American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)

ISSN (Print) 2313-4410, ISSN (Online) 2313-4402

© Global Society of Scientific Research and Researchers

http://asrjetsjournal.org/

SharePoint 2013 – CRUD on List Items Using REST Services & jQuery

Matias Borgeaud [a], Laura Berger [b]

[a] *Email:* mborgeaud@yahoo.com

[b] *Email:* berger.laura@gmail.com

Abstract

SharePoint has proven to be a good solution for large organizations to store and share information, but it's front end is slow to load and gives us almost no control on how the information is shown. Fortunately, SharePoint 2013 has greatly expanded the REST services available to developers. Also, these new REST Services use the ODATA query standards, which means that we can easily write and test our queries using a web browser, because we'll be executing standard GET requests. With all this, we have much more SharePoint functionality exposed via APIs and Web Services. In other words, the ability to implement our own business rules and logic, with just a few lines of jQuery code and still use SharePoint as a data repository.

Keywords: SharePoint; REST; jQuery

1. Introduction

We can perform basic create, read, update, and delete (CRUD) operations by using the Representational State Transfer (REST) interface [1] provided by SharePoint 2013. The REST interface exposes all of the SharePoint entities and operations that are available in the SharePoint client APIs. One advantage of using REST is that we don't have to add references to any SharePoint 2013 libraries or client assemblies. Instead, we make HTTP requests to the appropriate endpoints to retrieve or update SharePoint entities, such as lists and list items.

To access SharePoint resources using REST, we need to construct a RESTful HTTP request, using the Open Data Protocol (OData) standard [2], for the desired client object model API.

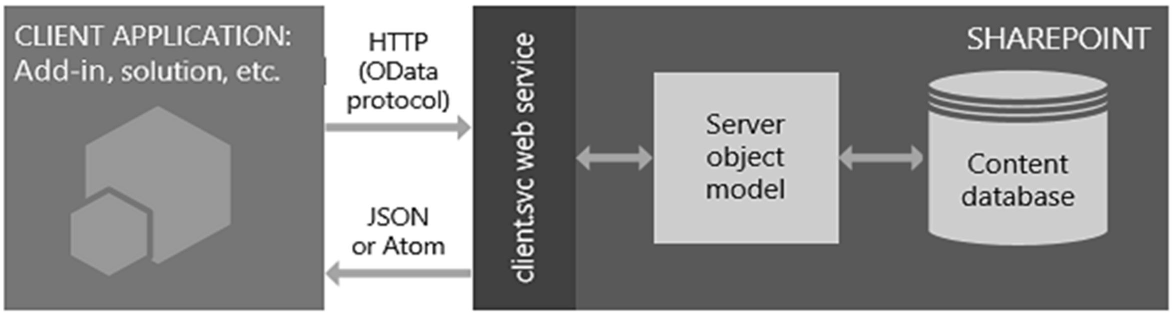

Figure 1: SharePoint REST service architecture [3]

1.1. Basic operations in REST

Below is a list of the basic commands used to get List Items from a SharePoint List through the SharePoint 2013 REST Services:

Table 1: List of basic Get commands and endpoints

COMMAND	**URL**
Get All Lists	http://server/site/_api/lists
Get All List Items	http://server/site/_api/lists/getbytitle(‘listname’)/items
Get a Single List Item	http://server/site/_api/lists/getbytitle(‘listname’)/items(id)
Get Back Certain Columns	http://server/site/_api/lists/getbytitle(‘listname’)/items?$select=Title,Id
Order the Results	http://server/site/_api/lists/getbytitle(‘listname’)/items?$orderby=Title

Given that REST Services use the ODATA query standards, we can test these commands using a web browser. To test a command, paste its URL into a web browser and the results will be shown as XML. We recommend using Chrome or Firefox, because IE10 will not display the XML by default.

1.2. Making REST Service calls with jQuery

With jQuery’s [4] AJAX (Asynchronous JavaScript and XML) calls, we can interact with a remote server using an HTTP requests. The following table lists the HTTP request types used for CRUD operations.

Table 2: List of HTTP Request Types [3]

OPERATION	**HTTP Request Type**	**COMMENTS**
Read a resource	**GET**	

Create or update a resource	**POST**	Use **POST** to create entities such as lists and sites. The SharePoint 2013 REST service supports sending **POST** commands that include object definitions to endpoints that represent collections. For **POST** operations, any properties that are not required are set to their default values. If we attempt to set a read-only property as part of a **POST** operation, the service returns an exception.
Update or insert a resource	**PUT**	Use **PUT** and **MERGE** operations to update existing SharePoint objects. Any service endpoint that represents an object property **set** operation supports both **PUT** requests and **MERGE** requests. · For **MERGE** requests, setting properties is optional; any properties that we do not explicitly set retain their current property. · For **PUT** requests, if we do not specify all required properties in object updates, the REST service returns an exception. In addition, any optional properties we do not explicitly set are set to their default properties.
Delete a resource	**DELETE**	Use the HTTP **DELETE** command against the specific endpoint URL to delete the SharePoint object represented by that endpoint. In the case of recyclable objects, such as lists, files, and list items, this results in a **Recycle** operation.

By default, the data is returned as XML in AtomPub format, but we can retrieve it in JSON (JavaScript Open Notation) format [5] by modifying the accept header in the HTTP request.

The following table shows properties that are commonly used in HTTP requests for the SharePoint 2013 REST services and when to use them.

Table 2: List of properties used in REST requests

PROPERTIES	REQUIRED IN	DESCRIPTION
url	All requests	The URL of the REST resource endpoint. Example: `http://<siteurl>/_api/web/lists`
method (or **type**)	All requests	The HTTP request method: **GET** for read operations and **POST** for write operations. **POST** requests can perform update or delete operations by specifying a **DELETE**, **MERGE**, or **PUT** verb in the X-HTTP-Method header.

body (or **data**)	**POST** requests that send data in the request body	The body of the POST request. Sends data (such as complex types) that can't be sent in the endpoint URI. Used with the **content-length** header.
Authentication *header*	Remote add-ins that use OAuth to authenticate users. Does not apply when using JavaScript or the cross-domain library.	Sends the OAuth access token (obtained from a Microsoft Access Control Service (ACS) secure token server) that's used to authenticate the user for the request. Example: `"Authorization": "Bearer " + accessToken`, where `accessToken` represents the variable that stores the token. Tokens must be retrieved by using server-side code.
X-RequestDigest *header*	**POST** requests (except SP.RequestExecutor requests)	Remote add-ins that use OAuth can get the form digest value from the `http://<siteurl>/_api/contextinfo` endpoint. SharePoint-hosted add-ins can get the value from the **#__REQUESTDIGEST** page control if it's available on the SharePoint page.
accept *header*	Requests that return SharePoint metadata	Specifies the format for response data from the server. The default format is `application/atom+xml`. Example: `"accept":"application/json; odata=verbose"`
content-type *header*	**POST** requests that send data in the request body	Specifies the format of the data that the client is sending to the server. The default format is `application/atom+xml`. Example: `"content-type":"application/json;odata=verbose"`
content-length *header*	**POST** requests that send data in the request body (except SP.RequestExecutor requests)	Specifies the length of the content. Example: `"content-length":requestBody.length`
IF-MATCH *header*	**POST** requests for **DELETE**, **MERGE**, or **PUT** operations, primarily for changing lists and libraries.	Provides a way to verify that the object being changed has not been changed since it was last retrieved. Or, lets us specify to overwrite any changes, as shown in the following example: `"IF-MATCH":"*"`
X-HTTP-Method *header*	**POST** requests for **DELETE**, **MERGE**, or **PUT** operations	Used to specify that the request performs an update or delete operation. Example: `"X-HTTP-Method":"PUT"`

The following working code shows how to retrieve a List Item (row) from a SharePoint list by its id. The `getListItem` function is a wrapper function that performs an Ajax call to the `getbytitle/items` REST service at `url`, for a `list` and returns the List Item #`id`. `Success` and `fail` are callback functions that will run after its execution.

2. Retrieving List Items

Code Sample 1: Get a single List Item by id

```
function getListItem(url, list, id, success, fail) {
  $.ajax({
    url     : url+"/_api/web/lists/getbytitle('"+list+"')/items("+id+")",
    type    : "GET",
    headers : { "Accept": "application/json; odata=verbose" },
    success : function (data) { success(data.d.results); },
    error   : function (data) { fail(data); }
  });
}

/* Custom success callback function */
function _success(rows) {
  $.each(rows, function () {
    console.log(this.ID + ":" + this.Title);
  });
}
/* Custom error callback function */
function _fail(error) {
  alert(JSON.stringify(error));
}
/* eg. Get employee whose id is 1 */
getListItem('http://site.org','Employee',1,_success,_fail);
```

2.1. Filtering results

SharePoint 2013 REST interface also supports sorting, filtering and limiting results. The following table shows some of the OData query options to control what data is returned. These options are passed as parameters, prepended to the Get command.

Table 2: List of ODATA query options

OPTION	PURPOSE
$select	Specifies which fields are included in the response, separated by colons (,).
$filter	Specifies which members of a collection, such as the items in a list, are returned.
$expand	Specifies which projected fields from a joined list are returned.
$top	Returns only the first n items of a collection or list ($top=n).
$skip	Skips the first n items of a collection or list and returns the rest ($skip=n).
$orderby	Specifies the field that is used to sort the data before it is returned.

The following working code example shows how to limit the results to a specific set of columns and order the results by a given field.

Code Sample 2: Get all List Items filtered by a query

```
function getListItems(url, list, query, success, fail) {
  $.ajax({
    url: url+"/_api/web/lists/getbytitle('"+list+"')/items?" + query,
    type: "GET",
    headers: { "Accept": "application/json; odata=verbose" },
    success: function (data) { success(data.d.results); },
    error: function (data) { fail(data); }
  });
}

/* eg. Get the Name, Age and City of all employees ordered by Age */
getListItems('http://site.org','Employee','$select=Name,Age,City&$orderby=A
ge desc',_success,_fail);
```

The `$filter` parameter needs special attention. For example, `$filter=City eq 'London'` will return only those employees who live in London. There are several comparison operators and functions that can be used:

Table 3: List of comparison operators and functions for the `$filter` option

NUMERIC comparison	**STRING comparison**	**DATE-TIME functions**
lt (lower than)	startswith	day()
le (lower or equal to)	substringof	month()
gt (greater than)	eq	year()
ge (greater or equal to)	ne	hour()
eq (equal to)		minute()
ne (not equal to)		second()

We can combine multiple filters to produce complex queries. For example: (parenthesis is optional, we use them because of readability)

```
$filter=((([FieldName1] eq 'value1') and ([FieldName2] eq 'value2')) or
([FieldName3] eq 'value3'))
```

2.2. Dealing with special characters

SharePoint doesn’t like special characters. But a simple solution to avoid this issue is, before sending the query text, convert it to Unicode using 'encodeURIComponent' method. This will encode the special characters which may create problems. The special character single quote (‘) needs extra attention though. So, whenever our filter text contains a single quote replace it with two single quotes. This is frequently called escaping.

3. Creating a List Item

Creating list items gets a little bit tricky, because we'll need a few key pieces of information:

- The List Item type
- REQUESTDIGEST value to prevent replay attacks (Form Digest)
- An object containing the List Item values

3.1. The List Item type

The `getListItemType` is just a string manipulation function, used to build the item type which is based on the title of the list, but it's case sensitive so we will also ensure to capitalize it.

```
function getListItemType(list) {
  return "SP.Data."+list[0].toUpperCase()+list.substring(1)+"ListItem";
}
```

3.2. The REQUESTDIGEST header

Even though it's pretty similar to fetching List Items, adding, modifying or removing them requires a POST (instead of a GET) request, plus an "authorization". This authorization is then sent as the "X-RequestDigest" header, in the headers section of the ajax call. This means that before posting to the REST service, we need to acquire the Form Digest. Without it, we'll receive nothing but a "The security validation for this page is invalid and might be corrupted. Please use your web browser's Back button to try your operation again." error message.

Sometimes it is enough to use the jQuery line `$("#__REQUESTDIGEST").val()` to get the Form Digest value, but if we don't have a master page defined or we use our own, the DOM element will not be present in our page. So, another way to get it [6] is by calling the ContextInfo API, as shown here:

```
function getFormDigest() {
  return $.ajax({
    url: "_api/contextinfo",
    type: "POST",
    headers: {
      "Accept": "application/json;odata=verbose",
      "contentType": "text/xml"
    }
  });
} // data.d.GetContextWebInformation.FormDigestValue
```

There's one caveat: we must get a new Form Digest each time we execute a POST operation (i.e. update, modify or delete a List Item) because they expire after some time. Given that both calls are asynchronous, the best way to ensure that the Form Digest was acquired before calling the POST operation is by chaining them, i.e. using jQuery.when() as in `$.when(getFormDigest()).done(function(f){})`. Inside the second function block, `f.d.GetContextWebInformation.FormDigestValue` will have the Form Digest value.

3.3. The object containing the List Item values

This object will serve as a container for the List Item that we're creating, updating or deleting. It is basically a serialized JSON string containing field-value pairs, for example:

```
var employee = {name: "John", lastname: "Doe", age: "33"}
```

3.4. The addListItem function

The following working code shows how to create a new List Item. The parameters received by the `createListItem` function are very similar to the ones used in the `getListItem` function, with just one exception: instead of sending the List Item Id, we are sending a List Item.

Code Sample 3: Creating a List Item

```
function createListItem(url, list, item, success, fail) {
  var _item = $.extend({
    "__metadata": { "type": getListItemType(list) }
  }, item);

  $.ajax({
    url: url + "/_api/web/lists/getbytitle('" + list + "')/items",
    type: "POST",
    contentType: "application/json;odata=verbose",
    data: JSON.stringify(_item),
    headers: {
      "Accept": "application/json;odata=verbose",
      "X-RequestDigest": $("#__REQUESTDIGEST").val()
    },
    success : function (data) { success(data.d.results); },
    error   : function (data) { fail(data); }
  });
}

/* eg. Add John Doe (33) */
var employee = {name: "John", lastname: "Doe", age: "33"};
createListItem('http://site.org','Employee',employee,_success,_fail);
```

4. Updating a List Item

Updating an existing List Item is not very different than adding one, except for two extra headers: `IF-MATCH` and `X-HTTP-METHOD`. `IF-MATCH` is used for managing concurrency, where "*" means "override other pending changes forcefully"; `X-HTTP-METHOD` is used to tell the REST Service the operation to be performed, i.e.: `MERGE` (to edit a list item) or `DELETE` (to delete a list item). Another important difference is that we need to explicitly pass the List Item Id (the primary key) to identify the List Item to be updated (or deleted).

Also, note that when updating list items, any property that we do not explicitly set, retain their current value. This means that if we want to update one single filed, we don't care about the others.

Code Sample 4: Updating a List Item

```
function updateListItem(url, list, item, success, fail) {
  $.ajax({
    var __item = $.extend({
      "__metadata": { "type": getListItemType(list) }
    }, item);

    url: url+"/_api/web/lists/getbytitle('"+list+"')/items("+item.ID+")",
    data: JSON.stringify(__item),
    contentType: "application/json;odata=verbose",
    type: "POST",
    headers: {
      "Accept": "application/json;odata=verbose",
      "X-RequestDigest": $("#__REQUESTDIGEST").val(),
      "IF-MATCH": "*",
      "X-HTTP-Method": "MERGE"
    },
    success : function (data) { success(data); },
    error   : function (data) { fail(data); }
  });
}

/* eg. Update Item ID:1 (34) */
var employee = {ID: 1, age: "34"};
updateListItem('http://site.org','Employee',employee,_success,_fail);
```

5. Deleting a List Item

The following code deletes a List Item. Note that it's very similar to the `updateListItem` function, and even more simple, given that we just need to pass the Id of the List Item to delete. We also added some function chaining to get the Form Digest, as an example.

Code Sample 5: Deleting a List Item

```
function deleteListItem(url, list, id, success, fail) {
  $.when(this.getFormDigest()).done(function(f) {
    $.ajax({
      url: url+"/_api/web/lists/getbytitle('"+list+"')/items("+id+")",
      type: "POST",
      headers: {
        "Accept": "application/json;odata=verbose",
        "X-RequestDigest": fd.d.GetContextWebInformation.FormDigestValue,
        "IF-MATCH": "*",
        "X-HTTP-Method": "DELETE"
      },
      success : function (data) { success(data); },
      error   : function (data) { fail(data); }
    });
  });
}
/* eg. Delete Item ID:1 */
deleteListItem('http://site.org','Employee',1,_success,_fail);
```

Conclusions

With the amount of new client-side application that are created every day, proven to be robust and secure, people that were apprehensive about using JavaScript or jQuery for full applications have come to accept them. Call it resignation, call it evolution, SharePoint is moving that way at a rapid pace.

By using the code in this work [7] and with the aid of great resources like [6], getting started creating your own custom SharePoint solution is just a matter of time. If you already moved to SharePoint 2016 or SharePoint Online, you'll notice that it works the same way. And if you're still on SharePoint 2010, it also has REST Services [8], JavaScript Client Side Object Model [9] and SOAP Web Services [10] to work with on the client side.

References

[1] "Representational state transfer," Wikipedia, 1 December 2016. [Online]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer. [Accessed 10 December 2016].

[2] "OData - The Best Way to REST," OData.org, Microsoft, 2015. [Online]. Available: http://www.odata.org/. [Accessed 10 12 2016].

[3] "Get to know the SharePoint 2013 REST service," 30 October 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/office/fp142380.aspx.

[4] "jQuery," jQuery.com, 2016. [Online]. Available: https://jquery.com/. [Accessed 10 December 2016].

[5] "JSON Format (OData Version 2.0)," OData.org, 2016. [Online]. Available: http://www.odata.org/documentation/odata-version-2-0/JSON-format/. [Accessed 10 December 2016].

[6] B. Atkinson, "Chapter 9. Working with the REST API," in *Custom SharePoint Solutions with HTML and JavaScript: For SharePoint 2016 and SharePoint Online*, Apress, 2015, pp. 191-192.

[7] M. Borgeaud, "mborgeaud@github," 2016. [Online]. Available: https://github.com/mborgeaud/SP/.

[8] "SharePoint Foundation REST Interface," Microsoft, 9 March 2015. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff521587(v=office.14).aspx. [Accessed 11 December 2016].

[9] "Common Programming Tasks in the JavaScript Object Model," Microsoft, 20 June 2011. [Online]. Available: https://msdn.microsoft.com/en-us/library/hh185015(v=office.14).aspx. [Accessed 11 December 2016].

[10] "SharePoint 2010 Web Services," Microsoft, 19 April 2011. [Online]. Available: https://msdn.microsoft.com/en-us/library/ee705814(v=office.14).aspx. [Accessed 11 December 2016].