# Grafana Web API Interfaces

January 15, 2024

Contents

[Grafana Web API Interfaces 1](#_Toc156548223)

[Naming Conventions: 1](#_Toc156548224)

[Query Data Operation 2](#_Toc156548225)

[Query Data UI Considerations 3](#_Toc156548226)

[Query Data Source *Value Types* Operation 4](#_Toc156548227)

[Query Data Source *Value Type Tables* Operation 5](#_Toc156548228)

[Query Data Source *Value Type Table Fields* Operation 5](#_Toc156548229)

[Query Data Source *Value Type* *Functions* Operation 6](#_Toc156548230)

[Search Metadata Operation 7](#_Toc156548231)

[Query Annotations Operation 8](#_Toc156548232)

[Reload Data Source Value Types Operation 8](#_Toc156548233)

[Reload Grafana Functions Operation 8](#_Toc156548234)

## Naming Conventions:

* JSON Data Type: *Italizied*
* JSON Field Names: “double-quoted”
* JSON Descriptions: <datatype: in angle brackets>
* C# Named Entity: ‘single-quoted’

## Query Data Operation

**URL:** /Query (POST)

**Body**: *QueryRequest*

**Response:** *TimeSeriesValues* array

*QueryRequest*:

{

“dataTypeIndex”: <int: target data source value type index>.

“range”: <*Range*: requested time range>,

“interval”: <string: interval of request – includes units, e.g., “200ms”>,

“maxDataPoints”: <int: maximum data points to return>,

“targets”: <*Target* array: set of requested targets \*>,

“adhocFilters”: <*AdHocFilter* array: ad-hoc filters to apply>,

“excludedFlags”: <uint: any data quality flags to exclude from query results>,

“excludeNormalFlags”: <bool: flag indicating if normal flags should be excluded \*\*>

}

\* A *QueryRequest* might only have a single item in the “targets” array representing the entire user query expression – there is no need for UI to break anything up (e.g., splitting by semi-colon, etc.) since the backend code already handles this. However, if a single query request can be used for multiple data source queries, i.e., where each query in the panel (often labeled “A”, “B”, “C”, etc.) can be referenced by its label as part of the *Target* where “refID” is the query name identifier – then multiple targets can be defined in a single query. It should be noted that UI supports a “-- Mixed --” mode for data sources, in which case each query has its own assignable data source, in these cases the UI code would need to ensure that the data source was the same for each *Target* in the *QueryRequest*.

\*\* Exclusion of normal is a special case since bit flags for normal are zero, i.e., the only discernible test for normal is if “flags == 0”, so if user does not want normal flags, then “excludeNormalFlags” should be set to true.

*Range*:

{

“from”: <string: from time ISO>,

“to”: <string: to time ISO>

}

*Target*:

{

“refID”: <string: query name identifier assigned by Grafana (user editable)>,

“target”: <string: query expression, e.g., point tags and filter expressions>,

“metadataSelections”: <*MetadataSelections* array: selected metadata to return with query>

}

*MetadataSelections*:

{

“tableName”: <string: source table for selected metadata>,

“fieldNames”: <string array: selected metadata field names>

}

*AdHocFilter*:

{

“key”: <string: filter key>,

“operator”: <string: filter operator to apply, e.g., “<”, “<=”, “>”, etc.>,

“value” <string: filter value>

}

*TimeSeriesValues*:

{

“target”: <string: time-series value point, derived from expression>,

“rootTarget”: <string: root point tag for expression, if applicable>,

“refID”: <string: matching query name identifier from *Target*>,

“syntaxError”: <string: message indicating an expression syntax error \*>,

“metadata”: <map<string, string>: “field name” to “field value” map \*\*>,

“datapoints”: <double array of arrays: values and time, field content based on value type>

}

\* Other fields, e.g., “metadata” and “datapoints”, will be empty if “syntaxError” is not empty. This field only indicates errors in the expression syntax, not other kinds of exceptions. When a *TimeSeriesValues* is returned with a “syntaxError”, there may be only one *TimeSeriesValues* instance in the array representing the “target” with the syntax error; the “refID” will match one of the targets specified in the *QueryRequest*.

\*\* The returned values are based on user selections as defined in the *MetadataSelections* of each *Target* in the *QueryRequest*. Field name will not include table name if it is unique in the return set, otherwise, field name will be formatted as “TableName.FieldName”.

### Query Data UI Considerations

Command level parameters can be added to a query expression to perform custom operations. Consider the following:

|  |  |
| --- | --- |
| Command | Description |
| DropEmptySeries | Hides any empty series from display. Example:  ; dropEmptySeries |
| IncludePeaks | Ensures decimated data includes both min/max interval peaks for derived data source (requires full resolution back-end query, reducing performance). Example:  ; includePeaks |
| FullResolutionQuery | Requests a full resolution query from derived data source (reduces query performance). Example:  ; fullResolutionQuery |
| RadialDistribution | When defined, updates overlapping “longitude” and “latitude” metadata values to have a radial distribution. Settings include *radius*, *zoom* and *tolerance* defined as key value pairs. Example:  ; radialDistribution={radius=1.5; zoom=2; tolerance=0.000275} |

Although a user can add these manually to a text-based query expression, having UI options, e.g., using checkboxes / options, which would add these settings automatically to the target expression on behalf of the user would be useful.

Note command level parameters should only be added to the end of a query expression being separated by semi-colons. Also, radial distribution processing only executes anything when “Longitude” and “Latitude” metadata have been selected by the user – UI could auto-select these metadata fields if they exist and option is selected.

## Query Data Source *Value Types* Operation

**URL:** /GetValueTypes (POST)

**Body**: <empty>

**Response:** *DataSourceValueType* array

When defining a new Grafana openHistorian data source, end-user will need to be provided with a selection of possible supported data types. There are currently two built-in types: ‘DataSourceValue’ and ‘PhasorValue’, however more can be loaded dynamically and more may be added in the future, e.g., ‘DigitalValues’ and ‘AnalogValues’, which could provide an enhanced user experience for metadata and data type value processing for these types.

*DataSourceValueType:*

{

“name”: <string: type name of structure that implements ‘IDataSourceValue’ >,

“index”: <int: index of the data source value type in loaded value-types array \*>,

“timeSeriesDefinition”: <string: comma delimited string of time-series field names \*\*>,

“metadataTableName”: <string: name of primary metadata table>

}

\* The “index” is a key field in the returned *ValueType*. As an optimization, all data source value type references are accessed by index. This index operates with an array for fastest value type lookups instead of each request processing a string-based dictionary lookup by name. Note that a “dataTypeIndex” field will be a required key input for most requests -- this value will always match the target *ValueType* “index” field, as such this value should be tracked with the data type selection when configuring a new data source.

\*\* The “timeSeriesDefinition” is a list of field names for the values that get returned from the “dataPoints” in the *TimeSeriesValues* structure, the response to a *QueryRequest*, see Query Data Operation for details. For example, the ‘DataSourceValue’ type returns {"Value", "Time"}, and the ‘PhasorValueType’ returns {"Magnitude", "Angle", "Time"}. It is expected that this property could be used to help the UI determine the field composition of the *TimeSeriesValues* “dataPoints” return value at run-time without needing to hard code anything related to a new data type. Additionally, this might be useful information to show the end-user when selecting a data source value type while setting up a new Grafana data source to help them better understand the nature of the time-series data that will be returned for that data type. Note that code requests these values be ordered by values first, then a time, i.e., a single timestamp value will always be last value in the “dataPoints” list.

## Query Data Source *Value Type Tables* Operation

**URL:** /GetValueTypeTables (POST)

**Body**: *SearchRequest*

**Response:** string array

When “dataTypeIndex” is a valid index for a selected data source value type, response is an array of table names that the selected data source value type can support, i.e., each returned table name, at a minimum, contains all the fields that the value type has defined as required, see the ‘RequiredMetadataFieldNames’ property in the ‘IDataSourceValue’ interface. The returned list will always at least include the primary metadata table as defined in the *ValueType* “metadataTableName” from the Query Data Source *Value Types* Operation response. When UI exposes these values in a dropdown list and there is more than one value in the list, the “metadataTableName” value should be the initial selected value. When “dataTypeIndex” is -1, all available metadata tables are returned.

*SearchRequest:*

{

“dataTypeIndex”: <int: target data source value type index, or, -1 for all metadata tables>,

“expression”: <string: unused for value type tables request \*>

}

\* For a query of data source value type tables, expression is ignored and can be an empty string.

## Query Data Source *Value Type Table Fields* Operation

**URL:** /GetValueTypeTableFields (POST)

**Body**: *SearchRequest*

**Response:** *FieldDescription* array

*SearchRequest:*

{

“dataTypeIndex”: <int: target data source value type index, or, -1 for any metadata table>,

“expression”: <string: table name for fields request>

}

*FieldDescription*:

{

“name”: <string: name of field in requested table>,

“type”: <string: data type name of the field, e.g., “string”, “int”, etc.>,

“required”: <bool: flag that indicates if field is required by data source value type>

}

## Query Data Source *Value Type* *Functions* Operation

**URL:** /GetValueTypeFunctions (POST)

**Body**: *SearchRequest*

**Response:** *FunctionDescription* array

Functions are commonly defined for all data source value types, however, in some cases a function may not exist for a specific value type since its operation would be meaningless or confusing. Additionally, end-user custom functions can be created and dynamically loaded that may only support a specific set of value types. As a result, the “dataTypeIndex” is a required parameter to get a proper list of available functions.

*SearchRequest:*

{

“dataTypeIndex”: <int: target data source value type index>,

“expression”: <string: group operation filter for value type functions request \*>

}

\* For a query of data source value type functions, expression is used to filter functions by group operation, specifically a value of ‘None’, ‘Slice’, or ‘Set’ as defined in the ‘GroupOperations’ enumeration. If all function descriptions are desired, regardless of group operation, an empty string can be provided. Combinations are also supported, e.g., ‘Slice,Set’. For general UI guidance, one thought was having a drop-down selection for group operations, perhaps with values of “None”, “Slice”, “Set”, and “All” as options for filtering function selection list on the UI. Regardless, the thinking is that these options should accommodate most use cases.

*FunctionDescription*:

{

“name”: <string: name of function>,

“description”: <string: description of function>,

“aliases”: <string array: other names the function supports>,

“allowedGroupOperations”: <string: list of allowed group operations>,

“publishedGroupOperations”: <string: list of published group operations \*>,

“parameters”: <*ParameterDescription* array: defined function parameters \*\*>

}

\* List of function descriptions returned by the Query Data Source *Value Type* *Functions* Operation will only ever include those targeted for publication, i.e., visible to the end-user, minus any applied filters from the *SearchRequest* “expression”. The API, per function configuration, will “allow” more operations than it will publish to the user, for example, if a user manually enters “SliceFloor” into a text expression, it will not throw an exception; but this will not be an available option for selection since the calculated result for “Floor” and “SliceFloor” result in same result matrix when slice tolerances match incoming data rate. Normally, the published group operations will be a subset of the allowed group operations, but the “Evaluate” function is an exception to this rule. The “Evaluate” function always operates as a slice, but the user is not required to enter “SliceEvaluate” to use the function; as such, the function always appears in the list with the name “Evaluate” but includes a slice tolerance parameter. If filtering is applied in the *SearchRequest*, “Evaluate” will appear for both ‘None’ and ‘Slice’ filtered groups, but in both cases will only have the name “Evaluate”.

\*\* List of defined parameters will include *required* parameters first (*ParameterDescription* “required” = ‘true’), followed by any *optional* parameters (*ParameterDescription* “required” = ‘false’). All functions also include a target expression, e.g., a list of point-tags, sub-functions, or filter expressions that the function will operate on, which is always the last parameter.

*ParameterDescription*:

{

“name”: <string: name of the parameter>,

“description”: <string: description of the parameter>,

“type”: <string: data type name of the parameter, e.g., “string”, “int”, etc.>,

“required”: <bool: flag indicating if the parameter is required>,

“default”: <string: string-based representation of the default value>

}

## Search Metadata Operation

**URL:** /Search (POST)

**Body**: *SearchRequest*

**Response:** string array

*SearchRequest:*

{

“dataTypeIndex”: <int: target data source value type index>,

“expression”: <string: target search expression \*>

}

\* Target search expression will execute a search against metadata available to data source value type referenced by the *SearchRequest* “dataTypeIndex”. The search has two modes of operation for target expression:

1. ‘SELECT’ statement
2. ‘LIKE’ statement

Operation attempts to parse provided expression as a ‘SELECT’ statement first. If this succeeds, metadata available to referenced data source value type will be filtered from in memory data set (not subject to SQL injection issues), returning specified results. Any valid table that supports the data value type’s ‘RequiredMetadataFieldNames’ will be available for query. If ‘SELECT’ specifies multiple fields, values will be comma delimited in each array result – this can include ‘\*’ to return all fields. This is a useful operation for defining a selection variable in Grafana that should be pulled from metadata, e.g., a device list. If an empty expression is provided, all point tags will be returned, up to ‘MaximumSearchTargetsPerRequest’ property value. When “dataTypeIndex” is -1, all available metadata tables can be searched (select only).

If expression is determined to not be a ‘SELECT’ statement, then statement is assumed to be a search request, hence a ‘LIKE’ statement. In this case, only the data type value’s defined primary ‘MetadataTableName’ will be searched since user has no way of specifying any other table name. The following is the “LIKE” statement that gets executed against provided expressions:

$"ID LIKE '{InstanceName}:%' AND PointTag LIKE '%{target}%'"

The ‘InstanceName’ is provided by the ‘GrafanaDataSourceBase’ derived implementation, for example, this is often “STAT” or “PPA”. The maximum returned records is limited to the defined ‘MaximumSearchTargetsPerRequest’ property which defaults to two hundred, but limit can be overridden by ‘GrafanaDataSourceBase’ derived implementation. Note that returned values for a ‘LIKE’ statement are always the matching “PointTag” field values from the data type value’s defined primary table as defined in the *ValueType* “metadataTableName” from the Query Data Source *Value Types* Operation response.

## Query Annotations Operation

**URL:** /Annotations (POST)

**Body**: *AnnotationRequest*

**Response:** *AnnotationResponse* array

*AnnotationRequest:*

{

“annotationQuery”: <string: annotation query expression>,

“range”: <*Range*: requested time range>

}

*AnnotationResponse:*

{

“title”: <string: annotation title>,

“time”: <double: (start) time of annotation in in Unix epoch milliseconds>,

“endTime”: <double: end time of annotation in in Unix epoch milliseconds>,

“text”: <string: text of the annotation>,

“tags”: <string: tags associated with the annotation>

}

NOTE: These structures may be updated for additional functionality in the future.

## Reload Data Source Value Types Operation

**URL:** /ReloadValueTypes (GET)

**Response:** HTML text response indicating success or failure.

Expected to be called from an admin restricted end point to request reload of available data source value types, e.g., end user adds a new DLL that contains a new data source value type implementation and wants a hot refresh without restarting host service.

## Reload Grafana Functions Operation

**URL:** /ReloadGrafanaFunctions (GET)

**Response:** HTML text response indicating success or failure.

Expected to be called from an admin restricted end point to request reload of available Grafana functions, e.g., end user adds a new DLL that contains a new Grafana function implementation and wants a hot refresh without restarting host service.