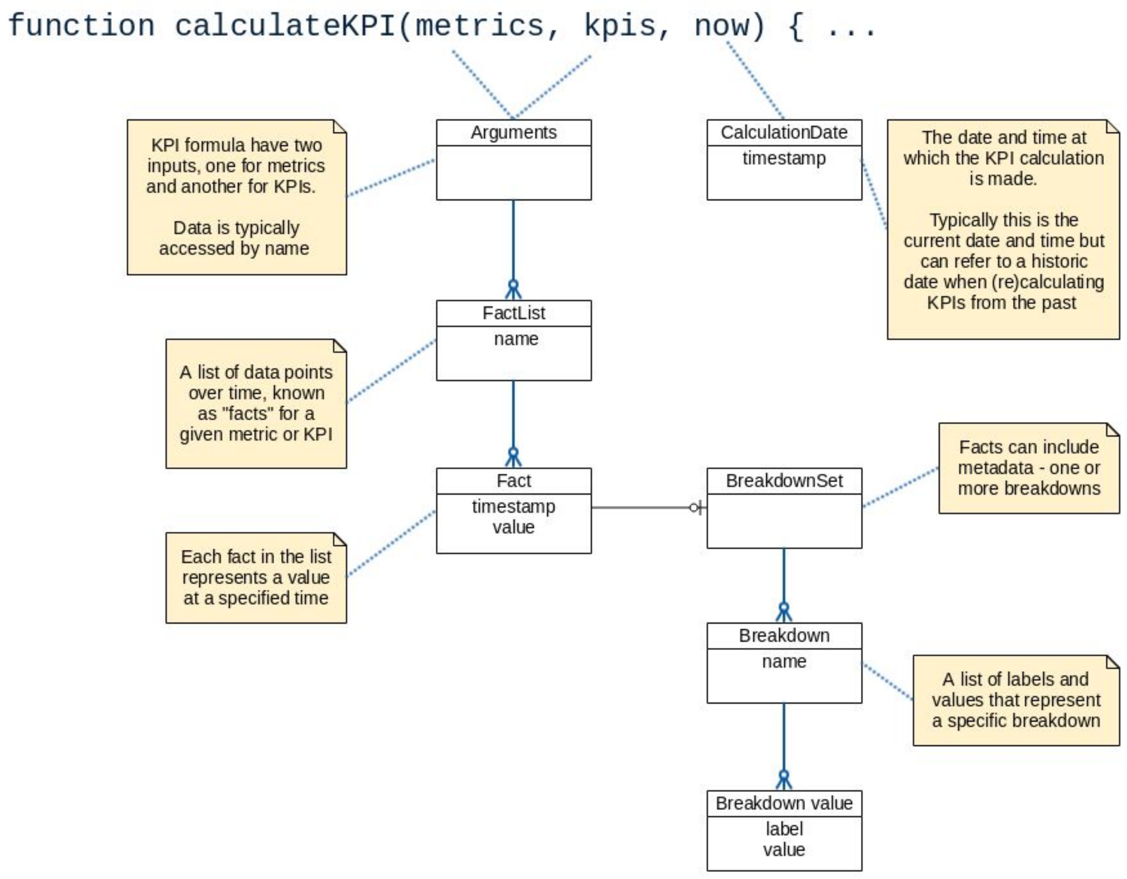
```
 1  //Get the unresolved metric
 2  var unresolved = metrics.get('*unresolved*').getLatest();
 3
 4  //Check there is a value for the day
 5  if (!unresolved.isNull()){
 6
 7      //Get the breakdown for unresolved, which includes the 'priority'
 8      var breakdown = unresolved.breakdown();
 9      if (breakdown){
10
11          //Convert the priority breakdown to a % of the total
12          breakdown = breakdown.get("priority").asPercentage();
13
14          //Filter out all priorities other that the two highest
15          breakdown = breakdown.filter("High", "Highest")
16
17          return {
18              'value' : breakdown.sum()
19          }
20      }
21  }
22
```

- On line 2 as before we use the variable ***metrics*** to get the last known value for the metric using the wild card expression: *unresolved*
- On line 5 we add some defensive coding to check that values are available for the date of the calculation
- On line 8 get the breakdown metadata for the unresolved issues
- On line 9 we have more defensive coding
- On line 12 we get the priority breakdown and convert it to % values
- On line 15 remove all low priority values from the priority breakdown
- On line 18 return the sum of all remain priority breakdown values as the value for the KPI

The anatomy of a KPI formula

The previous examples refer to special ***metrics, kpis*** and ***now*** variables and make use of some aggregation, prediction and breakdown functionality that hints at greater capabilities. To discuss these we need to look at how ServiceClarity interprets your KPI formula and at the underlying scripting object model that ServiceClarity provides. Your KPI formula are treated as sandboxed javascript functions with three input variables:

1. ***metrics*** - an Argument object that can contain zero, one or more lists of metric data
2. ***kpis*** - an Argument object that can contain zero, one or more lists of KPI data
3. ***now*** - a Calculation Date object that represents the date and time when the KPI is calculated for, nominally this is 'now' but is could be date in the past.


function calculateKPI(metrics, kpis, now) { ...
Arguments
KPI formula have two inputs, one for metrics and another for KPIs.
Data is typically accessed by name
CalculationDate
timestamp
The date and time at which the KPI calculation is made.
Typically this is the current date and time but can refer to a historic date when (re)calculating KPIs from the past
FactList
name
A list of data points over time, known as "facts" for a given metric or KPI
Fact
timestamp
value
BreakdownSet
Facts can include metadata - one or more breakdowns
Each fact in the list represents a value at a specified time
Breakdown
name
A list of labels and values that represent a specific breakdown
Breakdown value
label
value

Arguments:

Which metrics and KPIs are available to the calculation depends on how the KPI parameters are configured. It is common to have no KPIs or no metrics, a single metric or KPI or a collection of both. For this reason the Arguments object provides functions to pick out individual metrics/KPIs or filter sets of them. It also provides functions for finding the latest or first time values and additional checks for missing values. A detailed description of the functionality provided by the Arguments object can be found in the Reference section of this document.

Calculation Date:

KPI calculations are made for a specific date and time. Normally this date and time will be time at which the calculation is run but if a KPI is (re)calculated for past dates the date value of the ***now*** variable will reflect the historic date of the calculation and the time will always be set to 23:59:59 - the last moment of the day. This is because only one value can be stored for a KPI for a given date and the last time stamp always overrides a previous one.
As illustrated in the examples from the previous section the ***now*** variable has a number of useful functions for defining date ranges that can be used in filters and prediction calculations. A detailed description of the functionality provided by the Calculation Date object can be found in the Reference section of this document.

Outputs:

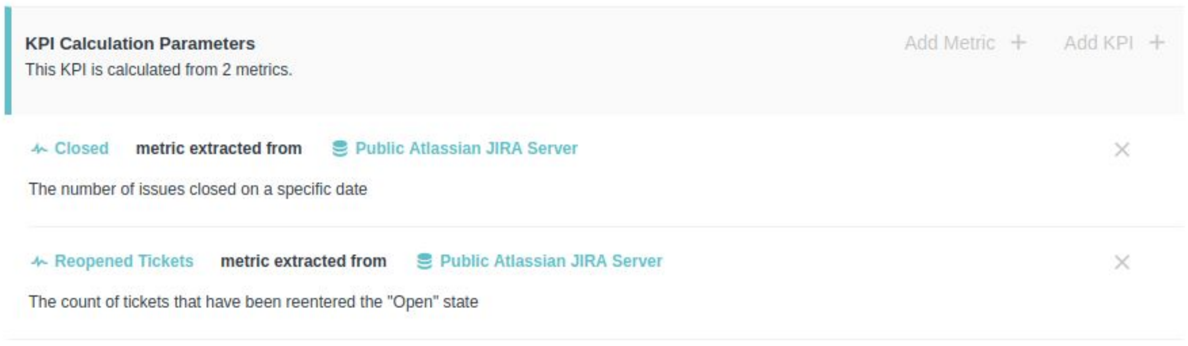
The output of your KPI javascript functions can be:

1. ***undefined*** - There is no requirement to return anything as not all KPIs have a value for all dates.
2. ***numeric value*** - the value for the KPI on the date defined by ***now***
3. **object** - a simple javascript object that contains the properties:
    a. ***value*** for the KPI on the date, the
    b. ***breakdown*** for the KPI and optionally a numeric
    c. ***sample*** size for cases where aggregations can be affected by the significance of the input data.

Reference

Arguments object

The Arguments object is a set of named FactList objects that contains data for metrics or for KPIs. All KPI calculations are supplied with two Arguments: one for metrics and another for KPIs. These two Arguments will contain the data for the metrics and KPIs configured in the KPIs parameters:



KPI Calculation Parameters
This KPI is calculated from 2 metrics.

Closed **metric extracted from** Public Atlassian JIRA Server
The number of issues closed on a specific date

Reopened Tickets **metric extracted from** Public Atlassian JIRA Server
The count of tickets that have been reentered the "Open" state

The above example configuration for a KPI will populate the metrics Argument with two FactLists, one for Closed Issues and the other for Reopened Tickets and the kpis FactList will be empty

Functions

breakdown() returns a BreakdownSet

Return the breakdown metadata for all metrics or KPIs in the argument, merging all metadata into a single Breakdown Set.

This is a convenience method most applicable when a single metric or a single kpi is used in the calculation. Because the input metric and KPI data is normally for a collection of dates and times calling this function without first filtering the argument for the desired date will result in all dates being merged together.

```
var breakdown = metrics.latest().breakdown();
```

In the above example the metrics are first filtered to only contain the last known data point and then the breakdown is merged and returned. If a single metric is input to the formula the above would be equivalent to calling:

```
var breakdown = metrics.getLatest('issues').breakdown();
```

breakdown(name, ...) returns a BreakdownSet

Return one or more named breakdown metadata for all metrics or KPIs in the argument, merging all metadata into a single BreakdownSet.

This is a convenience method most applicable when a single metric or a single kpi is used in the calculation. Because the input metric and KPI data is normally for a collection of dates and times calling this function without first filtering the argument for the desired date will result in all dates being merged together.

```
var breakdown = metrics.latest().breakdown('product');
```

In the above example the metrics are first filtered to only contain the last known data point and then the **product** breakdown is merged and returned. If a single metric is input to the formula the above would be equivalent to calling:

```
var breakdown = metrics.getLatest('...').breakdown('product');
```

count() returns an integer

For each metric or KPI in the Argument count and sum the number of data points. This is a convenience method most applicable when a single metric or a single KPI is used in the calculation.

```
var count = metrics.count();
```

If a single metric is input to the formula the above would be equivalent to calling:

```
        var count = metrics.get('...').count();
```

earliest() returns an Argument

By default all metrics data collected for the day is provided to the KPI calculations and for KPI inputs the last 30 days of data is provided. This function filters all metrics or KPIs in this Argument by date and time and returns a new Argument that only includes the first chronologically data points. For example, if the KPI calculation can discard all but the first value for each metric then the metrics variable can be filtered and re-assigned as follows:

```
        metrics = metrics.earliest();
        var all = metrics.get('Unresolved Issues');    //first data for unresolved
        var critical = metrics.get('Critical Issues'); //first data for critical
```

filter(name, ...) returns an Argument

Filter all metrics or KPIs in this Argument by name and return a new Argument that only contains the entries that match the input names. Matching is case insensitive and can include wild cards.

```
        var subsetOfMetrics = metrics.filter('*issues');
```

The above example will search the metrics in the argument for all those that match the wildcard expression ***Issues**. If none match this expression an empty Argument object will be returned.

Filter functions can also accept javascript regular expressions as filters. The previous example could be rewritten as, note the need to account for case when using javascript RegEx filter.

```
        var subsetOfMetrics = metrics.filter(/.+[Ii]ssues/);
```

get() returns a FactList

This is a convenience function for returning the first entry in the Argument. It is most applicable when a single metric or a single kpi is used in the calculation and can simplify the complexity of the formula. For example the following are equivalent expressions when a single metric is used in the KPI calculation:

```
var issues = metrics.get();
var issues = metrics.get('Critical Issues'); //find a metric by name
var issues = metrics.get(0);                 //the first metric by index
```

Note that the returned list of facts may include multiple data points depending on previous date/time filters and the input data set. By default metrics data for the day is provided to the KPI calculations and for the last 30 days for KPI inputs. Both of these defaults can be changed on a per KPI basis.

get(index) returns a FactList

This is a convenience function for returning the entry in the Argument by index. It is most applicable when a single metric or a single KPI is used in the calculation and can simplify the complexity of the formula. For example the following are equivalent expressions when a single metric is used in the KPI calculation:

```
var issues = metrics.get();
var issues = metrics.get('Critical Issues'); //find a metric by name
var issues = metrics.get(0);                 //the first metric by index
```

Note that the returned list of facts may include multiple data points depending on previous date/time filters and the input data set. By default metrics data for the day is provided to the KPI calculations and for the last 30 days for KPI inputs. Both of these defaults can be changed on a per KPI basis.

get(name) returns a FactList

Find the data for a specific metric or KPI in the Argument by name or by wildcard. Metric and KPI name searches are case insensitive.

If a wildcard is used and more than one entry matches the expression the first found will be returned - this is not usually what was intended so care should be taken when using wildcards to ensure that they match a single entry.

```
var issues = metrics.get('Critical Issues'); //find a metric by name
var issues = metrics.get('Critical*');       //find a metric by wildcard
```

Note that the returned list of facts may include multiple data points depending on previous date/time filters and the input data set. By default metrics data for the day is provided to the KPI calculations and for the last 30 days for KPI inputs. Both of these defaults can be changed on a per KPI basis.

isEmpty() returns a boolean

Returns **true** if there are no metrics or KPIs in the argument, otherwise **false**. This can be a useful debug function rather than function for calculating KPI values.

isNull() returns a boolean

Returns **true** if there are no metrics or KPIs in the argument, or if all of the entries in the Argument are themselves empty of data. This can be a useful debug function rather than function for calculating KPI values. It can also be valuable for defensive coding function if the incoming metrics are regularly missing due to the nature of the data or data source.

latest() returns an Argument

Filter all metrics or KPIs in this Argument by date and time and return a new Argument that only includes the most recent data points. For example, if the KPI calculation can discard all but the latest value for each metric then the metrics variable can be filtered and re-assigned as follows:

```
metrics = metrics.latest();
var all = metrics.get('Unresolved Issues');    //latest data for unresolved
var critical = metrics.get('Critical Issues'); //latest data for critical
```

By default all metrics data collected for the day is provided to the KPI calculations and for KPI inputs the last 30 days of data is provided. For this reason the above reassignment is a very common pattern as many KPIs are built from the latest metrics. There are other ways of achieving the same date filtering without discarding all other data but this can be the cleanest and can reduce the apparent complexity of KPI formula.

maximum() returns an Argument

By default all metrics data collected for the day is provided to the KPI calculations and for KPI inputs the last 30 days of data is provided. This function filters all metrics or KPIs in this Argument by ***value*** and returns a new Argument that only includes the data points for the highest value for each metric or KPI in the original Argument. For example, if the KPI calculation can discard all but the maximum value for each metric then the metrics variable can be filtered and re-assigned as follows:

```
metrics = metrics.maximum();
var all = metrics.get('Unresolved Issues');    //max data for unresolved
var critical = metrics.get('Critical Issues'); //max data for critical
```

minimum() returns an Argument

By default all metrics data collected for the day is provided to the KPI calculations and for KPI inputs the last 30 days of data is provided. This function filters all metrics or KPIs in this Argument by ***value*** and returns a new Argument that only includes the data points for the lowest value for each metric or KPI in the original Argument. For example, if the KPI calculation can discard all but the minimum value for each metric then the metrics variable can be filtered and re-assigned as follows:

```
metrics = metrics.maximum();
var all = metrics.get('Unresolved Issues');    //min data for unresolved
var critical = metrics.get('Critical Issues'); //min data for critical
```

names() an Array of Strings

Return the names of all metrics or KPIs in the Argument. This is a useful debug function and can be used for logging.

```
var names = metrics.names();
console.log('Loaded input metrics: %s', names);
```

size() return an integer

Return the number of metrics or KPIs in the Argument. This is a useful debug function and can be used for logging.

```
console.log('Loaded %s input metrics', metrics.size());
```

value() returns a decimal

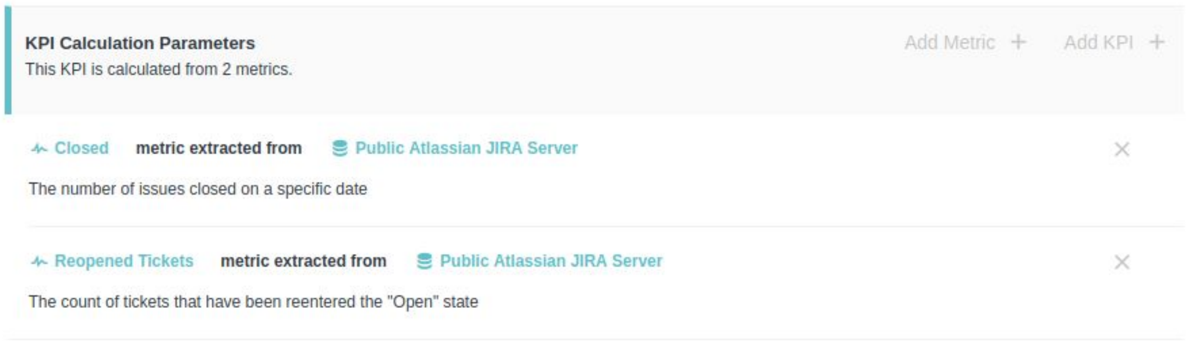
This is a convenience function that reduces the content of the Argument to a numeric value by summing all the values for all metrics or KPIs in the Argument. It is most applicable when a single KPI or metric is input to the KPI formula as it provides a shorthand way of processing the single input. For example

```
metrics = metrics.latest();      //reduce the input to the latest value
return {
 value :  metrics.value(),
 breakdown :  metrics.breakdown()
};
```

If the above formula is applied to a single metric input the output would be the same as the default behaviour for all KPIs that do not have a custom KPI formula.

FactList object

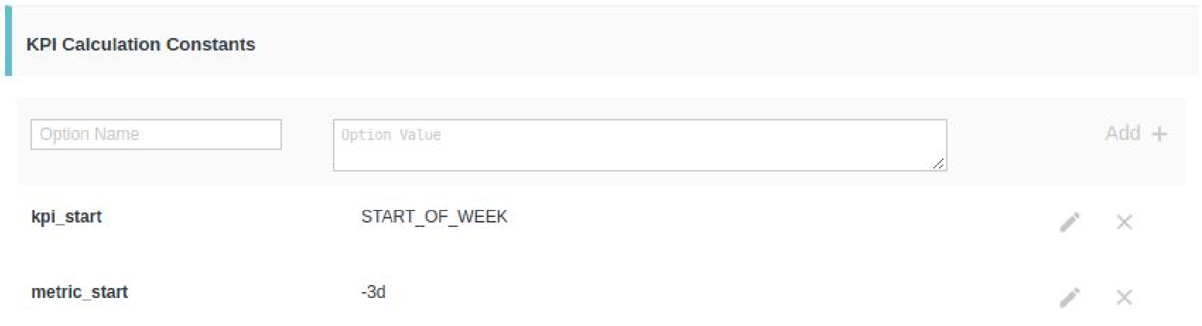
A FactList object is a chronological list of Fact objects that contain the values and breakdowns for a single metric or KPI. The FactList objects passed to a KPI calculation are configured in the KPIs parameters:


KPI Calculation Parameters
This KPI is calculated from 2 metrics.

Closed **metric extracted from** Public Atlassian JIRA Server
The number of issues closed on a specific date

Reopened Tickets **metric extracted from** Public Atlassian JIRA Server
The count of tickets that have been reentered the "Open" state

By default FactLists for metrics contain data for the date of the KPI calculation and FactLists for KPIs contain data for the previous 30 days. Both of these defaults can be changed by overriding the configuration options ***kpi_start*** and ***metric_start*** in the KPIs constants.


KPI Calculation Constants

kpi_start	START_OF_WEEK
metric_start	-3d

The above example sets the KPI FactList date range from the start of the week until the date of the KPI calculation and the metric FactList date range is for 3 days up until the date of the KPI calculation.

The default values for configuration options for ***kpi_start*** and ***metric_start*** are **`-30d`** and **`START_OF_DAY`** respectively and the default end date of the range is always the date and time of the KPI calculation. The end date of of ranges can also be overridden using the configuration options: ***kpi_end*** and ***metric_end***.

The following is a list of all the support date keywords that can be used to define the start and end date ranges for metric and KPI fact lists.

```
END_OF_DAY
END_OF_LAST_MONTH
END_OF_LAST_WEEK
END_OF_LAST_YEAR
END_OF_MONTH
END_OF_NEXT_MONTH
END_OF_NEXT_WEEK
END_OF_NEXT_YEAR
END_OF_WEEK
END_OF_YEAR
LAST_MONTH
LAST_WEEK
LAST_YEAR
NEXT_MONTH
NEXT_WEEK
NEXT_YEAR
NOW
START_OF_DAY
START_OF_LAST_MONTH
START_OF_LAST_WEEK
START_OF_LAST_YEAR
START_OF_MONTH
START_OF_NEXT_MONTH
START_OF_NEXT_WEEK
START_OF_NEXT_YEAR
START_OF_WEEK
START_OF_YEAR
TODAY
TOMORROW
YESTERDAY
```

All of the above date keywords can be extended with offsets - days, weeks months or years - to provide even greater control over the metric and KPI FactLists. For example,

```
NOW(-1d)              //equivalent to YESTERDAY
START_OF_MONTH(-1d)   //end of last month
START_OF_MONTH(-6M)   //6 months ago at the start of the month
START_OF_WEEK(-4w)    //4 weeks ago at the start of the week
END_OF_WEEK(-1w)      //end of last week
END_OF_MONTH(-1M)     //end of last month
END_OF_MONTH(-3M)     //go back 3 months at end of that month
START_OF_YEAR(-1y)    //start of last year
```

Functions

average(number) returns a decimal

Return the average numeric value of all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var avg = list.average(); // if list is empty NaN is returned
if (!isNaN(avg)){
    return avg;
}
```

breakdown() returns a BreakdownSet

Return the combined breakdown metadata for all metrics or KPIs Facts in the FactList, merging all metadata into a single BreakdownSet. This is largely a convenience method that is most applicable when the FactList has been filtered to a single data point.

```
var breakdown = kpis.latest().get('issues').breakdown();
```

The above example is equivalent to:

```
var breakdown = kpis.get('issues').filter(now).breakdown();

var sameBreakdown = kpis.getLatest('issues').breakdown();
```

breakdown(name, ...) returns a BreakdownSet

Return a filtered set of breakdown metadata for all metrics or KPIs Facts in the FactList, merging all metadata into a single BreakdownSet containing only the breakdowns named in the parameters. This is largely a convenience method that is most applicable when the FactList has been filtered to a single data point.

```
var breakdownByType = kpis.latest().get('issues').breakdown('Type');
```

The above example is equivalent to:

```
var breakdown = kpis.get('issues').filter(now).breakdown('Type');

var sameBreakdown = kpis.getLatest('issues').breakdown('Type');
```

count() return integer

Return the number of Facts in a metric or KPI FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return 0.

```
var list = kpis.get('daily expenses'); // returns a FactList
var count = list.count(); // if list is empty zero is returned

var sum = list.sum(0);

return sum / count;
```

countGreaterOrEqualTo(number) returns an integer

Return the number of Facts in a metric or KPI FactList that have values greater than or equal to the number supplied, ignoring null or NaN values. If no facts are in the list or all the facts are null, NaN or less than the requested threshold return 0.

```
var list = kpis.get('daily expenses'); // returns a FactList
var count = list.countGreaterThanOrEqualTo(50);
console.log('Found %s expenses above proof of receipt threshold', count);
```

countGreaterThan(number) returns an integer

Return the number of Facts in a metric or KPI FactList that have values greater than the number supplied, ignoring null or NaN values. If no facts are in the list or all the facts are null, NaN or less than the requested threshold return 0.

```
var list = kpis.get('daily expenses'); // returns a FactList
var count = list.countGreaterThan(50);
console.log('Found %s expenses above proof of receipt threshold', count);
```

countLessOrEqualTo(number) returns an integer

Return the number of Facts in a metric or KPI FactList that have values less than or equal to the number supplied, ignoring null or NaN values. If no facts are in the list or all the facts are null, NaN or less than the requested threshold return 0.

```
var list = kpis.get('utilisation'); // returns a FactList
var count = list.countLessThanOrEqualTo(50);
console.log('Found %s under utilised entries', count);
```

countLessThan(number) returns an integer

Return the number of Facts in a metric or KPI FactList that have values less than the number supplied, ignoring null or NaN values. If no facts are in the list or all the facts are null, NaN or less than the requested threshold return 0.

```
var list = kpis.get('utilisation'); // returns a FactList
var count = list.countLessThan(50);
console.log('Found %s under utilised entries', count);
```

countNaN() returns an integer

Return the number of Facts in the metric or KPI FactList that do not have a value. This can be used to check for missing data that is expected.

```
var ping = metrics.get('ping response');
var total = ping.size();
var failed = ping.countNaN();
var slow = ping.countGreaterThan(500);

var availability = (total - (failed+slow)) * 100 / total;
return {
    value : availability,
    sample : total
}
```

earliest(number) returns a decimal

Returns the numeric value of the first metric or KPI Fact in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var min = list.earliest(); // if list is empty NaN is returned
if (!isNaN(min)){
    return min;
}
```

filter(DateRange) returns a FactList

Returns a new FactList filtered that includes only those facts from the original that are within the specified DateRange. All DateRanges are built from the ***now*** parameter which provides a number of methods for expressing complex ranges relative to the date of the KPI calculation.

```
//filter out data not from the calendar month of the calculation date
var filtered = kpis.get('issue count').filter(now.monthToDate());

//filter out data not from the week (mon-sun) of the calculation date
var thisWeek = now.startOfWeek().until(now.endOfWeek());
Var filtered = kpis.get('issue count').filter(thisWeek);


//filter out data not from first half of the day of the calculation date
var firstHalf = now.withTime(0,0,0,).until(now.withTime(12,0,0));
Var filtered = metrics.get('issue count').filter(firstHalf);
```

filter(Date) returns a FactList

Returns a new FactList filtered that includes only those facts from the original that are on the specified Date. All Dates are defined as relative from the ***now*** parameter which provides a number of methods for expressing dates relative to the date of the KPI calculation.

```
//filter out data not from day before the KPI calculation
var filtered = kpis.get('issue count').filter(now.yesterday());

//filter out data not from the date of the KPI calculation
var filtered = kpis.get('issue count').filter(now);

//filter out data not from three days prior to the calculation date
var filtered = now.minus(3, 'days');
```

getMaximum() returns a Fact

Returns the Fact with the maximum value in a metric or KPI FactList. If there are no Facts in the list or all of the Facts contain null or NaN values return a MissingFact which has no data and not breakdown. The MissingFact is a special case of a Fact object that can be useful to simplify defensive coding checks.

```
// get the largest expense in the last 30 days
```

```
var expenses = kpis.get('expenses').getMaximum();

// if there are no facts in the 'expenses' list this will 0
return expenses.value(0);
```

getMinimum() returns a Fact

Returns the Fact with the minimum value in a metric or KPI FactList. If there are no Facts in the list or all of the Facts contain null or NaN values return a MissingFact which has no data and not breakdown. The MissingFact is a special case of a Fact object that can be useful to simplify defensive coding checks.

```
// get the quickest response time
var quickest = metrics.get('response time').getMinimum();

// return the value and the breakdown
return quickest;
```

getLatest() returns a Fact

Returns the last Fact, chronologically, from a metric or KPI FactList. If there are no Facts in the list or all of the Facts contain null or NaN values return a MissingFact which has no data and not breakdown. The MissingFact is a special case of a Fact object that can be useful to simplify defensive coding checks.

```
// get the latest
var latest = metrics.get('critical issues').getLatest();

// return the value and the breakdown
return latest;
```

getEarliest() returns a Fact

Returns the first Fact, chronologically, from a metric or KPI FactList. If there are no Facts in the list or all of the Facts contain null or NaN values return a MissingFact which has no data and not breakdown. The MissingFact is a special case of a Fact object that can be useful to simplify defensive coding checks.

```
// get the earliest
var latest = metrics.get('critical issues').getEarliest();

// return the value and the breakdown
return latest;
```

isEmpty() returns a boolean

Returns ***true*** if a metric or KPI FactList contains no entries. If there are any Facts in the list even if they contain null or NaN values this function will return ***false***. This is a useful function when developing new KPI formula as it can be used for simple debug logging.

```
if (metrics.get('critical issues').isEmpty()){
    console.log('No critical issues for date: %s', now);
}
```

isNull() returns a boolean

Returns ***true*** if a metric or KPI FactList contains no entries that have real values. If there are Facts in the list but they contain null or NaN values this function will return ***true***. This is a useful function when developing new KPI formula as it can be used for simple debug logging.

```
if (metrics.get('critical issues').isNull()){
    console.log('No critical issues for date: %s', now);
}
```

latest(number) returns a decimal

Returns the numeric value of the latest, chronological, metric or KPI Fact in the FactList. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('recorded time'); // returns a FactList
var latest = list.latest(0); // if list is empty 0 is returned
```

```
        return latest;
```

maximum(number) returns a decimal

Returns the largest numeric value of all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var max = list.maximum(); // if list is empty NaN is returned
if (!isNaN(max)){
    return max;
}
```

median(number) returns a decimal

Returns the median numeric value of all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var median = list.median(); // if list is empty NaN is returned
if (!isNaN(median)){
    return median;
}
```

minimum(number) returns a decimal

Returns the smallest numeric value of all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var min = list.minium(); // if list is empty NaN is returned
if (!isNaN(min)){
    return min;
}
```

name() returns a String

Returns the name of the metric or KPI for a FactList. This is a useful function when developing new KPI formula as it can be used for debug logging

```
var list = kpis.get('AWS EC2 Costs');
console.log('KPI %s contains %s entries', list.name(), list.count());
```

predict(DateRange) returns a LinearFit

Returns a LinearFit object calculated from the FactList data points. If a DateRange is provided as a parameter this range is used to filter out data point for the prediction. The predict function uses a linear regression analysis of the metric or KPI facts to enable prediction of future values. A typical use would be to analyse the recent trend, over 7-10 days and use that to provide a prediction of the end of month value for the KPI.

```
var spent = kpis.get('total budget spent'); // The list of values
var tenDays = now.last10Days();             // a DateRange for 10 days
var predict = spent.predict(tenDays);       // analyse the last 10 days

// output the predicted end of of month value
return predict.valueFor(now.endOfMonth());
```

When making predictions for an end of month date it is often necessary to be careful with how you treat early predictions. For example, as there is insufficient data in the first day or two of each month the predicted value is unlikely to be helpful. Similarly the final days of the month should gradually trend towards the actual value on the final day of the month. A more sophisticated prediction KPI formula should take these special cases into account.

```
//Filter the data to the month of the KPI calculation date
var spent = kpis.get('budget spent').filter(now.monthToDate());

//At the end of the month the real value is always correct
if (now.isEndOfMonth()){
    return spent.getLatest();
}
```

```
//later in the month use the last weeks worth of data points to predict
if (now.day() > 15){
    var predict = spent.predict(now.last7days());
    return  predict.valueFor(now.endOfMonth());
}
//early in the month use all the available data
if (now.day() > 2){
    var predict = spent.predict();
    return  predict.valueFor(now.endOfMonth());
}
//In the first two days there is no value to the prediction
//As a placeholder why not return last months value?
return kpis.get('budget spent').valueFor(now.startOfMonth().minus(1, 'd'));
```

sum(number) returns a decimal

Return the sum of all values for all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = kpis.get('daily expenses'); // returns a FactList
var sum = list.sum(0); // if list is empty zero is returned
return sum;
```

the95thPercentile(number) returns a decimal

Return the 95th percentile of all metric or KPI Facts in the FactList, ignoring null or NaN values. If no facts are in the list or all the facts have null or NaN values then return the default value passed to the function or NaN if a default value is not supplied.

```
var list = metrics.get('support response time'); // returns a FactList
var outliers = list.the95thPercentile(); // if list is empty NaN is returned
if (!isNaN(outliers)){
    return outliers;
}
```

valueFor(Date) returns a Fact

Returns the latest Fact, chronologically, in a metric or KPI FactList that lands on the specified date.

```
var list = kpis.get('Total AWS Cost');
var start = list.valueFor(now.startOfWeek());
var current = list.valueFor(now);

// the cost difference this week, defaults set to zero
return current.value(0) - start.value(0);
```

This function does not consider the time of day and always returns the latest available Fact on the day. Consider the following example which returns the latest value for a metric on the date of the KPI calculation:

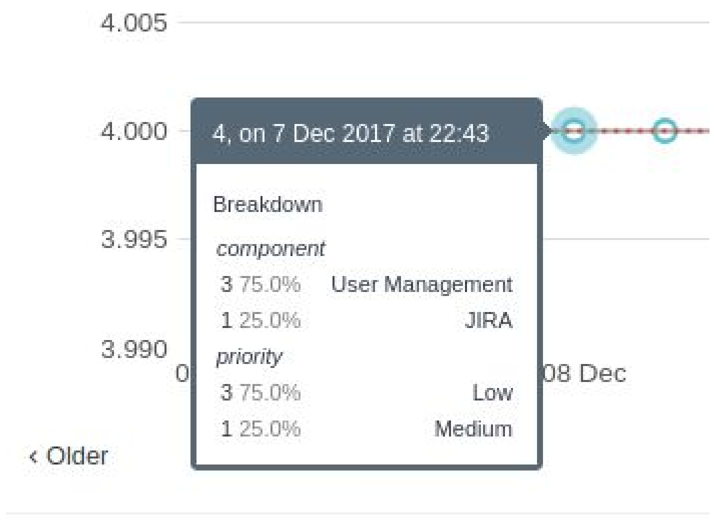
```
var list = metrics.get('Unresolved Issues');
return list.valueFor(now);
```

This is equivalent to:

```
return metrics.get('Unresolved Issues').getLatest();
```

Fact object

A Fact object represents single point of a metric or KPI data, the value and optional metadata at a specific date and time. ServiceClarity collects metric Facts and calculates KPI Facts. The output of all KPI calculation formula is a Fact, which is the value of the KPI on the date of the KPI calculation.



Each point in a KPI or metric data set is a single Fact about that KPI or metric. In the above example we that on the 7th December the value for the KPI is 4 issues and the available metadata is a breakdown by *component* and also by *priority*.

Metric Facts are collected from sources such as JIRA, AWS, custom database, etc. They are collected at intervals throughout the day, as regularly as every 10 minute intervals, or hourly, or as infrequently as once each day. KPI Facts are calculated for a day as metric data becomes available with each calculation refreshing the KPI Fact for that day.

Functions

breakdown() returns a BreakdownSet

Returns the BreakdownSet for this fact or null if no breakdown is available for the Fact.

```
var latest = metrics.get('critical issues').getLatest(); // the latest Fact
var breakdown = latest.breakdown();
console.log('The breakdown metadata for latest fact: %s', breakdown);
```

value(number) returns a decimal

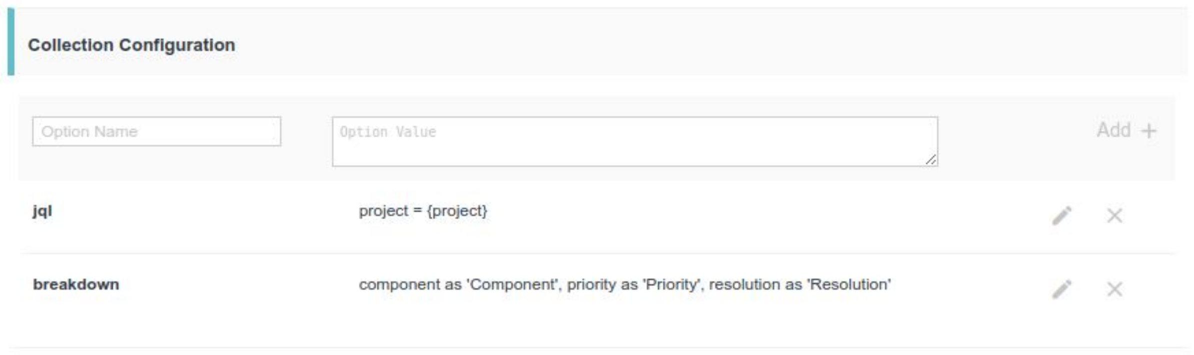
Returns the numeric value for the Fact or the supplied fallback number if the value is null or NaN.

```
var latest = metrics.get('critical issues').getLatest(); // the latest Fact
var value = latest.value(); // could be NaN

console.log('The value for latest fact = %s', value);
if (!isNaN(value)){
    return value
}
```

BreakdownSet object

Each point of data for a metric or a KPI can contain metadata, a breakdown of the value by one or more labels. For example, a count of open issues might be broken down by Issue Type and by Issue Severity. This collection of one or more breakdown is represented by a BreakdownSet object.


Collection Configuration

jql	project = {project}
breakdown	component as 'Component', priority as 'Priority', resolution as 'Resolution'

The breakdown of a metric is determined by the breakdown configuration property of its connection or its own Collection Configuration property as illustrated above. In this example the collected metric Facts will contain breakdown metadata for ***Component***, ***Priority*** and ***Resolution***. This set of three breakdowns is the metrics BreakdownSet.

Functions

asPercentage() returns a BreakdownSet

Create a copy of the BreakdownSet with the Breakdown values converted into a percentage of the total Breakdown. This is a common use or Breakdown metadata where the proportion is more useful than the absolute value.

The following example is a complex KPI formula that uses the % breakdown of Issue Priority to find the % time spent on the highest priority issue.

```
//Get the recorded time metric
var timeSpent = metrics.get('*Time*');
```

```
//Get the breakdown for the time spent, which includes the 'priority'
var breakdown = timeSpent.getLatest().breakdown();

//Convert the priority timings to % of total time
breakdown = breakdown.get("priority").asPercentage();



//Filter the highest priority
breakdown = breakdown.filter("Critical", "High");

//Return the total % time spent on highest priority issues
return {
       'value' : breakdown.sum()
}
```

divide(BreakdownSet) returns a BreakdownSet

Returns a new copy of the BreakdownSet divided by another BreakdownSet. There must be a corresponding Breakdown for each Breakdown in both sets for this function to succeed. The metadata Breakdowns within the set are divided and the results combined into a new BreakdownSet.

The following example illustrates how two related metrics can be divided and, as long as they contain the same metadata, the Breakdown of these metrics can also be divided to calculate a meaningful new BreakdownSet for the resulting output value.

```
var actual = metrics.get('Time Recorded').getLatest();
var estimated = metrics.get('Estimated Time').getLatest();

var accuracy = 100.0 * (actual.value() / estimated.value());
var breakdown = actual.breakdown().divide(estimated.breakdown()).scale(100);

return {
    value : accuracy,
    breakdown : breakdown
   };
```

filter(name, ...) returns a BreakdownSet

Returns a new BreakdownSet filtered so that contains the Breakdowns for the named values. Values are matched case regardless of case and may include wild cards.

```
var issues = metrics.get('critical issues').getLatest();

//filter breakdowns by name and wildcard
var breakdown = issues.breakdown().filter('Severity', '*Type');
```

get(name) returns a Breakdown

Returns the Breakdown metadata for the specific name which may include wildcards. Breakdown name searches are case insensitive.

If a wildcard is used and more than one entry matches the expression the first found will be returned - this is not usually what was intended so care should be taken when using wildcards to ensure that they match a single entry.

```
var issues = metrics.get('critical issues').getLatest();

//find specific breakdown by name
var severity = issues.breakdown().get('Severity');

//find specific breakdown using a wildcard
var type = metrics.get('*Type');
```

isEmpty() returns a boolean

Returns ***true*** if a BreakdownSet contains no entries. If there are any Breakdowns in the set even if they contain no values this function will return ***false***. This is a useful function when developing new KPI formula as it can be used for simple debug logging.

```
var issues = metrics.get('critical issues').getLatest();
if (issues.breakdown().isEmpty()){
```

```
            console.log('No breakdown data for critical issues for date: %s', now);
        }
```

isNull() returns a boolean

Returns ***true*** if a BreakdownSet contains no entries. If there are Breakdowns in the list but they do not contain values this function will return ***true***. This is a useful function when developing new KPI formula as it can be used for simple debug logging.

```
        var issues = metrics.get('critical issues').getLatest();
        if (issues.breakdown().isNull()){
            console.log('No breakdown values for critical issues on: %s', now);
        }
```

merge(BreakdownSet, ...) returns a BreakdownSet

Returns a new BreakdownSet that combines the Breakdown metadata from two or more BreakdownSets. If two Breakdowns have same name their values are merged into the sum of both, otherwise the individual named Breakdowns are combined into a single set.

```
        var opened = metrics.get('Issues Opened');
        var closed = metrics.get('Issues Closed');

        var burndown = opened.latest(0) - closed.latest(0);

        var openedMeta = opened.breakdown().prefixWith('Opened ');
        var closedMeta = closed.breakdown().prefixWith('Closed ');

        var breakdown = openMeta.merge(closedMeta);

        return {
                value : burndown,
                breakdown : breakdown
                };
```

multiply(BreakdownSet) returns a BreakdownSet

Returns a new copy of the BreakdownSet multiplied by another BreakdownSet. There must be a corresponding Breakdown for each Breakdown in both sets for this function to succeed. The metadata Breakdowns within the set are multiplied and the results combined into a new BreakdownSet.

This feature is similar to the BreakdownSet divide function and can be used to create meaningful Breakdown data of two metrics when the output value of the KPI formula is the value of one metric multiplied by another, then the output metadata would be on BreakdownSet multiplied by the other.

```
var adwordPrice = metrics.get('Adword Price').getLatest();
var adwords = metrics.get('Adwords').getLatest();

var cost = adwordPrice.value() * adwords.value();
var breakdown = adwords.breakdown().multiply(adwordPrice.breakdown());

return {
    value : cost,
    breakdown : breakdown
   };
```

rename(originalName, newName) returns a BreakdownSet

Returns a new BreakdownSet with all of the original Breakdowns renamed with the supplied alternative. This is useful when merging two or more BreakdownSets that have the same named Breakdowns but they should be treated as distinct and their values should not be summed.

```
var opened = metrics.get('Issues Opened');
var closed = metrics.get('Issues Closed');

var burndown = opened.latest(0) - closed.latest(0);

var openedMeta = opened.breakdown().filter('Priority').rename('Opened Priority');
var closedMeta = closed.breakdown().filter('Priority').rename(Closed Priority ');

var breakdown = openedMeta.merge(closedMeta);

return {
       value : burndown,
       breakdown : breakdown
       };
```

scale(number) returns a BreakdownSet

Returns a copy of the BreakdownSet with all the metadata values multiplied by the supplied number. This can be useful when a metric value is being scaled to convert from seconds to minutes or hours; the metadata of the metric can be similarly scaled.

```
       var minutes = metrics.get('Time Spent').getLatest();
       var timeInHours = minutes / 60;
       var meta = timeSpent.breakdown().scale(1/60);

       return {
          value : timeInHours,
          Breakdown : meta
       }
```

subtract(BreakdownSet) returns a BreakdownSet

Returns a new copy of the BreakdownSet with another BreakdownSet subtracted from it. There must be a corresponding Breakdown for each Breakdown in both sets for this function to succeed. The metadata Breakdowns within the set are subtracted and the results combined into a new BreakdownSet.

This feature is similar to the BreakdownSet divide function and can be used to create meaningful Breakdown data of two metrics when the output value of the KPI formula is the value of one metric subtracted from another, then the output metadata could be on BreakdownSet subtracted from the other.

```
       var opened = metrics.get('Issues Opened');
       var closed = metrics.get('Issues Closed');
```

```
        var burndown = opened.latest(0) - closed.latest(0);

        var openedMeta = opened.breakdown();
        var closedMeta = closed.breakdown();

        var breakdown = openMeta.subtract(closedMeta);

        return {
               value : burndown,
               breakdown : breakdown
               };
```

valueOf(breakdown, label) returns a number

Returns the numeric value for a specific label in a specific Breakdown. This enabled direct access to and extraction of values from BreakdownSets as a convenience of using Breakdown metadata in the calculation of KPI values.

```
        var invoices = metrics.get('Invoices').getLatest();
        var breakdown = invoices.breakdown();

        var total = invoices.sum();
        var subscriptions = breakdown.valueOf('Account*', 'Subscriptions');

        return (subscriptions / total) * 100;
```

withPrefix(prefix) returns a BreakdownSet

Returns a new BreakdownSet with all of the original Breakdowns renamed with the supplied prefix. This is useful when merging two or more BreakdownSets that have the same named Breakdowns but they should be treated as distinct and their values should not be summed.

```
        var opened = metrics.get('Issues Opened');
        var closed = metrics.get('Issues Closed');

        var burndown = opened.latest(0) - closed.latest(0);

        var openPriority = opened.breakdown().prefixWith('Opened ');
```

```
var closedPriority = closed.breakdown().prefixWith('Closed ');

var breakdown = openPriority.merge(closedPriority);

return {
       value : burndown,
       breakdown : breakdown
       };
```

CalculationDate object

KPIs always track values on a daily basis, their formula are executed and the result is stored for a specific date. As new metrics flow into ServiceClarity and are converted into KPIs this date is normally the current date, ***today***. However, it is often possible to backdate KPI calculation or to recalculate values after updating a KPI formula.

KPI formula are always executed in the context of a calculation date, which is supplied to the formula are the special variable ***now***. This variable is a CalculationDate object and it provides various functions and constants for expressing ***dates and date ranges relative to the date of the calculation***.

For example,

```
var yesterday = now.yesterday();               //a date
var thisDayLastWeek = now.lastWeek();          //a date
var firstOfThisMonth = now.startOfMonth();     //a date
Var thisMonth = now.monthToDate();             //a range of dates
```

The CalculationDate functions and constants can be used to find and filter specific values from metric and KPI FactLists.

Functions

day() returns an integer

Returns the day of the month for the CalculationDate.

```
var date = now.valueOf('2016-01-23');

console.log('Day of the month = %s', date.day());
// Day of the month = 23
```

dayOfWeek() returns an integer

Returns the day of the week for the CalculationDate.

```
var date = now.valueOf('2016-02-22'); // a monday

console.log('Day of the week = %s', date.dayOfWeek());
// Day of the week = 1
```

daysInMonth() returns an integer

Returns the number of days in the month that contains the CalculationDate.

```
var date = now.valueOf('2016-02-22'); // February

console.log('Days in the month = %s', date.daysInTheMonth());
//Days in the month = 28
```

endOfDay() returns a CalculationDate

Returns the number of days in the month that contains the CalculationDate.

```
var date = now.valueOf('2016-02-22'); // February
var endOfDay = date.endOfDay();

console.log('endOfDay = %s', date);
//endOfMonth = 2017-02-22 00:00:00
```

endOfMonth() returns a CalculationDate

Returns the last day in the month that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var endOfMonth = date.endOfMonth();

console.log('endOfMonth = %s', endOfMonth);
//endOfMonth = 2017-03-31 00:00:00
```

endOfPreviousMonth() returns a CalculationDate

Returns the last day in the month before that month that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var endOfMonth = date.endOfPreviousMonth();

console.log('endofLastMonth = %s', endOfMonth);
//endOfLastMonth = 2017-02-28 00:00:00
```

endOfPreviousWeek() returns a CalculationDate

Returns the last day of the week in before the week that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var endOfLastWeek = date.endOfPreviousWeek();

console.log('endOfLastWeek = %s', endOfLastWeek);
//endOfLastWeek = 2017-03-12 00:00:00
```

endOfWeek() returns a CalculationDate

Returns the last day of the week that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var endOfWeek = date.endOfWeek();

console.log('Day of the month = %s', endOfWeek);
//endOfWeek = 2017-03-19 00:00:00
```

endOfYear() returns a CalculationDate

Returns the last day of the year that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var endOfYear = date.endOfYear();

console.log('endOfYear = %s', endOfYear);
//endOfYear = 2017-12-31 00:00:00
```

isEndOfMonth() returns a boolean

Returns the ***true*** if the CalculationDate falls on the end of the month, otherwise ***false***.

```
var date = now.valueOf('2017-03-15');

if (date.isEndOfMonth()){
    //doesn’t run
}
```

isEndOfWeek() returns a boolean

Returns the ***true*** if the CalculationDate falls on the end of the week (Sunday), otherwise ***false***.

```
var date = now.valueOf('2017-03-15');

if (date.isEndOfWeek()){
    //doesn’t run
}
```

isEndOfYear() returns a boolean

Returns the ***true*** if the CalculationDate falls on the end of the year, otherwise ***false***.

```
var date = now.valueOf('2017-03-15');

if (date.isEndOfYear()){
    //doesn’t run
}
```

last10days() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting nine days before the CalculationDate up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.last10days();

console.log('Date range = %s', range);
//Date range = 2017-03-06 00:00:00 -> 2017-03-15 00:00:00
```

last14days() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting 13 days before the CalculationDate up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.last14Days();

console.log('Date range = %s', range);
//Date range = 2017-03-02 00:00:00 -> 2017-03-15 00:00:00
```

last30days() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting 29 days before the CalculationDate up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.last30Days();

console.log('Date range = %s', range);
//Date range = 2017-02-13 00:00:00 -> 2017-03-15 00:00:00
```

last7days() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting 6 days before the CalculationDate up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.last7days();

console.log('Date range = %s', range);
//Date range = 2017-03-08 00:00:00 -> 2017-03-15 00:00:00
```

lastMonth() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate on the same day of the month for the month previous to the one that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var lastMonth = date.lastMonth();

console.log('lastMonth = %s', lastMonth);
//lastMonth = 2017-02-15 00:00:00
```

When the previous month has fewer days and the CurrentDate cannot match the day of the month the date is shifted to always be the end of the previous month.

```
var date = now.valueOf('2017-03-31');
var lastMonth = date.lastMonth();

console.log('lastMonth = %s', lastMonth);
//lastMonth = 2017-02-28 00:00:00
```

lastWeek() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate on the same day of the week for the week previous to the one that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15'); // a Wednesday
var lastWeek = date.lastWeek();

console.log('lastWeek = %s', lastWeek);
//lastWeek = 2017-02-08 00:00:00
```

minus(number, 'days' | 'weeks' | 'Months' | 'years' | ‘hours’ | ‘minutes’ | ‘seconds’)

Returns a new CalculationDate relative to the CalculationDate by the number of units, which can be days, weeks, months or years, etc.

```
var date = now.valueOf('2017-03-15'); // a Wednesday
var before = date.minus(1, ‘day’);

console.log('before = %s', before);
//before = 2017-03-14 00:00:00
```

month() returns an integer

Returns the month of the year for the CalculationDate.

```
var date = now.valueOf('2016-01-23');

console.log('Month of the year = %s', date.month());
// Month of the year = 1
```

monthToDate() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting at the beginning of the month that contains the CalculationDate extending up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.monthToDate();

console.log('Date range = %s', range);
//Date range = 2017-03-01 00:00:00 -> 2017-03-15 00:00:00
```

nextMonth() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate on the same day of the month for the month next to the one that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var nextMonth = date.nextMonth();

console.log('nextMonth = %s', nextMonth);
//nextMonth = 2017-04-15 00:00:00
```

When the previous month has fewer days and the CurrentDate cannot match the day of the month the date is shifted to always be the end of the previous month.

```
var date = now.valueOf('2017-03-31');
var nextMonth = date.nextMonth();

console.log('nextMonth = %s', nextMonth);
//nextMonth = 2017-04-30 00:00:00
```

nextWeek() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate on the same day of the week for the week following the one that contains the CalculationDate.

```
var date = now.valueOf('2017-03-15'); // a Wednesday
var nextWeek = date.nextWeek();

console.log('nextWeek = %s', nextWeek);
//nextWeek = 2017-02-22 00:00:00
```

plus(number, 'days' | 'weeks' | 'Months' | 'years' | ‘hours’ | ‘minutes’ | ‘seconds’)

Returns a new CalculationDate relative to the CalculationDate by the number of units, which can be days, weeks, months or years, etc.

```
var date = now.valueOf('2017-03-15'); // a Wednesday
var after = date.plus(1, ‘day’);

console.log('after = %s', after);
//after = 2017-03-16 00:00:00
```

startOfDay() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted so that the time of day is 00:00:00.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var date = date.startOfDay();

console.log('date = %s', date);
//date = 2017-03-15 00:00:00
```

startOfMonth() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted so that the day of the month is the first.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var date = date.startOfMonth();

console.log('date = %s', date);
//date = 2017-03-01 15:00:00
```

startOfWeek() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted so that the day of the week is a Monday.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var date = date.startOfWeek();

console.log('date = %s', date);
```

```
//date = 2017-03-13 15:00:00
```

startOfYear() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted so that the day of the year is a January the first.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var date = date.startOfYear();

console.log('date = %s', date);
//date = 2017-01-01 15:00:00
```

tomorrow() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted to the following day.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var tomorrow = date.tomorrow();

console.log('tomorrow = %s', date);
//yesterday = 2017-03-16 15:00:00
```

until(date) returns a DateRange

Returns a DateRange that starts at the CalculationDate and continues to the specified date.

```
var date = now.valueOf('2017-03-15');
var nextMonth = date.nextMonth();
var range = date.until(nextMonth);

console.log('range = %s', range);
//range = 2017-03-15 00:00:00 -> 2017-04-15 00:00:00
```

valueOf('yyyy-MM-dd' | ‘yyyy-MM-dd hh:mm:ss’) returns a CalculationDate

Returns a new CalculationDate on the specified date and optionally time. If the time is not supplied in the input it will default to 00:00:00.

```
var date = now.valueOf('2017-03-15');

console.log('date = %s', date);
//date = 2017-03-15 00:00:00

date = now.valueOf('2017-03-15 23:59:59');

console.log('date = %s', date);
//date = 2017-03-15 23:59:59
```

weekToDate() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting at the beginning of the week that contains the CalculationDate extending up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.weekToDate();

console.log('Date range = %s', range);
//Date range = 2017-03-13 00:00:00 -> 2017-03-15 00:00:00
```

withTime(hours, minutes, seconds) returns a CalculationDate

Returns a new CalculationDate on the same date with the specified time.

```
var date = now.valueOf('2017-03-15').withTime(23,59,59);

console.log('Date = %s', date);
//Date = 2017-03-15 23:59:59
```

withTime(hours, minutes, seconds, milliseconds) returns a CalculationDate

Returns a new CalculationDate on the same date with the specified time, including milliseconds.

```
var date = now.valueOf('2017-03-15').withTime(23,59,59,999);
```

```
console.log('Date = %s', date);
//Date = 2017-03-15 23:59:59
```

year() returns an integer

Returns the year that contains the CalculationDate.

```
var date = now.valueOf('2016-01-23');

console.log('Year = %s', date.year());
// Year = 2017
```

yearToDate() returns a DateRange

Returns a new DateRange relative to the CalculationDate starting at the beginning of the year that contains the CalculationDate extending up to and including the CalculationDate.

```
var date = now.valueOf('2017-03-15');
var range = date.yearToDate();

console.log('Date range = %s', range);
//Date range = 2017-00-01 00:00:00 -> 2017-03-15 00:00:00
```

yesterday() returns a CalculationDate

Returns a new CalculationDate relative to the CalculationDate shifted to the day before.

```
var date = now.valueOf('2017-03-15 15:00:00'); // 3pm on a Wednesday
var yesterday = date.yesterday();

console.log('yesterday = %s', date);
//yesterday = 2017-03-14 15:00:00
```