Metrics Framework

User Guide

Table of Contents

[Overview 3](#_Toc485654415)

[Terminology 3](#_Toc485654416)

[Metrics Framework Scoped App 3](#_Toc485654417)

[Setup 3](#_Toc485654418)

[Usage 4](#_Toc485654419)

[Customization 5](#_Toc485654420)

[Performance Analytics 6](#_Toc485654421)

[Indicators 7](#_Toc485654422)

[Breakdowns 7](#_Toc485654423)

[Users, Roles, Groups 8](#_Toc485654424)

[Appendix 10](#_Toc485654425)

[Base Class Methods 10](#_Toc485654426)

[MFCommon 12](#_Toc485654427)

Overview

The Metrics Framework is a scoped application that facilitates the collection of metrics. The goal is to standardize metrics collection into a common set of extendable tables and classes in a way that can be easily adopted by existing applications and integrated into existing reporting structures and dashboards such as SUHA or Performance Analytics.

Terminology

|  |  |
| --- | --- |
| Facts table | A table that is the source for interesting things.  For indicators, this table holds the metrics that you have collected and want to display.  For breakouts, this table holds the list of values that you want to break down. |
| Breakdown field | The name of the field in the facts table that contains the value you want to breakdown. |
|  |  |

Metrics Framework Scoped App

The Metrics Framework scoped application provides a base framework for collecting raw metrics data from your applications. Base (extendable) tables are included for the following key metrics:

|  |  |  |
| --- | --- | --- |
| Metric | Description | Base Table |
| **Usage** | Tracks who is doing what in the application. | x\_snc\_mf\_base\_usage |
| **Execution** | Tracks the execution/status of the application. | x\_snc\_mf\_base\_execution |
| **Performance** | Tracks the execution time of the application. | x\_snc\_mf\_base\_performance |
| **Failures** | Tracks the failures encountered in the application. | x\_snc\_mf\_base\_failure |
| **Metrics Key[[1]](#footnote-1)** | Correlates the other tables with a common Metrics Key. | x\_snc\_mf\_base\_key |

Setup

To start collecting metrics, you need to create a custom class and set of tables[[2]](#footnote-2).

**Tables**

Start by extending the key table and each of the base tables for which you want to collect metrics. You don’t need to do anything with them other than create them as extensions of the MF base tables. Try to maintain a consistent naming convention that follows the standard in the base tables. E.g. If your application’s abbreviation is ‘adc’, then you would create tables:

|  |  |  |
| --- | --- | --- |
| u\_mf\_adc\_key | 🡪 | x\_snc\_mf\_base\_key |
| u\_mf\_adc\_failure | 🡪 | x\_snc\_mf\_base\_failure |

etc.

**Class Script**

Create a Metrics class for your application that extends `x\_snc\_mf.MetricsFrameworkBase`.

Then, define the table names that you just created.

e.g.

**var** AdcMetrics = Class.create();

AdcMetrics.prototype = Object.extendsObject(x\_snc\_mf.MetricsFrameworkBase, {

initialize: **function**() {

x\_snc\_mf.MetricsFrameworkBase.prototype.initialize.call(**this**, "ADC Management");

},

TABLES: {

key: 'u\_mf\_adc\_key',

usage: 'u\_mf\_adc\_usage',

performance: 'u\_mf\_adc\_performance',

execution: 'u\_mf\_adc\_execution',

failure: 'u\_mf\_adc\_failure',

},

type: 'AdcMetrics'

});

**Users and Roles**

In order to provide access to the various pieces, there are a number of roles that need to be created and associated with user groups[[3]](#footnote-3).

To set this up properly, you are encouraged to create two new roles[[4]](#footnote-4):

|  |  |
| --- | --- |
| Role | Access |
| u\_mf\_<app>\_user | READ for each extended table |
| u\_mf\_<app>\_admin | CRUD for each extended table |

You should also create two new groups:

|  |  |
| --- | --- |
| Group | Roles |
| MF <app> User | u\_mf\_<app>\_user |
| MF <app> Admin | u\_mf\_<app>\_admin |

Now, you can either assign these rules to existing groups or assign existing users to these groups as needed.

Usage

Using the framework is as simple as calling the functions where you want to record the metric. Alternatively, MF offers a wrapFunction decorator method to add the metrics collection to a function.

Original function:

**function** addVip() {

**var** vips;

reserveIPs();

vips = createVIPs();

configureDevices();

**return** {"success": vips };

}

with Metrics:

**function** addVip() {

**var** vips;

**var mf = new AdcMetrics();**

**var** action = "addVip";

**mf.usage(action);**

**var perfRecordId = mf.performanceStart("Processing", action);**

reserveIPs();

vips = createVIPs();

configureDevices();

**mf.performanceEnd(perfRecordId, "Finished");**

**mf.execution("Finished", action);**

**return** {"success": vips };

}

wrapped:

**function** addVip() {

**var** vips;

reserveIPs();

vips = createVIPs();

configureDevices();

**return** {"success": vips };

}

**var mf = new AdcMetrics();**

**addVip = mf.wrapFunction(addVip, {}, "addVip")**

Customization

MF’s flexible customization options allow you to add custom columns to your tables and create new metrics, specific to your application.

**Add columns to existing metrics**

One of the advantages of extending the base tables is the ability to add your own columns. Each of the provided metrics allows for an optional `kwargs` argument that will be mapped to your table. Out of the box, when you call the metric function, pass the `kwargs` formatted with the column name as the key and the value as the value.

e.g. If you create a `u\_rack\_type` field in your extended Usage table, then you would call usage as such:

mf.usage(action, {u\_rack\_type: 'My Rack Type'});

By default, the keys to kwargs are expected to map exactly to the column name. However, if you need to modify this, you can override the appropriate `\_convertArgs<Metric>` function to perform the translation.

e.g.

Override the \_convertArgs function:

\_convertArgsUsage: **function**(kwargs) {

**var** newargs = { u\_rack\_type: kwargs['rackType'] };

**return** newargs;

}

Then, in your code:

mf.usage(action, {rackType: 'My Rack Type'});

**Create your own metrics**

1. If you need metrics beyond the included usage, failure, etc, then you can create your own.
2. Create a new metrics table and extend `x\_snc\_mf\_base\_metric`.
3. Add your new table to `TABLES` in your extended Class.

TABLES: {

mymetric: "u\_mf\_mymetric", // extention of x\_snc\_mf\_base\_metric

usage: <etc>

},

1. In your extended Class, add a method for this new metric modeled after the existing metric methods[[5]](#footnote-5).

myMetric: **function**(breakdown, kwargs) {

**var** recordId;

**var** convertFunc = **this**.\_convertArgsMyMetric;

**var** defaultFields = {

u\_metrics\_key: **this**.getMetricsKeyId(),

u\_breakdown: breakdown,

};

recordId = **this**.\_createMetricRecord('mymetric', convertFunc, defaultFields, kwargs);

return recordId;

},

1. In your extended Class, add a conversion method (if you don’t need to convert, just return kwargs):

\_convertArgsMyMetric: **function**(kwargs) **{** **return** kwargs; **}**,

**Custom key lookups**

By default, the Metrics Key is stored with the instance. This means that, for the example above, every `new AdcMetrics`, will generate a unique key for that instance. For some applications, correlation will not work. For example, if your application makes use of multiple asynchronous pieces, each of these might end up generating a new unique Metrics Key. In these cases, you can overload the `\_lookupMetricsKeyId()` and `setMetricsKeyId()` functions. For example, you can modify these functions to set the key per session instead of per instance. Alternatively, if your application has a different datum that can be used for correlation, you can create a new column in your extended key table (e.g. `u\_mf\_adc\_key`) and then modify `\_lookupMetricsKeyId()` to perform the lookup.

Performance Analytics

[Performance Analytics](https://docs.servicenow.com/bundle/istanbul-performance-analytics-and-reporting/page/use/performance-analytics/reference/r_PALandingPage.html) widgets are used to display data in a chart or graph on a dashboard (see [ADC Metrics Dashboard](https://datacenterdev.service-now.com/$pa_dashboard.do?sysparm_dashboard=9f54a6e1db6e72002a4cfd5e0f9619a8&sysparm_tab=79351de7db2a3e00c84e76821f961958&sysparm_cancelable=true&sysparm_editable=false&sysparm_active_panel=false&sysparm_ignore_default)). To get started, you need to:

1. Create an indicator source.
2. Create an indicator (that uses this indicator source).
3. Create a widget to host one or more indicators.
4. Create Daily and Historic Data Collection scheduled jobs (if they don’t already exist for your application).
5. Add the indicator to the scheduled jobs.

Optionally, you can create a breakdown for the data. This will require:

1. A breakdown source.
2. A breakdown (that uses this breakdown source).
3. A breakdown-indicator mapping.

Indicators

Indicators are, essentially, statistics calculated from metrics and are meant to be an “indicator” of a product’s performance. The indicator aggregates the raw metric source data (MF extended tables) into a single number that represents the statistic for that grouping of data. E.g. If the source contains a list of `mf\_error` records for a single day, you might want to configure the indicator with an aggregation strategy of “count” to indicate the number of errors for that day. Likewise, if the source were of `mf\_performance` records for a day, an appropriate aggregation strategy might be “average” to indicate the average execution duration for that day.

Breakdowns

Breakdowns are a way to drill down further into your data, based on additional fields in your facts tables. For example, if you wanted to chart the number of errors encountered per day, you would define an indicator that counted the number of rows in your errors table. Then, if you wanted to see the details for specific types of errors (how many of those were Syntax errors vs Reference errors), you would define a breakdown to match on the error\_type field in your facts table.

**Automatic Breakdowns**

Automatic breakdowns are meant to work with automatic indicators. Just like with indicators, a breakdown needs a source table.

If the breakdown field has an associated choice list, then the breakdown source will point to the `sys\_choice` table[[6]](#footnote-6), matching on the metric facts tableName and breakdown fieldName. This breakdown source is used to create a breakdown, which you can then map to the breakdown fields in your metrics tables.

If, however, your breakdown field does **not** have an associated choice list, then you will have to create a manual breakdown to use as your breakdown source and write a script to perform the mappings.

This can be a bit confusing, so buckle up.

**Manual Breakdowns**

If your breakdown field does not have an associated choice list, then you will need to create a Manual Breakdown to serve as that choice list for the purposes of PA. Create a new breakdown in the `pa\_breakdowns` table with the `type` field set to “Manual”. In the “Manual” tab, list the values that you want to use as breakdown values. These values will actually be stored in a separate table: `pa\_manual\_breakdowns`[[7]](#footnote-7).

Now, you can create a breakdown source, using `pa\_manual\_breakdowns` as your facts table, matching on the condition where `Breakdown` is the manual breakdown you just created. I suggest appending “ (from manual)” to the name of this breakdown source.

Now, you can create an automatic breakdown, using this new breakdown source as the breakdown source.

Finally, you need to create the Breakdown Mappings (click “new” in the appropriate tab on the Breakdown form). However, instead of simply mapping fields, you need to create a script (in `pa\_scripts`) to do the mapping for you. Fortunately, MF has a handy script (mapBreakdownFromManual()) that you can use.

**Examples:**

|  |  |
| --- | --- |
| Indicator Source | [MF ADC Performance today](https://datacenterdev.service-now.com/nav_to.do?uri=pa_cubes.do?sys_id=7ca02301db76b2c0c84e76821f96199d) |
| Indicator | [ADC performance duration per day](https://datacenterdev.service-now.com/nav_to.do?uri=pa_indicators.do?sys_id=a5112b41db76b2c0c84e76821f961949%26sysparm_view=automated) |
| Manual Breakdown | [MF Adc Actions Breakdown](https://datacenterdev.service-now.com/nav_to.do?uri=pa_breakdowns.do?sys_id=d2dfe15bdb6ab600c84e76821f9619db%26sysparm_view=manual) |
| Breakdown Source | [MF Adc Actions (from manual)](https://datacenterdev.service-now.com/nav_to.do?uri=pa_dimensions.do?sys_id=2d78fd17db2eb600c84e76821f9619c5%26sysparm_view=Automated) |
| Automated Breakdown | [MF Adc Performance Actions Breakdown](https://datacenterdev.service-now.com/nav_to.do?uri=pa_breakdowns.do?sys_id=9b93eb89db76b2c0c84e76821f9619de%26sysparm_view=automated) |
| Breakdown Mapping | [MF Adc Performance Actions Breakdown Mapping](https://datacenterdev.service-now.com/nav_to.do?uri=pa_breakdown_mappings.do?sys_id=af94e78ddb76b2c0c84e76821f96196b%26sysparm_view=automated) |
| Breakdown Script | [MF Adc Performance Breakdown Script](https://datacenterdev.service-now.com/nav_to.do?uri=pa_scripts.do?sys_id=c3b4af8ddb76b2c0c84e76821f961932%26sysparm_view=automated) |
| Daily Scheduled Job | [[Cdns] Daily Data Collection](https://datacenterdev.service-now.com/nav_to.do?uri=sysauto_pa.do?sys_id=ad180fa5dbe2b2002a4cfd5e0f96191d) |
| Historical Scheduled Job | [[Cdns] Historic Data Collection](https://datacenterdev.service-now.com/nav_to.do?uri=sysauto_pa.do?sys_id=2fd88329dbe2b2002a4cfd5e0f9619a6) |

Users, Roles, Groups

First, there are two system roles that are required for access to anything related to performance analytics. Any user that wishes to view or modify any dashboards needs at a minimum, one of these. Each of these roles provides access to **all** of performance analytics. Additional roles may be added to restrict access.

|  |  |  |
| --- | --- | --- |
| Role | Access | Add to group |
| pa\_viewer | READ access to all dashboards | MF <app> User |
| pa\_admin | Full access to all dashboards | MF <app> Admin |

However, in order for a user to view a dashboard, in addition to being granted the `pa\_viewer` role, he/she must also be “invited”[[8]](#footnote-8). Users can be invited to the dashboard by clicking the “Sharing” icon from the dashboard and selecting appropriate access for desired roles, groups, and/or users.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 🡪 |  | 🡪 |  |

Appendix

Base Class Methods

The Metrics Framework Base Class is an extendable class that provides the basic functions required to allow an application to collect metrics. The following public functions are available:

class **MetricsFrameworkBase**(productName)

*The base class for MetricsFramework. This is meant to be extended by the application that needs metrics and provides methods for storing basic metrics. Each of the metrics methods accepts a set of required positional arguments required to populate the default base table columns as well as an optional `*kwargs*` argument that is used for custom columns that may be added to the extended metrics tables.*

|  |  |  |
| --- | --- | --- |
| **Parameters:** |  |  |
| productName | *String* | The name of the product.  This should be set when extending the class. |
| **Attributes:** |  |  |
| TABLES | *Object* | A lookup table for extended table names.  You need at least the tables to which your application will write metrics.  If you compose your own metrics, then you will need additional keys. |

Metrics Methods

**usage**(breakdown, kwargs)

*Purpose: Track what users and what they are doing in the application*

|  |  |  |
| --- | --- | --- |
| **Table Fields:** | | |
| u\_breakdown | Required argument | |
| u\_user\_name | Derived, but can be overridden in kwargs | |
| u\_metrics\_key | Derived from `this.getMetricsKeyId()` | |
|  | | |
| **Parameters:** | | |
| breakdown | *String* | Single word to describe what the user is doing.  *e.g. Viewing, Updating, Creating, Launching, Re-Launching, etc* |
| kwargs | *Object* | (optional) A key:value table of additional arguments.  *This will be sent to `*\_convertArgs\**` for extra fields in extended tables.* |
| **Returns:** | *String* | sys\_id of the new record |

**performanceStart**(status, breakdown, kwargs)

*Purpose: Track the execution time of the application*

|  |  |  |
| --- | --- | --- |
| **Table Fields:** | | |
| u\_status | Required argument | |
| u\_breakdown | Required argument | |
| u\_metrics\_key | Derived from `this.getMetricsKeyId()` | |
| u\_start\_time | Derived from new Date().getTime() | |
| u\_end\_time | \*\*not set\*\* (unless provided in kwargs) | |
|  | | |
| **Parameters:** | | |
| Status | *String* | Single word to describe the current status of the application.  *e.g. Processing* |
| breakdown | *String* | Single word to describe what the user is doing.  *e.g. Viewing, Updating, Creating, Launching, Re-Launching, etc* |
| kwargs | *Object* | (optional) A key:value table of additional arguments.  *This will be sent to `*\_convertArgs\**` for extra fields in extended tables.* |
| **Returns:** | *String* | sys\_id of the new record |

**performanceEnd**(recordId, status, kwargs)

*Purpose: Track the execution time of the application*

|  |  |  |
| --- | --- | --- |
| **Table Fields:** | | |
| u\_status | Required argument | |
| u\_breakdown | \*\*not set\*\* (unless provided in kwargs) | |
| u\_metrics\_key | \*\*not set\*\* (unless provided in kwargs) | |
| u\_start\_time | \*\*not set\*\* (unless provided in kwargs) | |
| u\_end\_time | Derived from new Date().getTime() | |
|  | | |
| **Parameters:** | | |
| recordId | *String* | The sys\_id of the record created with `performanceStart()`. |
| Status | *String* | Single word to describe the current status of the application.  *e.g. “Processing”* |
| kwargs | *Object* | (optional) A key:value table of additional arguments.  *This will be sent to `*\_convertArgs\**` for extra fields in extended tables.* |
| **Returns:** | *String* | sys\_id of the new record |

**execution**(status, breakdown, kwargs)

*Purpose: Track the execution/failures of the automations as a summary of what happened.*

|  |  |  |
| --- | --- | --- |
| **Table Fields:** | | |
| u\_status | Required argument | |
| u\_breakdown | Required argument | |
| u\_metrics\_key | Derived from `this.getMetricsKeyId()` | |
|  | | |
| **Parameters:** | | |
| Status | *String* | Single word to describe the current status of the application.  *e.g. Finished, Failed* |
| breakdown | *String* | Single word to describe what the user is doing.  *e.g. Viewing, Updating, Creating, Launching, Re-Launching, etc* |
| kwargs | *Object* | (optional) A key:value table of additional arguments.  *This will be sent to `*\_convertArgs\**` for extra fields in extended tables.* |
| **Returns:** | *String* | sys\_id of the new record |

**failure**(errorType, errorMsg, breakdown, kwargs)

*Purpose: Track the execution/failures of the automations as a summary of what happened.*

|  |  |  |
| --- | --- | --- |
| **Table Fields:** | | |
| u\_error\_type | Required argument | |
| u\_error\_message | Required argument | |
| u\_breakdown | Required argument | |
| u\_metrics\_key | Derived from `this.getMetricsKeyId()` | |
|  | | |
| **Parameters:** | | |
| errorType | *String* | The type of Error  *When you catch an error `catch (error)`, this would be the value of error.name.*  *e.g. SyntaxError, PctErrors.InvalidCredentials, etc* |
| errorMsg | *String* | The error message.  *When you catch an error `catch (error)`, this would be the value of error.message.* |
| breakdown | *String* | Single word to describe what the user is doing.  *e.g. Viewing, Updating, Creating, Launching, Re-Launching, etc* |
| kwargs | *Object* | (optional) A key:value table of additional arguments.  *This will be sent to `*\_convertArgs\**` for extra fields in extended tables.* |
| **Returns:** | *String* | sys\_id of the new record |

MFCommon

The **x\_snc\_mf.MFCommon** class provides some convenient functions to help in creating PA and SUHA dashboard items.

**mapBreakdownFromManual**(currentGr, breakdownId, breakdownField)

*Finds the sys\_id of the record in the* pa\_manual\_breakdowns *table that corresponds to the desired breakdown value.*

|  |  |  |
| --- | --- | --- |
| **Parameters:** | | |
| CurrentGR | *GlideRecord* | The metrics record |
| breakdownId | *String* | The sys\_id of the manual breakdown record (pa\_breakdowns) that holds the list of breakdown values. |
| breakdownField | *String* | The name of the field in your metric facts table that contains the breakdown values. |
| **Returns:** | *String* | the sys\_id of the record in the pa\_manual\_breakdown table that corresponds to the breakdown element |

This is meant to be used in a pa\_scripts record to map the breakdown fields. The global var `current` maps to the current record in the metrics facts table.

**Example:**

    var id = 'd2dfe15bdb6ab600c84e76821f9619db';

    var fieldName = 'u\_breakdown';

    var sysID = x\_snc\_mf.MFCommon.mapBreakdownFromManual(current, id, fieldName);

    sysID || '';

**calcPerfHealthBreakdown**(ds, performanceField, targetJson, breakdownField)

*Calculates a % health value for a performance metric on the SUHA dashboards.*

***\*\*Specific for SUHA metrics\*\****

*The value is derived as "The % of executions that completed within the specified target duration". This also allows for any breakdowns to be evaluated against targets specific to each breakdown. The idea is that for each breakdown (e.g. addPool, delPool, addVip, etc), we could assign a different standard for "acceptable performance". If that target threshold is breached, then we know that something is running slowly.*

***\*\*Important\*\*****: When setting up your source, be sure that `Is Aggregated` is \*****not****\* checked!*

|  |  |  |
| --- | --- | --- |
| **Parameters:** | | |
| ds | *GlideRecord* | The data set if there is a selected Data Source.  This is passed to the script in the `u\_suha\_group\_metric\_definition` which is where this script is called from. |
| performanceField | *String* | The sys\_id of the manual breakdown record (pa\_breakdowns) that holds the list of breakdown values. |
| targetJson | *Object* | The target duration, in ms, for performance metrics.  **Format:**  {  "default": <default target>,  "<breakdown 1>": <target for breakdown 1>,  "<breakdown 2>": <target for breakdown 2>,  }  This value will be used to generate a performance health metric (e.g. n% of executions completed under the target).  The value is a JSON blob that must, at least, contain a "default" key. Other keys can be added that map to other breakdowns against which those will be measured.  e.g. If you specify an `addVip` key, then all addVip records in the `u\_mf\_adc\_performance` table will be required to finish under that value in order to be counted toward the Success tally.  All other records would be evaluated against the value of the "default" key. |
| breakdownField | *String* | The name of the field in your metric facts table that contains the breakdown values. |
| **Returns:** | *Float* | A number representing the %health for the collection of records. |

**Example:**

**var** params = **new** AdcParameters();

**var** targetJson = JSON.parse(params.get('mf.performance.targetDuration'));

    result = x\_snc\_mf.MFCommon.calcPerfHealthBreakdown(ds, 'u\_duration', targetJson, 'u\_breakdown');

**calcPerfHealthBreakdown**(ds, performanceField, targetValue)

*A simplified version of calcPerfHealthBreakdown without breakdowns.*

***\*\*Specific for SUHA metrics\*\****

***\*\*Important\*\*****: When setting up your source, be sure that `Is Aggregated` is \*****not****\* checked!*

|  |  |  |
| --- | --- | --- |
| **Parameters:** | | |
| ds | *GlideRecord* | The data set if there is a selected Data Source.  This is passed to the script in the u\_suha\_group\_metric\_definition which is where this script is called from. |
| performanceField | *String* | The field in the record that holds the duration value to be evaluated. |
| targetValue | *Integer* | The target duration, in ms, for performance metrics. |
| **Returns:** | *Float* | A number representing the %health for the collection of records. |

**Example:**

**var** target = 1500;

result = x\_snc\_mf.MFCommon.calcPerfHealth(ds, 'u\_duration', target);

1. Not a metric, but included here for convenience. The Metrics Key table correlates the metrics in the other tables. [↑](#footnote-ref-1)
2. The plan is to eventually automate all of the setup into a service catalog or UI page. [↑](#footnote-ref-2)
3. There is a MF Admin group that provides full admin access to the MF scoped application’s tables. You should not need this unless you are developing the scoped application, but it provides a good example model. [↑](#footnote-ref-3)
4. You can also go ahead and delete any default roles that were created for each table and use these as an aggregate role covering all tables. [↑](#footnote-ref-4)
5. TODO: create a new wrapper function to generate this function. [↑](#footnote-ref-5)
6. When configuring the breakout source, the table selection is called a “Facts table”, which is not to be confused with the facts table that holds your collected metrics. [↑](#footnote-ref-6)
7. This will feel very similar to creating columns in a table. [↑](#footnote-ref-7)
8. Permissions are stored in the pa\_dashboards\_permissions table. [↑](#footnote-ref-8)