@pnp/sp/items

GET

Getting items from a list is one of the basic actions that most applications require. This is made easy through the library and the following examples demonstrate these actions.

Basic Get

import { sp } from "@pnp/sp";

// get all the items from a list

sp.web.lists.getByTitle("My List").items.get().then((items: any[]) => {

console.log(items);

});

// get a specific item by id

sp.web.lists.getByTitle("My List").items.getById(1).get().then((item: any) => {

console.log(item);

});

// use odata operators for more efficient queries

sp.web.lists.getByTitle("My List").items.select("Title", "Description").top(5).orderBy("Modified", true).get().then((items: any[]) => {

console.log(items);

});

Get Paged Items

Working with paging can be a challenge as it is based on skip tokens and item ids, something that is hard to guess at runtime. To simplify things you can use the getPaged method on the Items class to assist. Note that there isn't a way to move backwards in the collection, this is by design. The pattern you should use to support backwards navigation in the results is to cache the results into a local array and use the standard array operators to get previous pages. Alternatively you can append the results to the UI, but this can have performance impact for large result sets.

import { sp } from "@pnp/sp";

// basic case to get paged items form a list

let items = await sp.web.lists.getByTitle("BigList").items.getPaged();

// you can also provide a type for the returned values instead of any

let items = await sp.web.lists.getByTitle("BigList").items.getPaged<{Title: string}[]>();

// the query also works with select to choose certain fields and top to set the page size

let items = await sp.web.lists.getByTitle("BigList").items.select("Title", "Description").top(50).getPaged<{Title: string}[]>();

// the results object will have two properties and one method:

// the results property will be an array of the items returned

if (items.results.length > 0) {

console.log("We got results!");

for (let i = 0; i < items.results.length; i++) {

// type checking works here if we specify the return type

console.log(items.results[i].Title);

}

}

// the hasNext property is used with the getNext method to handle paging

// hasNext will be true so long as there are additional results

if (items.hasNext) {

// this will carry over the type specified in the original query for the results array

items = await items.getNext();

console.log(items.results.length);

}

getListItemChangesSinceToken

The GetListItemChangesSinceToken method allows clients to track changes on a list. Changes, including deleted items, are returned along with a token that represents the moment in time when those changes were requested. By including this token when you call GetListItemChangesSinceToken, the server looks for only those changes that have occurred since the token was generated. Sending a GetListItemChangesSinceToken request without including a token returns the list schema, the full list contents and a token.

import { sp } from "@pnp/sp";

// Using RowLimit. Enables paging

let changes = await sp.web.lists.getByTitle("BigList").getListItemChangesSinceToken({RowLimit: '5'});

// Use QueryOptions to make a XML-style query.

// Because it's XML we need to escape special characters

// Instead of & we use &amp; in the query

let changes = await sp.web.lists.getByTitle("BigList").getListItemChangesSinceToken({QueryOptions: '<Paging ListItemCollectionPositionNext="Paged=TRUE&amp;p\_ID=5" />'});

// Get everything. Using null with ChangeToken gets everything

let changes = await sp.web.lists.getByTitle("BigList").getListItemChangesSinceToken({ChangeToken: null});

Get All Items

*Added in 1.0.2*

Using the items collection's getAll method you can get all of the items in a list regardless of the size of the list. Sample usage is shown below. Only the odata operations top, select, and filter are supported. usingCaching and inBatch are ignored - you will need to handle caching the results on your own. This method will write a warning to the Logger and should not frequently be used. Instead the standard paging operations should be used.

import { sp } from "@pnp/sp";

// basic usage

sp.web.lists.getByTitle("BigList").items.getAll().then((allItems: any[]) => {

// how many did we get

console.log(allItems.length);

});

// set page size

sp.web.lists.getByTitle("BigList").items.getAll(4000).then((allItems: any[]) => {

// how many did we get

console.log(allItems.length);

});

// use select and top. top will set page size and override the any value passed to getAll

sp.web.lists.getByTitle("BigList").items.select("Title").top(4000).getAll().then((allItems: any[]) => {

// how many did we get

console.log(allItems.length);

});

// we can also use filter as a supported odata operation, but this will likely fail on large lists

sp.web.lists.getByTitle("BigList").items.select("Title").filter("Title eq 'Test'").getAll().then((allItems: any[]) => {

// how many did we get

console.log(allItems.length);

});

Retrieving Lookup Fields

When working with lookup fields you need to use the expand operator along with select to get the related fields from the lookup column. This works for both the items collection and item instances.

import { sp } from "@pnp/sp";

sp.web.lists.getByTitle("LookupList").items.select("Title", "Lookup/Title", "Lookup/ID").expand("Lookup").get().then((items: any[]) => {

console.log(items);

});

sp.web.lists.getByTitle("LookupList").items.getById(1).select("Title", "Lookup/Title", "Lookup/ID").expand("Lookup").get().then((item: any) => {

console.log(item);

});

Retrieving PublishingPageImage

The PublishingPageImage and some other publishing-related fields aren't stored in normal fields, rather in the MetaInfo field. To get these values you need to use the technique shown below, and originally outlined in [this thread](https://github.com/SharePoint/PnP-JS-Core/issues/178). Note that a lot of information can be stored in this field so will pull back potentially a significant amount of data, so limit the rows as possible to aid performance.

import { Web } from "@pnp/sp";

const w = new Web("https://{publishing site url}");

w.lists.getByTitle("Pages").items

.select("Title", "FileRef", "FieldValuesAsText/MetaInfo")

.expand("FieldValuesAsText")

.get().then(r => {

// look through the returned items.

for (var i = 0; i < r.length; i++) {

// the title field value

console.log(r[i].Title);

// find the value in the MetaInfo string using regex

const matches = /PublishingPageImage:SW\|(.\*?)\r\n/ig.exec(r[i].FieldValuesAsText.MetaInfo);

if (matches !== null && matches.length > 1) {

// this wil be the value of the PublishingPageImage field

console.log(matches[1]);

}

}

}).catch(e => { console.error(e); });

Add Items

There are several ways to add items to a list. The simplest just uses the *add* method of the items collection passing in the properties as a plain object.

import { sp, ItemAddResult } from "@pnp/sp";

// add an item to the list

sp.web.lists.getByTitle("My List").items.add({

Title: "Title",

Description: "Description"

}).then((iar: ItemAddResult) => {

console.log(iar);

});

Content Type

You can also set the content type id when you create an item as shown in the example below:

import { sp } from "@pnp/sp";

sp.web.lists.getById("4D5A36EA-6E84-4160-8458-65C436DB765C").items.add({

Title: "Test 1",

ContentTypeId: "0x01030058FD86C279252341AB303852303E4DAF"

});

User Fields

There are two types of user fields, those that allow a single value and those that allow multiple. For both types, you first need to determine the Id field name, which you can do by doing a GET REST request on an existing item. Typically the value will be the user field internal name with "Id" appended. So in our example, we have two fields User1 and User2 so the Id fields are User1Id and User2Id.

Next, you need to remember there are two types of user fields, those that take a single value and those that allow multiple - these are updated in different ways. For single value user fields you supply just the user's id. For multiple value fields, you need to supply an object with a "results" property and an array. Examples for both are shown below.

import { sp } from "@pnp/sp";

import { getGUID } from "@pnp/common";

sp.web.lists.getByTitle("PeopleFields").items.add({

Title: getGUID(),

User1Id: 9, // allows a single user

User2Id: {

results: [ 16, 45 ] // allows multiple users

}

}).then(i => {

console.log(i);

});

If you want to update or add user field values when using **validateUpdateListItem** you need to use the form shown below. You can specify multiple values in the array.

import { sp } from "@pnp/sp";

const result = await sp.web.lists.getByTitle("UserFieldList").items.getById(1).validateUpdateListItem([{

FieldName: "UserField",

FieldValue: JSON.stringify([{ "Key": "i:0#.f|membership|person@tenant.com" }]),

},

{

FieldName: "Title",

FieldValue: "Test - Updated",

}]);

Lookup Fields

What is said for User Fields is, in general, relevant to Lookup Fields: - Lookup Field types: - Single-valued lookup - Multiple-valued lookup - Id suffix should be appended to the end of lookup's EntityPropertyName in payloads - Numeric Ids for lookups' items should be passed as values

import { sp } from "@pnp/sp";

import { getGUID } from "@pnp/common";

sp.web.lists.getByTitle("LookupFields").items.add({

Title: getGUID(),

LookupFieldId: 2, // allows a single lookup value

MuptiLookupFieldId: {

results: [ 1, 56 ] // allows multiple lookup value

}

}).then(console.log).catch(console.log);

Add Multiple Items

import { sp } from "@pnp/sp";

let list = sp.web.lists.getByTitle("rapidadd");

list.getListItemEntityTypeFullName().then(entityTypeFullName => {

let batch = sp.web.createBatch();

list.items.inBatch(batch).add({ Title: "Batch 6" }, entityTypeFullName).then(b => {

console.log(b);

});

list.items.inBatch(batch).add({ Title: "Batch 7" }, entityTypeFullName).then(b => {

console.log(b);

});

batch.execute().then(d => console.log("Done"));

});

Update

The update method is very similar to the add method in that it takes a plain object representing the fields to update. The property names are the internal names of the fields. If you aren't sure you can always do a get request for an item in the list and see the field names that come back - you would use these same names to update the item.

import { sp } from "@pnp/sp";

let list = sp.web.lists.getByTitle("MyList");

list.items.getById(1).update({

Title: "My New Title",

Description: "Here is a new description"

}).then(i => {

console.log(i);

});

Getting and updating a collection using filter

import { sp } from "@pnp/sp";

// you are getting back a collection here

sp.web.lists.getByTitle("MyList").items.top(1).filter("Title eq 'A Title'").get().then((items: any[]) => {

// see if we got something

if (items.length > 0) {

sp.web.lists.getByTitle("MyList").items.getById(items[0].Id).update({

Title: "Updated Title",

}).then(result => {

// here you will have updated the item

console.log(JSON.stringify(result));

});

}

});

Update Multiple Items

This approach avoids multiple calls for the same list's entity type name.

import { sp } from "@pnp/sp";

let list = sp.web.lists.getByTitle("rapidupdate");

list.getListItemEntityTypeFullName().then(entityTypeFullName => {

let batch = sp.web.createBatch();

// note requirement of "\*" eTag param - or use a specific eTag value as needed

list.items.getById(1).inBatch(batch).update({ Title: "Batch 6" }, "\*", entityTypeFullName).then(b => {

console.log(b);

});

list.items.getById(2).inBatch(batch).update({ Title: "Batch 7" }, "\*", entityTypeFullName).then(b => {

console.log(b);

});

batch.execute().then(d => console.log("Done"));

});

Recycle

Sending an item to the Recycle Bin is as simple as calling the .recycle method.

import { sp } from "@pnp/sp";

let list = sp.web.lists.getByTitle("MyList");

list.items.getById(1).recycle().then(\_ => {});

Delete

Delete is as simple as calling the .delete method. It optionally takes an eTag if you need to manage concurrency.

import { sp } from "@pnp/sp";

let list = sp.web.lists.getByTitle("MyList");

list.items.getById(1).delete().then(\_ => {});

Resolving field names

It's a very common mistake trying wrong field names in the requests. Field's EntityPropertyName value should be used.

The easiest way to get know EntityPropertyName is to use the following snippet:

import { sp } from "@pnp/sp";

sp.web.lists

.getByTitle('[Lists\_Title]')

.fields

.select('Title, EntityPropertyName')

.filter(`Hidden eq false and Title eq '[Field's\_Display\_Name]'`)

.get()

.then(response => {

console.log(response.map(field => {

return {

Title: field.Title,

EntityPropertyName: field.EntityPropertyName

};

}));

})

.catch(console.log);

Lookup fields' names should be ended with additional Id suffix. E.g. for Editor EntityPropertyName EditorId should be used.