crudeSP guide

Introduction

CrudeSP (csp) is a JavaScript library which eases handling list items in Sharepoint lists and document libraries. CrudeSP provides all the functions to create, update delete and view list entries as well as uploading to document libraries. It also comes with its own CAML-builder to select data efficiently.

Requirements and integration

CrudeSP is a service for Sharepoint and can only run on Sharepoint Apps and Solutions. It uses client side Sharepoint CSOM and REST services to interact with the host system and jQuery. In order to get crudeSP working you need to load Sharepoint's JavaScript files (SP.Core.js, SP.Runtime.js), jQuery and the csp.js file (or csp-min.js file in production) within your project.

```
<script type="text/javascript" src="/_layouts/15/zeiterfassung2015/scripts/jquery-2.1.1.js" />
<script type="text/javascript" src="/_layouts/15/SP.Runtime.js" />
<script type="text/javascript" src="/_layouts/15/SP.Core.js" />
<script type="text/javascript" src="/_layouts/15/project/scripts/csp.js" />
```

Using crudeSP

CrudeSP as a query provider orientates itself roughly on SQL's syntax, allowing you to use the basic 'select', 'where' and 'orderBy' keywords for data selection. In order to explain the query structure we use a Sharepoint list called 'test' with its visible columns 'Title' and 'salary'. Its contents are:

Title	salary
Anthony	9999
Bert	2000
Clara	5000

A basic csp-query to return all the data of 'test':

```
var op = new csp.Operation({
    site: "https://mySharepoint.corperation.com/project",
    select: "Title, salery",
    list: "test",
    debug: true
});

op.readItems(function (output) {
    console.log(output.length + " lines returned");
},function (message) { console.log(message); });
```

>> 3 lines returned

Firstly a new csp.Operation-object called ‘op’ is instantiated with the properties ‘site’, ‘select’, ‘list’ and ‘debug’. Property ‘site’ sets the Sharepoint-site of the desired list. If an app runs on the same site where the list is stored on, you don’t have to set ‘site’ and csp infers the current site context for you. In ‘select’ you can define the columns to return. Attention, these columns are case-sensitive and will throw errors if the case or spelling is not right. ‘list’ defines the Sharepoint list or library to retrieve data from and ‘debug’: true adds a debug mode which throws more precise error messages, helping you debug your queries.

Secondly you call an operation method from the instantiated object ‘op’. Following methods are available:

- readItems(success, failure)
- updateItems(success, failure)
- deleteItems(success, failure)
- createItems(success, failure)
- uploadToLibrary(success, failure)

These methods generally take two callback methods as arguments. All queries are sent asynchronously to the server, thus requiring callback methods to run when the process has finished. The first one is executed when the operation is successful and the second one fires in case something fails.

Reading list items

The csp-query-provider includes the possibility to select data based on conditions. In Sharepoint queries this is facilitated by CAML. The csp-CAML-generator allows for a lightweight sql-esque syntax which gets converted to CAML. The CAML generator supports the following operands:

"<>" "!="	not equal	"IS NULL"	field is null
"="	equal	"IS NOT NULL”	field is not null
"<"	less than	"CONTAINS"	field contains value
">"	greater than	"BEGINS WITH"	field value begins with
"<="	less or equal than	"IN"	field equals value in list
">="	greater or equal than	"and" "or"	logical operators

```
var op = new csp.Operation({
    select: "Title, salery",
    list: "test",
    where: "Title = 'Anthony'* or salery > 4000",
    debug: true
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Title);
    }
},function (message) { console.log(message); });
```

>> Anthony

>> Clara

* csp uses commas (,) for separation. You may put strings in quotation marks and you should do so if the string contains a comma as this would break your query.

The next query shows the usage of brackets:

```
var op = new csp.Operation({
    select: "Title, salery",
    list: "test",
    where: "Title = 'Anthony' or (salery > 1000 and salery < 3000)",
    debug: true
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Title);
    }
},function (message) { console.log(message); });
```

>> Anthony

>> Bert

To select items based on a single ID it is possible to solely pass an integer to the 'where' property:

```
var op = new csp.Operation({
    select: "Title, salery",
    list: "test",
    where: 3,
    debug: true
});
```

The keyword 'In' can be used to provide a list of conditions to meet within a single column:

```
var op = new csp.Operation({
    select: "Title, salery, ID",
    list: "test",
    where: "ID in 1,2,3",
    debug: true
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Title + " " + output[item].salery + " " +
output[item].ID);
    }
}, function (message) { console.log(message); });
```

>> Anthony 9999

>> Clara 5000

>> Bert 2000

The 'readItems' method returns the data as an array of JavaScript objects. This data can be consumed by using its variable as a parameter in the first call back (success) method. The above query therefore returns:

```
output = [
    {
        Title: "Anthony",
        salary: 9999
    }, {
        Title: "Bert",
        salary: 2000
    }
];
```

The select keyword supports column aliasing and inline expressions. Aliases can be set by adding ‘as’ followed by the desired name. The columns in the result set are addressed by the alias whereas in the query we use the original names. This is especially helpful if you want to aggregate data from different sources into one view.

Inline expressions can be used to add data to a result set.

```
var op = new csp.Operation({
    select: "Title as Name, salery as Wage, (5 as Level)",
    list: "test",
    where: "Title = 'Anthony' or (salery > 1000 and salery < 3000)",
    debug: true
});

op.readItems(function (output) {
    for (var item in output) {
       console.log(output[item].Name + " " + output[item].Wage + " " +
                   output[item].Level);
    }
},function (message) { console.log(message); });
```

>> Anthony

>> Bert

This query returns:

```
output = [
    {
        Name: "Anthony",
        Wage: 9999,
        Level: 5
    }, {
        Name: "Bert",
        Wage: 2000,
        Level: 5
    }
];
```

The readItems operation furthermore supports ‘orderBy’ and ‘take’ keywords. The sort order in orderBy is set by following the desired columns name with ‘descending’ or ‘ascending’ (default). You may also use abbreviations ‘desc’ and ‘asc’.

```
var op = new csp.Operation({
    select: "Title as Name, salery as Wage",
    list: "test",
    debug: true,
    orderBy: "salery descending"
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Name + " " + output[item].Wage);
    }
},function (message) { console.log(message); });
```

>> Anthony 9999

>> Clara 5000

>> Bert 2000

```
var op = new csp.Operation({
    select: "Title as Name, salery as Wage",
    list: "test",
    debug: true,
    orderBy: "salery",
    take: 1
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Name + " " + output[item].Wage);
    }
},function (message) { console.log(message); });
```

>> Bert 2000

Joining lists

In order to join two lists a lookup has to be configured on the list that is intended to consume data from another list. In the following example the consuming list should be called ‘consumer’ and the providing list ‘provider’. In the list ‘consumer’ we added a lookup column called ‘lookup’.

When using joins, property ‘list’ must be provided with a join declaration. Unlike SQL the correct order of the joined lists matters. A consuming list has to be joined to a providing list with a lookup field configured on the consuming list.

```
var op = new csp.Operation({
    select: "provider.Title as Title_A, Level, Title as Title_B",
    list: "consumer join provider with lookup",
    debug: true
});


op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Title);
    }
},function (message) { console.log(message); });
```

When using debug, csp will display a warning that it cannot thoroughly check for existing fields in non-primary lists. Also when using debug, csp will warn you if you select columns name which exist in more than one table without an alias. Doing so makes the two columns undistinguishable in the output.

It is possible to chain joins. In the following example the list ‘CamlJoin_B’ consumes data from ‘CamlJoin_A’ (via ‘lookMe’) which itself consumes data from ‘C’ (via ‘lookup2’). Notice that the

names of consumed tables within a join can be chosen freely. Sharepoint infers listnames from lookup columns. Names are case sensible and have to consistent throughout the query though.

```
var op = new csp.Operation({
    site: "https://mySharepoint.corperation.com/project",
    select: "CamlJoin_A.Title as Person, Level, Title as State, C.Title as Address",
    list: "CamlJoin_B join CamlJoin_A with lookMe, CamlJoin_A join C with lookup2",
    debug: true
});

op.readItems(function (output) {
    for (var item in output) {
        console.log(output[item].Person + " " + output[item].Level + " " +
                    output[item].State + " " + output[item].Address);
    }
},function (message) { console.log(message); });
```

BDC-Support

When using csp on external lists, you can provide the filter keyword, telling Sharepoint to use filters set in bdc-models. The filter is set by adding a filter-object with the properties ‘name’, ‘value’ and optionally* ‘readOperationName’ to the query.

```
var op = new csp.Operation({
    select: "ID, Department, Parent, Active",
    list: "Departments",
    filter: {
        name: "Aktiv",
        value: 1,
        readOperationName: "Read List"
    },
    orderBy: "ID"
});
```

*When using bdc with visual studio the standard name is ‘ReadList’, while Sharepoint Desiginer calls it ‘Read List’. csp uses ‘ReadList’ when the property is not set.

Creating list items

To add a new person to our list ‘test’, we can use the following query, which basically adds ‘dora’ with a salary of 12.000. Notice that the values are passed through a ‘values’ object containing the fieldnames to be set as properties:

```
var op = new csp.Operation({
    list: "test",
    values: {
        Title: "dora",
        salery: 12000
    }
});

op.createItems(function () {
    console.log("done sucessfully");
}, function (msg) {
    console.log("something went wrong: " + msg);
});
```

If you want to insert more list items in one go you can pass an array of ‘values’ objects:

```
var op = new csp.Operation({
    list: "test",
    debug: true,
    values: [
              {
                  Title: "dora",
                  salery: 12000
              }, {
                  Title: "eugene",
                  salery: 500
              }
    ]
});

op.createItems(function () {
    console.log("done sucessfully");
}, function (msg) {
    console.log("something went wrong: " + msg);
});
```

Setting list item permissions

crudeSP has built in support for setting list item permission on create, update and file upload. In order to add ‘dora’ from above with list item permissions we’d use the following query:

```
var op = new csp.Operation({
    list: "test",
    debug: true,
    values: {
        Title: "dora",
        salery: 12000
    },
    roles: {
        deleteExisting: true,
        group: "aGroup",
        type: "write"
    }
});

op.createItems(function () {
    console.log("done sucessfully");
}, function (msg) {
    console.log("something went wrong: " + msg);
});
```

A ‘roles’ object was added to the query containing the properties ‘deleteExisting’, ‘group’ and ‘type’. Csp always breaks role inheritance on Sharepoint list items in order to be able to add new members. If you add ‘deleteExtisting: true’ to your first role, csp will delete all previous roles after breaking the inheritance. Otherwise it well keep all existing roles and add the desired role.
Within the roles object you can either set the property ‘group’ if you want to permit a Sharepoint group or the property ‘user’ which adds rights for single Sharepoint users. Users can be set with or without domain. If users or groups do not exist, the entire operation is aborted. Property ‘type’ sets the permission level for the desired users. Currently these three permission levels are available:

- read
- write
- contribute

If you choose to permit more than one user or group, you can pass an array of ‘roles’ objects:

```
roles: [
    {
        deleteExisting: true,
        group: "testGruppe",
        type: "write"
    },{
        user: "userName",
        type: "read"
    }
]
```

Updating list items

To update Sharepoint listitems use the updateItems method of the instantiated operation object. To change the name of ‘clara’ to ‘clementine’ in the test list we use the following query:

```
var updateItem = new csp.Operation({
    list: "test",
    where: 3,
    debug: true,
    set: {
        Title: "clementine"
    }
});

updateItem.updateItems(function () {
    console.log("item updated");
}, function (message) {
    console.log(message);
});
```

As mentioned above, csp lets you set list item permissions on update. You can do this with and without setting columns. The following query sets write permissions to ‘aGroup’ and read permissions to ‘userName’ for the items with an id from one to three.

```
var updateItem = new csp.Operation({
    list: "test",
    where: "ID in 1,2,3",
    debug: true,
    roles: [{
        deleteExisting: true,
        group: "aGroup",
        type: "write"
    },
       {
           user: "userName",
           type: "read"
       }]
});
```

```
updateItem.updateItems(function () {
    console.log("item updated");
}, function (message) {
    console.log(message);
});
```

Deleting list items

The following query deletes all entries from our test list with an id greater than three:

```
var op = new csp.Operation({
    list: "test",
    where: "ID > 3",
    debug: true
});

op.deleteItems(function () {
    console.log("done successfully");
}, function (msg) {
    console.log("something went wrong " + msg);
});
```

Uploading files

Csp supports uploading files to Sharepoint document libraries. The following query loads a file up to 'testLib', sets read permissions on the item for 'aGroup' and sets the column 'aColumnToSet' to 'first draft'.

```
var upload = new csp.Operation({
        site: "https://mySharepoint.corperation.com/project",
        list: "testLib",
        file: file,
        roles: {
            deleteExisting: true,
            type: "read",
            group: "aGroup"
        },
        set: {
            aColumnToSet: "first draft“
        }
    });

    upload.uploadToLibrary(function (data) {
        console.log("done successful");
    }, function (msg) {
        console.log("didn't work out: " + msg);
    });
```

It is possible to optionally set the property filename if you want a different file name. By default csp will use the original filename.