Undocumented APIs, Functions, & Properties

> ***NOTE**: This page is under **extreme construction**. Some of the links don't work, and some things that should be linked, are not. That's because this page is **not yet public**. If you found this page, it's because you clicked one of the rather sneaky links hidden within some of our blog articles. Bravo. You get early access to this page, while we work on it. We may indeed **never actually publish this page**, as **undocumented APIs** are not something we want to have to maintain **documentation** for. That would be silly.*
> *Anyway, have fun, and sorry for the mess.*

This page is dedicated to the undocumented APIs, functions, and properties that we've run into and found useful in ServiceNow. This page is always under review, and will change as often as we find new undocumented APIs. Have you got one you'd like to tell us about? Click **Contact Us** at the top of this page to let us know!

Client side undocumented APIs are much easier to discover. since we can just poke around in the code until we find them. Sometimes this leads to the discovery of a matching server-side API, but it would be fair to say that we'll have a lot more client side undocumented APIs, than server-side ones.

I got tired of discovering that the only (or at least, the easiest) way to do many of the things that I wanted to do was with undocumented APIs, so I decided to build a page dedicated to them. Most of these APIs, functions, and properties have existed for a long while, and are likely to stick around. However, it is important to remember that **undocumented APIs are not guaranteed to be stable, or to exist after the next patch**. For that reason, it would be very wise to (with as much irony as you like), document any scripts in which you've used undocumented APIs.

Personally, I recommend only calling undocumented code inside of **try { }** blocks, so that you can **catch()** any errors that crop up after an upgrade. You may even want to use the method in our post about making your log entries easier to find when building your catch code.

- Server-Side APIs
  - GlideSchedule
  - UINotification
  - GlideSysAttachment
  - GlideSysAttachmentInputStream
  - GlideChecksum
    - calculateMD5CheckSum.
  - GlideSecurityManager
    - setUser
- Server-Side Properties
  - Under Construction

Server-Side APIs

GlideSchedule

Note: This API is now documented on the ServiceNow Wiki.

UINotification

//UINotification (Documentation under Construction)

GlideSecurityManager

GlideSecurityManager is a class that allows you to do some things with user sessions and security. It can be instantiated as follows:

```
var gsm = GlideSecurityManager.get();
```

setUser

setUser (a method of GlideSecurityManager) allows you to re-cache the roles and permissions of a given user. If you've recently changed a user's permissions, this will re-cache their new roles without them having to log out and then back in. This could, with a little scripting, be used in a business rule on the sys_user_has_role m2m table, to prevent users from having to log out and back in whenever their roles change.

setUser accepts one argument: a **GlideUser** object. This is a type of object that is **different** from a GlideRecord object that contains a sys_user record. I know of two easy ways to get a user record:

1. Use **gs.getUser()** - This directly returns a **GlideUser** object for whoever the script is running for (the currently active user... if you ran the script, this would get **your** user object).
2. Use gs.getUser() to instantiate a user object, then use a method of GlideUser (getUserByID()) to retrieve another user.

```
var gsm = GlideSecurityManager.get(); //get GSM
var userObj = gs.getUser(); //get a user object with the current user
gsm.setUser(userObj); //reset the user's session
/* Alternate usage: */
var gsm = GlideSecurityManager.get(); //get GSM
var userObj = gs.getUser(); //get a user object with the current user
userObj.getUserByID('user_id_here'); //Get the specified user's GlideUser object
gsm.setUser(userObj); //reset the user's session
```

Server-Side Properties

//This section under construction

Client-Side APIs

Event

The Event class is an undocumented API provided by ServiceNow on the client side, which effectively acts as a layer between our code, and direct control of the **Document Object Model** (DOM) in some ways. Probably the method you're most likely to use, is **.observe()**. The observe method takes as input, 3 arguments:

1. An HTML element control for a field (which you can get using g_form.getControl(); as seen below)
2. A string containing an event name.
    1. From digging into the guts of this class, it seems to implement EventTarget.**addEventListener**(). For this reason, any event name that is listed in the addEventListener documentation *should* be supported here.
3. A callback function to execute when that event is observed.

```
function onLoad() {
    var control = g_form.getControl('description');
    Event.observe(control, 'mouseover', function() {
```

```
        g_form.addInfoMessage('Your mouse is over the Description field.');
    });
    Event.observe(control, 'mouseout', function() {
        g_form.clearMessages();
    });
    Event.observe(control, 'mousedown', function() {
        g_form.addInfoMessage('You have clicked the description field.');
    });
    Event.observe(control, 'mouseup', function() {
        g_form.clearMessages();
    });
}
```

The event class also contains methods like **.stopObserve()**, which ends the current Event.observe on a given HTML object/control. It might be helpful to instantiate this class first. It's also got functions to check what sort of click, for instances where you might observe for the 'click' event: **isLeftClick()**, **isMiddleClick()**, and **isRightClick()**.

getControl

The undocumented **.getControl()** method of the **g_form** object returns the HTML element for the specified field name. This allows a high degree of direct control over the UI, including the ability to monitor that field for events (like **mouseover** or **click**), and run scripts when those events occur. See the example for more details.

The getControl method accepts one argument: The **field name** to get the HTML element of, as a **string**.

Example

```
function onLoad() {
    var control = g_form.getControl('description');
```

```
    Event.observe(control, 'mouseover', function() {
        g_form.addInfoMessage('Your mouse is over the Description field.');
    });
    Event.observe(control, 'mouseout', function() {
        g_form.clearMessages();
    });
    Event.observe(control, 'mousedown', function() {
        g_form.addInfoMessage('You have clicked the description field.');
    });
    Event.observe(control, 'mouseup', function() {
        g_form.clearMessages();
    });
}
```

getDateFromFormat

This undocumented function attempts to parse a date (or date/time) from a given string. It accepts 2 arguments:

1. A string containing the value to be checked
2. A string with the user's preferred date (or date/time) format in ISO8601 standard format

If getDateFromFormat() **can** parse a date from the passed string, according to the format provided in the second argument, it will return the time (in milliseconds) since the UNIX epoch. However, if it **cannot** parse a date, it returns zero. This makes it a useful tool for doing client-side validation of date fields (such as in a **validation script**), which otherwise are often only validated or rejected on the server.

Example

```
function checkIfDateValid(date) {
    if (getDateFromFormat(date, g_user_date_format) != 0) {
        return true; //date is valid
    }
    else if (getDateFromFormat(date, g_user_date_time_format) != 0) {
        return true;
    }
    else {
        return false;
    }
}
//Dates
checkIfDateValid('2015-02-14'); //returns TRUE
checkIfDateValid('2015-2-14'); //returns FALSE
checkIfDateValid('2-14-2015'); //returns FALSE
checkIfDateValid('2015/2/14'); //returns FALSE
//Date/times
checkIfDateValid('2015-02-14 01:02:03'); //returns TRUE
checkIfDateValid('2015-02-14 01:02'); //returns FALSE
checkIfDateValid('2-14-2015'); //returns FALSE
checkIfDateValid('2015-02-14 01:02:03 AM'); //returns FALSE
```

GlideDialogForm

GlideDialogForm is a class which when instantiated, allows you to open the form for any table in a GlideDialog window, allow the user to create or modify record, and then return some information about that record and the action that was taken.
GlideDialogForm accepts 3 arguments:

1. A string for the name of the dialog

2. A string with the name of the table to open the form for
3. A callback function to be called when the form in the dialog is submitted

The callback function itself can take no arguments if you like, or it can accept four:

1. The name of the action (as in UI Action action) that was clicked (this would be like the **save**, **update**, or **delete** buttons)
2. The sys_id of the record that was created or updated
3. The table name
4. The display value for the record that was created or updated

The GlideDialogForm prototype also has a method called **setLoadCallback()**, which allows you to optionally specify a callback function to be called when the dialog loads. *That* callback function accepts one argument, which is the HTML element of **the frame itself**.

The inner HTML frame containing the form itself (which has the g_form and other client-accessible objects) is inside that element. In all browsers except IE, that inner frame is called **defaultView**. However, in IE (because it can't follow standards for some reason), it is called **parentWindow**. In our example, we add a "crumple zone" to account for IE's unique snowflake complex.

Example

```
//This script would go into a UI action with the 'client' tick-box checked, and the 'Onclick
function launch() {
    var dialog = new GlideDialogForm('Create an Incident', 'incident', myCallBack);
    dialog.addParm('sysparm_view', 'ess'); //show self-service view
    //Remove this line to visit the new incident form:
    dialog.addParm('sys_id', '47064b68a9fe19810186793eefffc9b7'); //Could also use dialog.se
    dialog.addParm('sysparm_form_only', 'true'); //don't show related lists
    dialog.setLoadCallback(myLoadCallBack); //This will execute ON LOAD in the dialog.
    dialog.render(); //display dialog
}
```

```
function myCallBack(actionName, sid, tableName, recordDisplayVal) {
    console.log(actionName + ' || ' + sid + ' || ' + tableName + ' || ' + recordDisplayVal);
    //We can set values on the form that called this dialog using g_form at this point, if w
}

function myLoadCallBack(frame) {
    //crumple zone for IE because it can't follow standards:
    var innerDialog = 'defaultView' in frame ? frame.defaultView : frame.parentWindow;
    //The following executes in the context of the form, and gets the incident number.
    var num = innerDialog.g_form.getValue('number');
    //We could even use this or other info on the form to set values in the form that called
`
```

setVariablesReadOnly

//g_form.setVariablesReadOnly()
//under construction

Client-Side Properties

g_user_date_format

This property contains a string representing the user's date preferred date format. The default value for this, is **yyyy-MM-dd**. For example, "2015-2-14" would be Valentine's day in 2015.

Example

```
function checkIfDateValid(date) {
    if (getDateFromFormat(date, g_user_date_format) != 0) {
        return true; //date is valid
    }
    else {
        return false; //date does not match user's selected format.
    }
}
checkIfDateValid('2015-02-14'); //returns TRUE
checkIfDateValid('2015-2-14'); //returns FALSE
checkIfDateValid('2-14-2015'); //returns FALSE
checkIfDateValid('2015/2/14'); //returns FALSE
checkIfDateValid('pickles'); //returns FALSE
```

g_user_date_time_format

This property contains a string representing the user's date preferred date format. The default value for this, is **yyyy-MM-dd HH:mm:ss**. For example, "2015-2-14 01:02:03" would be 1:02:03 AM on Valentine's day in 2015.

This property could be used in a function along with

Example

```
function checkIfDateTimeValid(date) {
    if (getDateFromFormat(date, g_user_date_time_format) != 0) {
        return true; //date/time is valid
    }
    else {
```

```
            return false; //date/time does not match user's selected format.
        }
    }
    checkIfDateTimeValid('2015-02-14 01:02:03'); //returns TRUE
    checkIfDateTimeValid('2015-02-14 01:02'); //returns FALSE
    checkIfDateTimeValid('2-14-2015'); //returns FALSE
    checkIfDateTimeValid('2015-02-14 01:02:03 AM'); //returns FALSE
    checkIfDateTimeValid('pickles'); //returns FALSE
```