COGNIZANT DN 3.0 Deep Skilling Program

ServiceNow

Week 4: ServiceNow Scripting Fundamentals

Submitted by: Vansh Nirwan
Superset ID: 5104096
Registered mail id: vansh.nirwan06@gmail.com

Open source learning video1: Introduction to Servicenow Scripting:

- Scripting in SN used to introduce new functionality,enhance features of existing applications, interaction with 3rd party apps and to automate business processes. Scripts in servicenow is run either on
  a) Client side(run on browser, used to deal with UI forms like showcasing field message, info message on form, making fields read only/ mandatory; does not deal with databases, only alter appearance of forms)
  b) Serverside(script run on backend, server side scripts are mostly invoked upon database actions, ACL processing, script includes, script actions)
  c) MID Server
- Scripting can significantly affect instance performance and hence must be used only if necessary. If 80% of a requirement can be solved using existing low code no code tools, avoid scripting in such cases
- Scripting done on Servicenow inbuild **syntax editor** that provides contextual help, syntax coloring, formatting, auto completion of braces and quotes and debugging functionalities; **syntax editor enabled by default for instances**
- Programming language used for scripting in SN: Javascript; JS provides several API's for both client side and server side

CLIENT SIDE SCRIPTING:

- Used to make cosmetic changes on form UI; runs on the supported browser
- Client scripts can affect performance of instance as it may delay form loading due to time required for processing; hence use minimal client scripts ● Types on client side scripts:
  a) onLoad→ initiated upon form loading, before transferring control to user; may used to prerequisite populate some field values
  b) onChange→script initiated and run if a particular field undergoes changes only due to user actions
  c) onSubmit—> used to validate form entries by a user before submitting the record to the database
  d) OnCellEdit→ used to monitor a particular field and execute the script if the field's values changes due to user actions in a list view
- Client script trigger→onLoad, onChange, onSubmit or onCellEdit; action→ javascript code
- If a table has multiple client side script the order of executing the scripts depends on value of the **order field** on the client script form; lower value executed first
- **In onChange type> the onChange script does not execute if the form is freshly loaded ot the new value set for the field concerned is a null value ●**

Catalog client scripts apply only to the catalog items ● Client side API:

a) GlideForm: (g_form)--> used to access form fields; has access to form properties(fields) and methods
b) GlideUser: (g_user)--> API used to access the details; properties and methods relevant to the current logged in user/ session user
c) Scratchpad: (g_scratchpad)-->API used to temporarily store values retrieved by the display business rule and make it available for the client script

- G_form methods: getValue(), setValue(), showFieldMsg(), addInfoMessage(), addOption(), flash(), clearValue(), isNewRecord()
- Ex: var short_desc=g_form.getValue('short_description')--> by default the getValue() output is in string format, to get int/decimal values> getIntValue()
- Glideuser: g_user→ details about current logged in user
- g_user.getFullName(), hasRole(), hasRoleExactly(), hasRoleFromList(), hasRoles()...
- Client side debugging: alert(), try/catch, response time indicator(if time required to process exceeds set time> implies the client script is faulty)
- UI policies> used to hide form fields, make them mandatory or read only
- Can be done with built in condition builder and setting UI actions, else use scripting in advanced view
- Scripts of UI policy have 2 script options:
  a) Execute if true
  b) Execute if false
- The if false script executes if condition set is not satisfied and the reverse if false is selected
- UI policy scripting can also use g_form, g_user and g_scratchpad
- **Catalog UI policy** used for catalog data

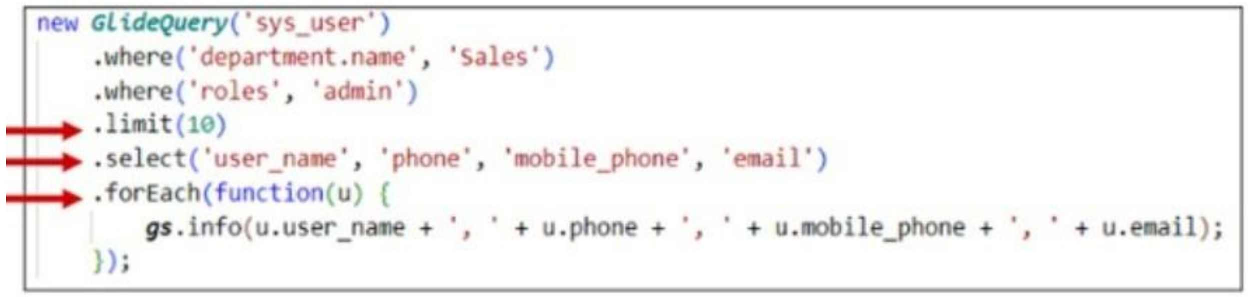
SERVER SIDE SCRIPTING:

- Business rules run on server side when database is manipulated or queried
- Business rules triggers: insert→ run when record is inserted, update→ run when record is updated, filter conditions and role condition
- In advanced view of BR→
  a) When: when the business rule has to run
  b) Order: order of execution of business rule when multiple rules exist for the table
  c) Delete-> run when record deleted

    d) query-> run when db queried

- BR rule objects current, previous, g_scratchpad
- Business rules: when to run
    A) Before→ before database is queried, action is synchronous ie current business rule must execute first before other rules execute(prevents user from seeing certain records)
    B) Display→ display database is used by client scripts to query db data from server side> server side data is fetched from db and loaded into an empty g_scratchpad object. The client side script can access the results stored in the g_scratchpad
    C) After—> after business rule executes after database is queried
    D) Async—> async BR, executes asynchronously and does not block other user operations
- BR advanced view allows scripting options
- Debugging: trycatch, script debugger, tracer, console debugging, Glidesystem methods
- GlideSystem→ server side API referred to as **gs** ● Glidesystem options:
    A) User methods: getUser(), getUserID(), hasRole(), hasRoleInGroup()
    B) System methods: getProperty(), getReference(), log(){not used in scoped apps}, print(), debug(), eventQueue()
    C) date and time: beginningOfLastWeek(), endOfLastWeek(), beginningOfNextMonth(), endOfNextMonth(), nowDateTime(), minutesAgo(), now()
- GlideRecord: used to query data from database
- Var records=new GlideRecord('<table_name>')
- Alternate to SQL
- GlideRecord execution:
    a) Var my_obj=new GlideRecord('incident')
    b) my_obj.addQuery('active','=','true)
    c) my_obj.query()/ _query()--> query data
    d) while(my_obj.next()){}
    e) For update: in the while(my_obj.next()){<make the change> my_obj,update() }
- To add more queries using or condition: q1.addQuery().addOrCondition()
- Every addQuery() is concatenated with **and** condition
- For querying single record: use my_obj.get(<condition>)
- GlideAggregate()--> aggregate functions like count/ use getRowCount()

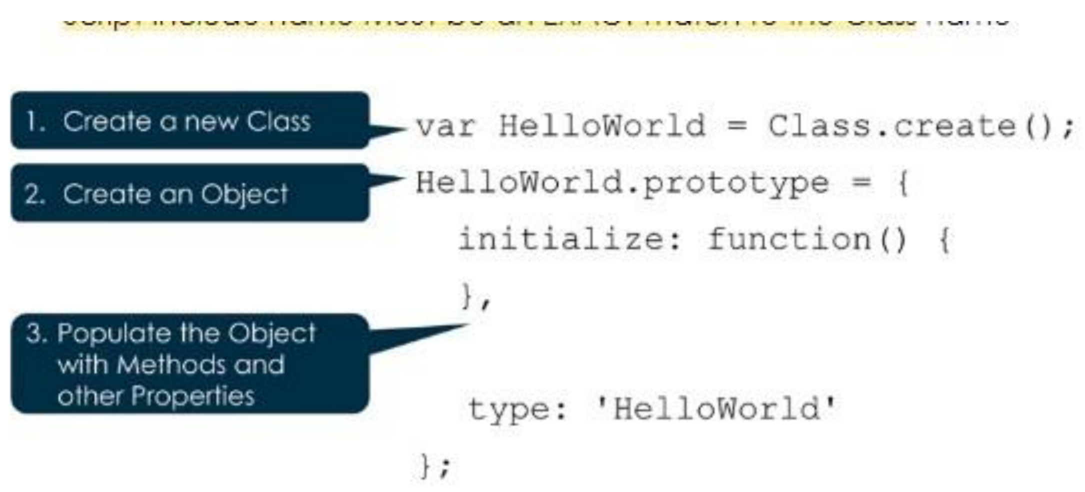
- addEncodedQuery()--> copy filter added to list using condition builder and include in the parentheses
- Alternate to gliderecord> glide query
- GlideQuery> 100% JS, fail fast, be expressive
- Issue with gliderecord> field checking not done, if query wrong the addQuey is skipped



```
new GlideQuery('sys_user')
    .where('department.name', 'Sales')
    .where('roles', 'admin')
    .limit(10)
    .select('user_name', 'phone', 'mobile_phone', 'email')
    .forEach(function(u) {
        gs.info(u.user_name + ', ' + u.phone + ', ' + u.mobile_phone + ', ' + u.email);
    });
```

SCRIPT INCLUDES:

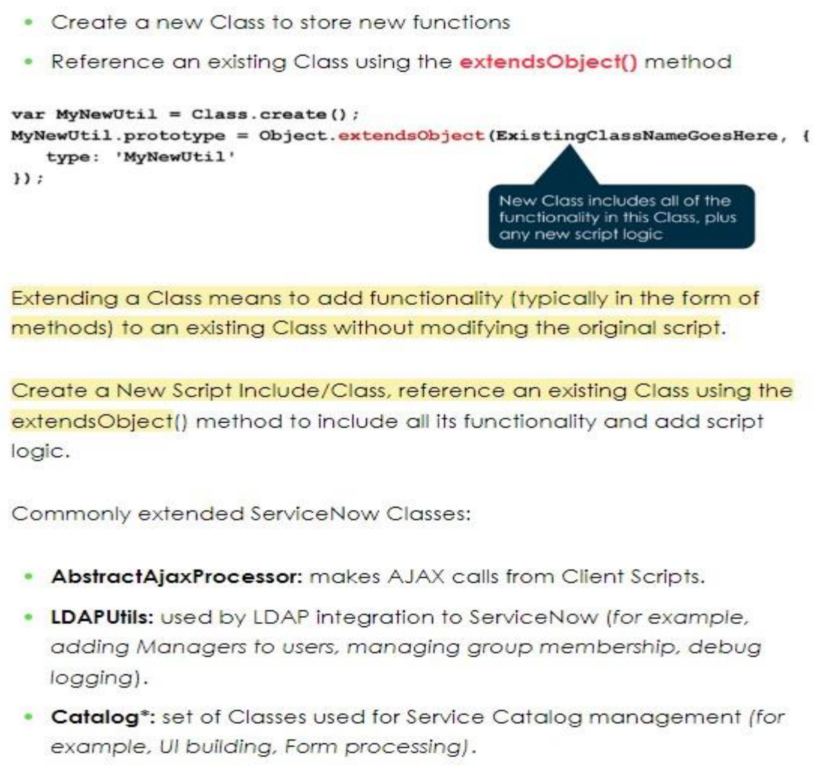
- Reusable code can be stored and invoked in script
- This code does not consume any space and remains dormant unless called upon
- Script include can be function or a class
- Execute on the server side and can be callable from client side
- Name of script include must be same as name of function or class; name must not include any spaces or special characters ● Script include types:
  - A) One function
  - B) Collection of function> class
  - C) Extend from class
- **One function not callable from client side**
- If 2 functions included in one function SI> then second one executes only after first function is completely executed
- Class based SI→ client callable



```
var HelloWorld = Class.create();
HelloWorld.prototype = {
  initialize: function() {
  },

  type: 'HelloWorld'
};
```

- Reference qualifiers: filters the data records available for reference fields ● Types:
  a) simple-> static filters using condition builder
  b) Dynamic→ dynamic query against reference field
  c) Advanced→directly filter reference qual field ● Extending existing class:



- Create a new Class to store new functions
- Reference an existing Class using the **extendsObject()** method

```
var MyNewUtil = Class.create();
MyNewUtil.prototype = Object.extendsObject(ExistingClassNameGoesHere, {
   type: 'MyNewUtil'
});
```

New Class includes all of the functionality in this Class, plus any new script logic

Extending a Class means to add functionality (typically in the form of methods) to an existing Class without modifying the original script.

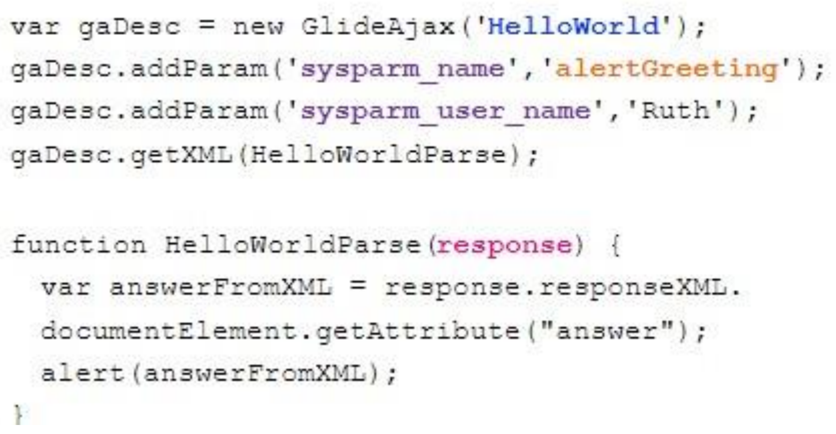
Create a New Script Include/Class, reference an existing Class using the extendsObject() method to include all its functionality and add script logic.

Commonly extended ServiceNow Classes:

- **AbstractAjaxProcessor:** makes AJAX calls from Client Scripts.
- **LDAPUtils:** used by LDAP integration to ServiceNow (*for example, adding Managers to users, managing group membership, debug logging*).
- **Catalog*:** set of Classes used for Service Catalog management (*for example, UI building, Form processing*).

- Abstract ajax processor: client callable —> used to receive data from the server

- glideAjax class> enables client script and UI policies to call server side code in SI ● Add parameters to the glideAjax object using addParam() function
- The glideAjax returns XML response—> extract response from the answer attribute

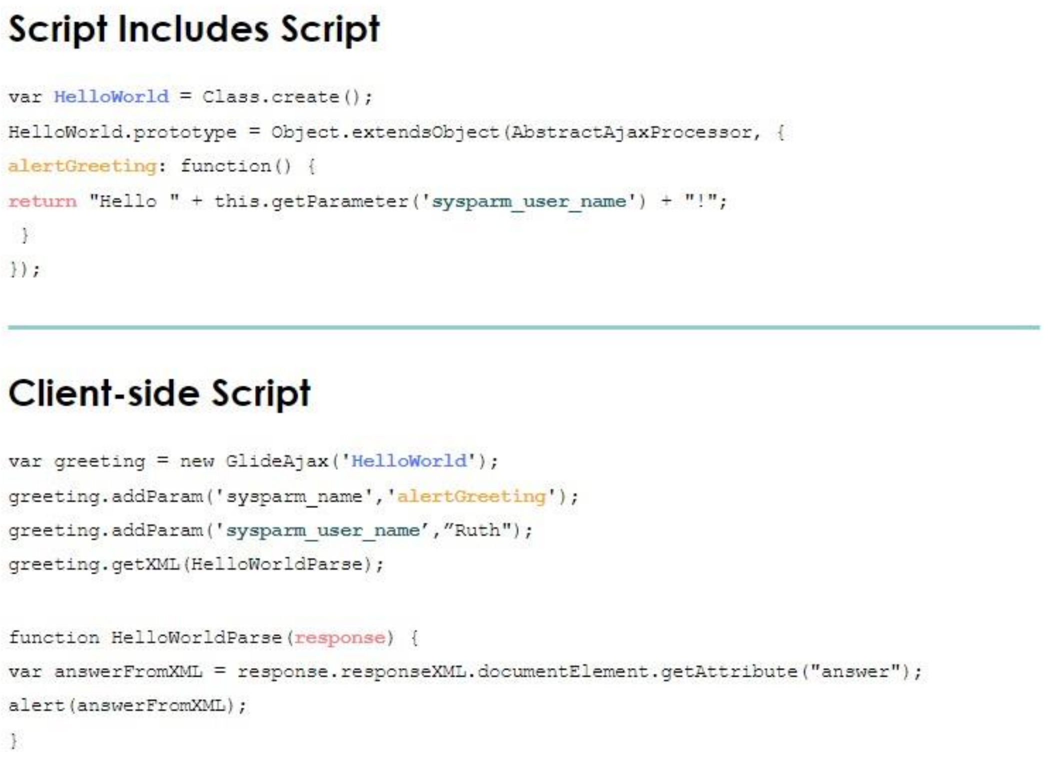


```
var gaDesc = new GlideAjax('HelloWorld');
gaDesc.addParam('sysparm_name','alertGreeting');
gaDesc.addParam('sysparm_user_name','Ruth');
gaDesc.getXML(HelloWorldParse);

function HelloWorldParse(response) {
  var answerFromXML = response.responseXML.
  documentElement.getAttribute("answer");
  alert(answerFromXML);
}
```

a

- Sysparm_name: **script include function name**
- Sysparm_: all other parameters passed
- getXML()--> get XML response
- getXMLAnswer()--> get the answer directly from the XML



Script Includes Script

```
var HelloWorld = Class.create();
HelloWorld.prototype = Object.extendsObject(AbstractAjaxProcessor, {
alertGreeting: function() {
return "Hello " + this.getParameter('sysparm_user_name') + "!";
 }
});
```

Client-side Script

```
var greeting = new GlideAjax('HelloWorld');
greeting.addParam('sysparm_name','alertGreeting');
greeting.addParam('sysparm_user_name',"Ruth");
greeting.getXML(HelloWorldParse);

function HelloWorldParse(response) {
var answerFromXML = response.responseXML.documentElement.getAttribute("answer");
alert(answerFromXML);
}
```

- To get JSON result> use json.stringify()
- On client side: json.parse(response)