IJoL International Journal of Librarianship

ISSN: 2474-3542 Journal homepage: https://journal.calaijol.org

Lessons Learned about FOLIO's APIs

Guy Dobson

Abstract:

APIs (Application Programming Interface) provide the ability to do what needs to be done. The fact that FOLIO includes API as one of its building blocks makes it that much more attractive. When my library's administration decided to switch from a legacy ILS (Integrated Library System) to a FOLIO LSP (Library Services Platform) the first thing that I looked at was the API. The lessons learned helped me to configure the system and massage the data from ILS output to FOLIO-friendly input. By building web applications and writing Perl scripts our staff is able to get the job done even when it is impossible to accomplish the task through the user interface (UI).

To cite this article:

Dobson, G. (2021). Lessons learned about FOLIO's APIs? *International Journal of Librarianship*, *6*(2), 30-39. https://doi.org/10.23974/ijol.2021.vol6.2.206

INTERNATIONAL JOURNAL OF LIBRARIANSHIP, 6(2), 30-39.
ISSN: 2474-3542

Lessons Learned about FOLIO's APIs

Guy Dobson, Drew University

ABSTRACT

APIs (Application Programming Interface) provide the ability to do what needs to be done. The fact that FOLIO includes API as one of its building blocks makes it that much more attractive. When my library's administration decided to switch from a legacy ILS (Integrated Library System) to a FOLIO LSP (Library Services Platform) the first thing that I looked at was the API. The lessons learned helped me to configure the system and massage the data from ILS output to FOLIO-friendly input. By building web applications and writing Perl scripts our staff is able to get the job done even when it is impossible to accomplish the task through the user interface (UI).

Keywords: API, FOLIO API, Web Apps

EXPLORING FOLIO'S APIS

I explored FOLIO's APIs by writing Perl scripts that use curl to send API calls to FOLIO. Every API call requires an Okapi[1] (which is a multitenant API Gateway to FOLIO) token so the first lesson learned was how to get one. By sending an API call to https://[path to my FOLIO environment]/authn/login with a $tenant, $username, and $password I was able to parse the Okapi token out of the results and use it to define a variable named $okapiToken like so:

```
$post = `curl -i -s -X POST -H 'Content-type: application/json'
-H 'X-Okapi-Tenant: $tenant' -d '{"username": "$username",
"password": "$password"}' $url/authn/login`;
@parts = split(/\n/,$post);
foreach $part (@parts) {
     if ($part =~ /^x-okapi-token:/) {
          $okapiToken = "X-Okapi-Token: " . substr($part,15);
     }
}
```

After submitting the following curl API call:

```
$json = `curl -s -X GET -H 'Content-type: application/json' -H
'$okapiToken' $url/coursereserves/terms?query=name=Fall 2021`;
```

I then decoded the $json variable using Perl's JSON module:

```
     $hash = decode_json $json;
```

[1] https://github.com/folio-org/okapi

```
for ( @{$hash->{terms}} ) {
     $termId = $_->{'id'};
}
```

and parsed the data I needed out of the JSON (JavaScript Object Notation) document.

Lesson learned: you don't have to get a new Okapi token every time you need to make a new API call. Get it once and save it to a file, perhaps okapiToken.txt, and you can use it over and over again which will help your Perl scripts run that much faster. You'll only need to get a new token if you have to give the relevant user new permissions or when you upgrade to a new version of FOLIO.

Using the above technique, I was able to explore the many database tables and begin to understand their relationships. I browsed through the API Documentation[2] website, sent GET calls to various endpoints, compared the resulting JSON documents to the above documentation and to what I was seeing in the UI, and took notes.

Lesson learned: contributor name types (labels associated with MARC (Machine-Readable Cataloging Record)'s 100, 110, and 111 tags) are nowhere to be found in the UI's Settings app. If they ever need to be added to, edited, or deleted that will have to be done via API.

With a test environment that EBSCO built for us which had been populated with reference (demo) data; the tables that needed to be populated before loading bibliographic or patron data were already set up. There was even a demo institution, Københavns Universitet, complete with demo patrons and demo books. Some of the reference data is likely to be the same across institutions as it is simply the labels that describe various fields in the MARC record. The same is true for those tables that contain RDA (Resource Description and Access) labels. Others, especially locations, will need to be customized by each individual library.

A closer look at the reference data revealed that the list of alternative title types was not as complete as we (a small group of librarians and staff) would like. It included "Uniform title" (aka 130, 240, or 730) and the options listed for the 2nd indicator of 246 but it did not include options that are relevant to other title fields (210, 222, 242, 243, 247, or 740). For this reason, we decided not to use the reference data and Perl scripts were built to load these labels into our test environment so that all I would have to do is change the scripts' URL (Uniform Resource Locator) and tenant to load the same labels into the production environment when EBSCO got around to setting that up for us.

[2] https://dev.folio.org/reference/api/

CONFIGURING THE DATABASE TABLES

Using the Library of Congress's MARC 21 Format for Bibliographic Data[3] website I was able to find content for several of FOLIO's tables, such as the above mentioned alternative title types and contributor name types. Others include:

- Contributor types
    - there are 278 of them – see MARC Code List for Relators[4]
- Formats
    - actually RDA Carrier Types[5]
- Identifiers
    - MARC's 01X-09X
- Mode of issuance
    - found in the MARC leader's position 7
- Instance note types
    - 5XX
    - we decided to include the MARC tag in the label e.g. Dissertation Note (502)
- Resource types
    - actually RDA Content Types[6]
    - MARC's 336 tag
    - found in the instanceTypes table via API
    - a required field for creating an instance
- URL relationship
    - 856's 2nd indicator

Lesson learned: a FOLIO instance is not a MARC record. Rather, it is a row in the instances table that you can create and/or edit by curling a JSON document that includes the title, author, etc., and various foreign keys: the UUIDs from other tables that describe the instance's content (e.g. alternative title types, contributor types, identifiers, etc.)

Other fields with labels that are not associated with MARC or RDA include…

- Instance status types
    - An instance does not require a status but once one is assigned it cannot be removed - it can only be changed to a different status. The reference data included the following prefab statuses (and codes):
        - cataloged (cat)
        - uncatalogued (uncat)
        - batch loaded (batch)
        - temporary (temp)
        - other (other)
        - not yet assigned (none)

[3] https://www.loc.gov/marc/bibliographic/
[4] https://www.loc.gov/marc/relators/relaterm.html
[5] https://www.loc.gov/standards/valuelist/rdacarrier.html
[6] https://www.loc.gov/standards/valuelist/rdacontent.html

    - ...but we decided to load these instead...
        - discarded (aka. Withdrawn)
        - original cataloging (notCopyCat)
        - more information needed (thereRquestions)
        - ready for circ (noProblem)
        - series error (oops)
        - waiting for invoice (stillWaiting)
- Locations
    - FOLIO's four-tiered location structure allows for a much more granular solution than we were able to accomplish in the ILS. We took full advantage of this.
- Material types
    - We also took advantage of the opportunity to consolidate our overgrown list of item types by eliminating the obsolete and making others consistent with RDA carrier types or ISBD material designations.
- Nature of content
    - These essentially work like tags. We decided to use them to help identify online titles that might not have holdings records or items and so no locations or material types:
        - GovDocs
        - Reference
        - Theses & Dissertations
        - Video
- Service points
    - Aka. the Circulation desk. This is associated with both users and locations (where one is required)

Fields that are relevant to Users include Address types and Patron groups. These were the most straight forward to set up.

This list is not meant to be exhaustive or complete, just illustrative. In fact, we have not implemented all of FOLIO's apps yet and so there are many tables in our environment that are still empty. Once the above tables were loaded with the labels that we decided to use we were able to begin loading data from our legacy ILS but our data wasn't ready for FOLIO.

MASSAGING, AND MIGRATING, THE DATA

There are a couple of important differences between FOLIO and our legacy ILS regarding which fields belong in which tables.

1. Locations are associated with Holdings Records in FOLIO whereas they were associated with items in our old system.
2. Volume numbers, and other details that are a part of the call number in the ILS, belong in FOLIO's item record, not with the call number in the holdings record.

Also, the item information as it came out of the ILS in 999 fields was not as complete as we wanted it to be, e.g. it was missing staff notes, etc. We would have to take a number of batches of data out of the ILS in addition to the MARCs w/999s and process all of this information to create FOLIO-friendly input. The first step was to put the ILS output into these MySQL tables:

```
mysql> describe marcRecords;
+---------------+-------------+
| Field         | Type        |
+---------------+-------------+
| marc          | mediumtext  |
| instanceId    | varchar(36) |
| shadowFlag    | varchar(1)  |
| catalogedDate | varchar(8)  |
| marcHoldings  | varchar(1)  |
| tcn           | int(11)     |
+---------------+-------------+


mysql> describe itemData;
+-------------------+-------------+
| Field             | Type        |
+-------------------+-------------+
| analyticPosition  | int(11)     |
| callNumShadowFlag | varchar(1)  |
| circNote          | mediumtext  |
| publicNote        | mediumtext  |
| staffNote         | mediumtext  |
| itemCat1          | varchar(10) |
| itemCat2          | varchar(10) |
| itemShadowFlag    | varchar(1)  |
| tcn               | int(11)     |
| barcode           | varchar(16) |
+-------------------+-------------+

mysql> describe marcHoldingsRecords;
+-------+-------------+
| Field | Type        |
+-------+-------------+
| tcn   | int(11)     |
| m852c | varchar(10) |
| m863  | mediumtext  |
| m866  | mediumtext  |
| m867  | mediumtext  |
| m868  | mediumtext  |
| id    | int(11)     |
+-------+-------------+
```

My Perl script could then:

1. Select one MARC record for which the instanceId field was NULL
2. Parse the MARC data into a FOLIO-friendly JSON document
3. curl (POST) the JSON document to /instance-storage/instances

    a. Parse the instanceId out of the resulting JSON document (Lesson learned: /inventory/instances will also create an instance but it does not return a helpful JSON document from which to parse the, very necessary, instanceId)
    b. Add the instanceId to the relevant row in the marcRecords MySQL table
4. Build an itemInfo array with the data from both the 999 field(s) and the itemData table
5. If there are MARC holdings records (marcRecords.marcHolidngs = "T") build a marcHoldings array
6. For each unique call number parse the data in the itemInfo array including…
    a. the call number,
    b. the location (formerly associated with the item),
    c. and any MARC holdings statements (formerly in separate records)

    …into a holdingsRecord JSON document that includes the instanceId
7. curl (POST) the JSON document to /holdings-storage/holdings
    a. parse the holdingsRecordId out of the resulting JSON document
8. For each item with that call number parse the rest of the data in the itemInfo array including…
    a. the barcode,
    b. the material type,
    c. the volume number (formerly a part of the call number)

        …into an item JSON document that includes the holdingsRecordId
9. curl (POST) the JSON document to /item-storage/items
10. Repeat all of the above until there are no NULL instanceIds in the marcRecords table

The UI's data import app wasn't used because I had trouble getting it to work. I had already written Perl scripts to create instances, holdings records, and items and so had a pretty clear vision of how to go about this via API.

A similar, and much less complicated, process was used to POST user data to /users after parsing the pipe delimited data from the University's Banner database. When I started doing this I was working with Fameflower which threw an error because I didn't include a userId in the JSON document. This was a big surprise since other tables provided a UUID when POSTed to. After learning how to craft a UUID and adding that functionality to my postUsers.pl script the test environment was upgraded. In Goldenrod my craft_uuid subroutine was no longer required but that was a very interesting lesson learned.

Current charges were migrated by stowing data from the ILS in this MySQL table:

```
mysql> describe chargeData;
+-------------+-------------+
| Field       | Type        |
+-------------+-------------+
| circRule    | varchar(10) |
| dateCharged | varchar(12) |
| dateDue     | varchar(12) |
| itemBarcode | varchar(14) |
| itemType    | varchar(10) |
| userBarcode | varchar(14) |
| userProfile | varchar(10) |
| loanId      | varchar(36) |
| id          | int(11)     |
+-------------+-------------+
```

Loans could then be created by POSTing to /circulation/check-out-by-barcode. For loans with rolling due dates it was necessary to parse the loanId out of the resulting JSON document and follow up with /circulation/loans/$loanId/change-due-date to modify the due date. The loanId could then be added to the MySQL table and the process repeated until none of the loanIds in the chargeData table were NULL.

BUILDING WEB APPS AND COMMAND LINE SOLUTIONS WITH PERL

One of the first requests from the folks at the Circulation desk was to be able to scan a barcode and see, at a glance, an item's author, title, publisher information, location, call numbers, status, and previous loan data including loan date, due date, return date, status, and the name and barcode number of the previous borrower(s). This information was available in the UI but not all on one screen. The solution was to build a simple HTML form that would pass a barcode and password to a Perl script that would then:

1. GET all of the item data including the itemId and holdingsRecordId

   `/item-storage/items?query=barcode="$barcode"`
2. GET all of the holdings data including the instanceId

   `/holdings-storage/holdings?query=id="$holdingsRecordId"`
3. GET all of the instance data

   `/instance-storage/instances?query=id="$instanceId"`
4. GET all of the loan data including any userIds

   `/circulation/loans?query=itemId="$itemId"`
5. GET all of the user data

   `/users?query=id="$userId"`

Depending on the reason for looking up an item the loan data may not be relevant and so it is put into a JavaScript variable so that it can be displayed on demand if necessary.

Lesson learned: note that in steps 2 and 3 above the API call includes a query and that the same information could be retrieved like so:

```
/holdings-storage/holdings/$holdingsRecordId
/instance-storage/instances/$instanceId
```

The JSON returned by these API calls does not include the name of the relevant table at the beginning ('`holdingsRecords`' and '`instances`') nor the total number of records returned at the end (which in this case would be '`"total records" : 1`'.) I find that these details are often helpful. It also makes the business of writing the code that parses the JSON more consistent and less confusing.

Considering the fact that we were not using FOLIO's Data import app we were going to need a web app to load new titles into our new LSP. Together with our cataloger I designed what we call *The Dashboard* which enables one to load MARC records either individually or en masse and provides follow up links that take you into either the UI or the OPAC so that you can check your work. The script behind this is basically the same as the one described above that was used to load MARCs from the ILS. The main difference is that this process does not attempt to create holdings records or items as that can be done by simply following the resulting link(s) into the UI. It also lets you either click on a link in the OPAC or enter an instance's HRID to work with titles that are already in the system. You can then download the MARC record and fuss with it in MarcEdit before overlaying it in MySQL and updating the instance in FOLIO.

Lesson learned: it is not necessary to stow MARC records in FOLIO. We already had a MySQL database where we keep the MARC records that support our OPAC and we decided that this would be where we would keep our MARC records of record.
Another important web app is *What's on Reserve*. The process of assembling the information on the page goes like so:

1. GET the termId for the current term
```
/coursereserves/terms?query=name=Fall 2021
```
2. GET the courseListingIds and Prof. names
```
/coursereserves/courselistings?query=termId=$termId
```
3. For each courseListingId GET the course number and name
```
/coursereserves/courses?query=courseListingId=$courseListin
gId
```
4. Push all of the above into an array
```
push(@courses,"$courseNumber|$profName|$courseName|\n")
```
5. For each courseListingId GET all of the titles that are on reserve
```
/coursereserves/reserves?query=courseListingId=$courseListi
ngId
```
6. Push each title's information into an array
```
push(@reserves,"$sortTitle|$title|$callNumber|$courseNumber
|$location|\n")
```
7. Sort each array and then parse through them to display each of the titles that are on reserve for each course like so:
   - LIBR 101 Introduction to Librarianship
     Prof. Dewey Decimal
     - Librarianship: the Art of Strategic Neglect
       Main Library Reserves 025 D519l

Not all solutions are best served by web apps. Some only need to happen once and/or are not interactive and so a simple Perl script, without the web page infrastructure, will get the job done. An example of this includes adding an item's cost as a note in the item record. After all of the data from the ILS was loaded into FOLIO we noticed that we had not included the cost of the item in the item record; which data was in the 999s. Then we noticed that the item record does not include a field for this data point. We created a new item note type named *Price*. Then I wrote a script that would sift through all of the 999s, GET each item's JSON document from /item-storage/items, use regular expressions to take the JSON document apart and then put it back together so that it would either add a new item note to the existing array or create a new item note array, and then PUT the new JSON document to /item-storage/items thus overlaying the old one.

Another example is extending the expiration dates for current users. This is something that we do each semester for students and once a year for faculty and staff. This process uses `/groups?query=group="*"` to get all usergroups and their groupIds and then uses `/users?query=patronGroup=="$groupId"` to get all of the desired users. Then, for each user, our Banner system is checked and if that user is current the expiration date is changed with a regular expression.

```
     $data =~ s/expirationDate":"\d\d\d\d-\d\d-
     \d\d/expirationDate":"2022-02-14/
```

Then `` `curl -s -X PUT -H 'Content-type: application/json' -H '$okapiToken' -d '$data' $url/users/$userId` `` gets the job done.

CONCLUSIONS AND ANTICIPATIONS

When we started working with FOLIO back in the Fameflower days, one of our first conversations with the folks at EBSCO was about documentation. To be more exact, it was about where we might find documentation regarding using the UI. The answer was that there wasn't any, yet. By using API to take a deep dive into the database tables and take their JSON output apart, I learned the lessons that I needed to configure the system and load our bibliographic and patron data. Now, while we are eagerly awaiting an upgrade to Kiwi, it is a very different landscape. The LSP includes more apps and the Project Wiki[7] includes a lot more information. We plan to begin using the Acquisitions and ERM apps very soon. Whether or not we need to use API to set up those apps, I will certainly begin with an inventory of the relevant database tables the way I did when I first started working with Fameflower. However it turns out, I'm sure that the lessons we've learned will serve us well as we expand our use of FOLIO and as FOLIO continues to grow.

[7] https://wiki.folio.org/

About the author

Guy Dobson (gdobson@drew.edu) is the director of Technical Services and Systems Librarian at Drew University in Madison, New Jersey. After working at the local public library while attending high school and college he earned his Master of Science from Columbia University's School of Library Service. In addition to public libraries he has worked in academic, medical, music, and school libraries. He has given several presentations at national conferences about building systems that add or enhance functionality for patrons and staff in library websites and catalogs.