GlideRecord Query Cheat Sheet

Idoubt if there’s a single concept in Service-now that is more valuable to understand than how to use GlideRecord methods to query, insert, update, and delete records in your system. These methods have a wide variety of uses and are found at the heart of many of the business rules, UI actions, and scheduled job scripts that are essential to tie together your organization’s processes in your Service-now instance.

While the content of this post isn’t new information (additional examples can be found on the [Service-now wiki](http://wiki.service-now.com/index.php?title=GlideRecord)), my aim is to provide a single page of information containing some common examples of these methods as a reference. This is an excellent page to keep bookmarked!

Note: These methods are designed for use in server-side JavaScript (everything EXCEPT client scripts and UI policies). In some rare cases, it may be necessary to perform a query from a client-side javascript (client script or UI policy). The few methods below that can be used in client-side JavaScript have been noted below.

**Query**

Can also be used in Client scripts and UI policies

A standard GlideRecord query follows this format.

**var** gr = **new** GlideRecord('incident'); *//Indicate the table to query from*  
*//The 'addQuery' line allows you to restrict the query to the field/value pairs specified (optional)*  
*//gr.addQuery('active', true);*  
gr.query(); *//Execute the query*  
while (gr.next()) { *//While the recordset contains records, iterate through them*  
   *//Do something with the records returned*  
   **if**(gr.category == 'software'){  
      gs.log('Category is ' + gr.category);  
   }  
}

**UPDATE: This same function applies to client-side GlideRecord queries! If at all possible, you should use an *asynchronous* query from the client.** [**See this post**](https://www.servicenowguru.com/scripting/client-scripts-scripting/gform-getreference-callback/) **for details.**

**var gr = new GlideRecord('sys\_user');  
gr.addQuery('name', 'Joe Employee');  
gr.query(myCallbackFunction); *//Execute the query with callback function*  
  
*//After the server returns the query recordset, continue here*  
function myCallbackFunction(gr){  
   while (gr.next()) { *//While the recordset contains records, iterate through them*  
      alert(gr.user\_name);  
   }  
}**

**‘Get’ Query Shortcut (used to get a single GlideRecord)**

Can also be used in Client scripts and UI policies IF YOU ARE GETTING A RECORD BY SYS\_ID.

The ‘get’ method is a great way to return a single record when you know the sys\_id of that record.

**var** gr = **new** GlideRecord('incident');  
gr.**get**(sys\_id\_of\_record\_here);  
*//Do something with the record returned*  
**if**(gr.category == 'software'){  
   gs.log('Category is ' + gr.category);  
}

You can also query for a specific field/value pair. The ‘get’ method returns the first record in the result set.

*//Find the first active incident record*  
**var** gr = **new** GlideRecord('incident');  
**if**(gr.**get**('active', **true**)){  
   *//Do something with the record returned*  
   gs.log('Category is ' + gr.category);  
}

**‘getRefRecord’ Query Shortcut (used to get a single GlideRecord referenced in a reference field)**  
The ‘getRefRecord’ method can be used as a shortcut to query a record populated in a reference field on a record.

**var** caller = current.caller\_id.getRefRecord(); *//Returns the GlideRecord for the value populated in the 'caller\_id' field*  
caller.email = 'test@test.com';  
caller.update();

**‘OR’ Query**  
The standard ‘addQuery’ parameter acts like an ‘and’ condition in your query. This example shows how you can add ‘or’ conditions to your query.

*//Find all incidents with a priority of 1 or 2*  
**var** gr = **new** GlideRecord('incident');  
**var** grOR = gr.addQuery('priority', 1);  
grOR.addOrCondition('priority', 2);  
gr.query();  
while (gr.next()) {  
   *//Do something with the records returned*  
   **if**(gr.category == 'software'){  
      gs.log('Category is ' + gr.category);  
   }  
}

Note that you can also chain your ‘OR’ condition as well, which is usually simpler

*//Find all incidents with a priority of 1 or 2*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('priority', 1).addOrCondition('priority', 2);  
gr.query();

**Insert**  
Inserts are performed in the same way as queries except you need to replace the ‘query()’ line with an ‘initialize()’ line as shown here.

*//Create a new Incident record and populate the fields with the values below*  
**var** gr = **new** GlideRecord('incident');  
gr.initialize();  
gr.short\_description = 'Network problem';  
gr.category = 'software';  
gr.caller\_id.setDisplayValue('Joe Employee');  
gr.insert();

**Update**  
You can perform updates on one or many records simply by querying the records, setting the appropriate values on those records, and calling ‘update()’ for each record.

*//Find all active incident records and make them inactive*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('active',**true**);  
gr.query();  
while (gr.next()) {  
   gr.active = **false**;  
   gr.update();  
}

**Delete**  
Delete records by performing a glideRecord query and then using the ‘deleteRecord’ method.

*//Find all inactive incident records and delete them one-by-one*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('active',**false**);  
gr.query();  
while (gr.next()) {  
   *//Delete each record in the query result set*  
   gr.deleteRecord();  
}

**deleteMultiple Shortcut**  
If you are deleting multiple records then the ‘deleteMultiple’ method can be used as a shortcut

*//Find all inactive incidents and delete them all at once*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('active', **false**);  
gr.deleteMultiple(); *//Deletes all records in the record set*

**addEncodedQuery**

CANNOT be used in Client scripts and UI policies! Use ‘addQuery(YOURENCODEDQUERYHERE)’ instead.

An alternative to a standard query is to use an encoded query to create your query string instead of using ‘addQuery’ and ‘addOrCondition’ statements. An easy way to identify the encoded query string to use is to create a filter or a module with the query parameters you want to use, and then hover over the link or breadcrumb and look at the URL. The part of the URL after ‘sysparm\_query=’ is the encoded query for that link.  
So if I had a URL that looked like this…  
https://demo.service-now.com/incident\_list.do?sysparm\_query=active=true^category=software^ORcategory=hardware

My encoded query string would be this…  
active=true^category=software^ORcategory=hardware

I could build that encoded query string and use it in a query like this…

*//Find all active incidents where the category is software or hardware*  
**var** gr = **new** GlideRecord('incident');  
**var** strQuery = 'active=true';  
strQuery = strQuery + '^category=software';  
strQuery = strQuery + '^ORcategory=hardware';  
gr.addEncodedQuery(strQuery);  
gr.query();

**GlideAggregate**  
[GlideAggregate](http://wiki.service-now.com/index.php?title=GlideAggregation) is actually an extension of the GlideRecord object. It allows you to perform the following aggregations on query recordsets…  
-COUNT  
-SUM  
-MIN  
-MAX  
-AVG

*//Find all active incidents and log a count of records to the system log*  
**var** gr = **new** GlideAggregate('incident');  
gr.addQuery('active', **true**);  
gr.addAggregate('COUNT');  
gr.query();  
**var** incidents = 0;  
**if** (gr.next()){  
   incidents = gr.getAggregate('COUNT');  
   gs.log('Active incident count: ' + incidents);  
}

**orderBy/orderByDesc**  
You can order the results of your recordset by using ‘orderBy’ and/or ‘orderByDesc’ as shown below.

*//Find all active incidents and order the results ascending by category then descending by created date*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('active', **true**);  
gr.orderBy('category');  
gr.orderByDesc('sys\_created\_on');  
gr.query();

**addNullQuery/addNotNullQuery**  
‘addNullQuery’ and ‘addNotNullQuery’ can be used to search for empty (or not empty) values

*//Find all incidents where the Short Description is empty*  
**var** gr = **new** GlideRecord('incident');  
gr.addNullQuery('short\_description');  
gr.query();

*//Find all incidents where the Short Description is not empty*  
**var** gr = **new** GlideRecord('incident');  
gr.addNotNullQuery('short\_description');  
gr.query();

**getRowCount**  
‘getRowCount’ is used to get the number of results returned

*//Log the number of records returned by the query*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('category', 'software');  
gr.query();  
gs.log('Incident count: ' + gr.getRowCount());

Although ‘getRowCount’ isn’t available client-side, you can return the number of results in a client-side GlideRecord query by using ‘rows.length’ as shown here…

*//Log the number of records returned by the query*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('category', 'software');  
gr.query();  
alert('Incident count: ' + gr.rows.length);

**setLimit**  
‘setLimit’ can be used to limit the number of results returned

*//Find the last 10 incidents created*  
**var** gr = **new** GlideRecord('incident');  
gr.orderByDesc('sys\_created\_on');  
gr.setLimit(10);  
gr.query();

**chooseWindow**  
The chooseWindow(first,last) method lets you set the first and last row number that you want to retrieve and is typical for chunking-type operations. The rows for any given query result are numbered 0..(n-1), where there are n rows. The first parameter is the row number of the first result you’ll get. The second parameter is the number of the row after the last row to be returned. In the example below, the parameters (10, 20) will cause 10 rows to be returned: rows 10..19, inclusive.

*//Find the last 10 incidents created*  
**var** gr = **new** GlideRecord('incident');  
gr.orderByDesc('sys\_created\_on');  
gr.chooseWindow(10, 20);  
gr.query();

**setWorkflow**  
‘setWorkflow’ is used to enable/disable the running of any business rules that may be triggered by a particular update.

*//Change the category of all 'software' incidents to 'hardware' without triggering business rules on updated records*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('category', 'software');  
gr.query();  
while(gr.next()){  
   gr.category = 'hardware';  
   gr.setWorkflow(**false**);  
   gr.update();  
}

**autoSysFields**  
‘autoSysFields’ is used to disable the update of ‘sys’ fields (Updated, Created, etc.) for a particular update. This really is only used in special situations. The primary example is when you need to perform a mass update of records to true up some of the data but want to retain the original update timestamps, etc.

*//Change the category of all 'software' incidents to 'hardware' without updating sys fields*  
**var** gr = **new** GlideRecord('incident');  
gr.addQuery('category', 'software');  
gr.query();  
while(gr.next()){  
   gr.category = 'hardware';  
   gr.autoSysFields(**false**);  
   gr.update();  
}

**setForceUpdate**  
‘setForceUpdate’ is used to update records without having to change a value on that record to get the update to execute. ‘setForceUpdate’ is particularly useful in situations where you need to force the recalculation of a calculated field for all records in a table or when you need to run business rules against all records in a table but don’t want to have to change a value on the records.  
This method is often used with ‘setWorkflow’ and ‘autoSysFields’ as shown below.

*//Force an update to all User records without changing field values*  
**var** gr = **new** GlideRecord('sys\_user');  
gr.query();  
while (gr.next()) {  
   gr.setWorkflow(**false**); *//Do not run business rules*  
   gr.autoSysFields(**false**); *//Do not update system fields*  
   gr.setForceUpdate(**true**); *//Force the update*  
   gr.update();  
}

**JavaScript Operators**  
The following operators can be used in addition to the standard field/value query searching shown above…

|  |  |  |
| --- | --- | --- |
| = | Field value must be equal to the value supplied. | addQuery('priority', '=', 3); |
| > | Field must be greater than the value supplied. | addQuery('priority', '>', 3); |
| < | Field must be less than the value supplied. | addQuery('priority', '<', 3); |
| >= | Field must be equal to or greater than the value supplied. | addQuery('priority', '>=', 3); |
| <= | Field must be equal to or less than the value supplied. | addQuery('priority', '<=', 3); |
| != | Field must not equal the value supplied. | addQuery('priority', '!=', 3); |
| STARTSWITH | Field must start with the value supplied. The example shown on the right will get all records where the short\_description field starts with the text 'Error'. | addQuery('short\_description', 'STARTSWITH', 'Error'); |
| ENDSWITH | Field must end with the value supplied. The example shown on the right will get all records where the short\_description field ends with text 'Error'. | addQuery('short\_description', 'ENDSWITH', 'Error'); |
| CONTAINS | Field must contain the value supplied anywhere in the field. The example shown on the right will get all records where the short\_description field contains the text 'Error' anywhere in the field. | addQuery('short\_description', 'CONTAINS', 'Error'); |
| DOES NOT CONTAIN | Field must not contain the value supplied anywhere in the field. The example shown on the right will get all records where the short\_description field does not contain the text 'Error' anywhere in the field. | addQuery('short\_description', 'DOES NOT CONTAIN', 'Error'); |
| IN | Field must contain the value supplied anywhere in the string provided. | addQuery('sys\_id', 'IN', '0331ddb40a0a3c0e40c83e9f7520f860,032ebb5a0a0a3c0e2e2204a495526dce'); |
| INSTANCEOF | Retrieves only records of a specified class for tables which are extended. For example, to search for configuration items (cmdb\_ci table) you many want to retrieve all configuration items that are have are classified as computers. The code uses the INSTANCEOF operator to query for those records. |  |