ScienceBase Catalog Item REST Services

This documentation explains ScienceBase REST services and ways to interact with the ScienceBase API. The documentation focuses on Item Class and its related objects as they are implemented in the ItemController and ItemsController REST Service end points. Examples for using GET, POST, PUT, UPSERT, and DELETE operations are provided. Documentation is concentrated in 4 areas:

1. The Item core objects model - includes information fields used to describe a ScienceBase Item.
2. JSON item example - JSON view of a ScienceBase item built to completeness using simple example text/files/objects.
3. A step-by-step example of using the REST Service Firefox Client to perform a GET, POST, PUT, UPSERT, and DELETE operations.
4. More advanced CRUD operations using identifiers, multiple items, async, and downloading links.

HINT: Before you get started with REST Service for an Item, it is helpful to create a test record for reference in JSON format. Go to ScienceBase: https://www.sciencebase.gov/catalog/item/editForm/new to create a new record. Complete requested information in entry tabs. Popup hints demarked by "?" provide information about the sought metadata and preferred format. These helpful popups describe the fields in detail and match up to the core objects below. Once an Item is created in ScienceBase, you can select the "view JSON" link (bottom left in item summary page, or append "?format=json" to item URL in the browser web address bar) to view the JSON representation of the ScienceBase Item.

Within the child pages of this page is a more syntactical and robust CRUD operations page along with a bevy of child pages dealing with the more advanced REST operations within ScienceBase.

Item Core Objects

Item contains the following core objects:

- id: ObjectId type that is a system generated unique identifier of an item
- title: String type, Item Title
- subTitle: String type, Item Subtitle
- body: String type, Item Description
- purpose: String type, Item Purpose
- rights: String type, Item Rights
- provenance: ItemProvenance type
    - annotation: String type, 'Item origin'
    - html: System derived
    - dataSource: System derived
    - dateCreated: System derived
    - lastUpdated: System derived
    - lastUpdatedBy: System derived
    - createdBy: System derived
- materialRequestInstructions: String type, Item Material Request Instructions
- alternateTitles: String type, List of Item Alternate Titles
- contacts: List<Contact>, Item Contact, some of the many sub-fields:
    - name: String type, Contact name (of person or organization)
    - type: String type, Contact type (user entered or drop-down selection)
    - contactType: String type, Contact contact type (person, organization, or blank)
- tags: List<Tag>
    - name: String type, tag name
    - scheme: String type, tag scheme (url)

        - type: String type, tag type (drop-down selection)
        - vocab: String type, user entered vocabulary
        - label: String type, user entered label
        - uri: String type, (uri)
- identifiers: List<Identifier>
    - scheme: String type
    - type: String type
    - key: String type
- facets: List<Facet>, sub-fields listed here. Facets can be ingested through the file upload process
- citation: String type, Citation as in Journal Article, etc.
- spatial: ItemSpatial type
    - representationalPoint: GIS value
    - representationalPointIsDerived: Boolean (default false)
    - boundingBox:
        - minY: (default null), GIS value
        - maxY: (default null), GIS value
        - minX: (default null), GIS value
        - maxX: (default null), GIS value
- distributionLinks: List<SB2WebLink>, system derived links (from shapefiles or uploaded files)
    - uri: String type, system derived uri
    - title: String type, system derived title
    - type: String type, system derived type
    - typeLabel: String type, system derived type label
    - rel: String type, system derived relationship
- dates: List<ItemDate>
    - type: String type, (drop-down selection provided)
    - dateString: String type, use authorized formats
    - label: String type, describe the date
- files: List<ItemFile>, some of the many sub-fields:
    - name: String type, system derived filename
    - contentType: String type, system derived content type (ex. jpeg/image)
    - pathOnDisk: String type, system derived identifier of where stored
    - url: String type, system derived url of where stored
    - size: String type, system derived size of file in bytes
- webLinks: List<ItemWebLink>
    - type: String type, system derived
    - typeLabel: String type, (user entered or drop-down selection)
    - title: String type
    - uri: String type
    - rel: String type, system derived relationship
    - contentType: String type, system derived content type (example, jpeg/image)
    - hidden: Boolean type
- extents: List<Long>, system derived from footprint studio
- parentId: ObjectId type that is the parent identifier of an item
- hasChildren: Boolean type
- browseCategories: Set<String> (tied to tag type), (user entered or drop-down selection)
- browseTypes: Set<String> (tied to tag type), (user entered or drop-down selection)
- locked: Boolean type, default false

Example JSON

JSON Format Example of a well-described item.

REST Service for Item Overview

The following documentation concentrates on the basics of ScienceBase REST Services for CRUD (creating, reading, updating, and deleting) using the Firefox REST Service Client and JSON, including:

- How to access and set up the Firefox REST Service Client
- How to view a test item from ScienceBase (you created one didn't you?)
- How to create a new item in ScienceBase using the JSON from the GET
- How to update that new item with some example values
- How to delete the new item

REST Service for Item End Point is implemented within ItemController.

https://www.sciencebase.gov/catalog/item is the base url on production.

?? an endpoint that runs itemService.buildMapForJson will also accept additional parameter strings of 'field=' and 'fieldset='. 'fieldset=' is simply a wrapper for a specific set of fields and resolves to a 'field=' parameter.

CRUD Formats

GET: The following formats are implemented/coded within the GET:

- catalog/item/ then the item id followed by '?format='

Example: /catalog/item/4f4e478de4b07f02db48951f?format=html

- html - Sciencebase default
- xml - Not implemented as an item format
- json - runs itemService.buildMapForJson(itemInstance, fields)
- jsonp - runs itemService.buildMapForJson(itemInstance, fields) as download
- atom - runs itemService.buildForATOM(delegate, itemInstance, fields, true)
- modsxml - runs modsXmlItemService.buildForModsXml(delegate, itemInstance)
- fgdc - runs fgdcItemService.buildForFgdcXml(delegate, itemInstance, null, true)
- iso - runs isoItemService.sbJson2IsoXmlString(itemInstance, fields)
- usgsCitation - runs itemService.buildCitations(itemInstance)

POST: The following formats are implemented/coded within the POST:

- catalog/item/ then the item id followed by ?format=

Example: /catalog/item/4f4e478de4b07f02db48951f?format=json

- html - Not implemented
- xml - Not implemented
- json - runs the jsonHandler within post method and also itemService.createOrUpdateItem
- jsonp - runs the jsonHandler within post method and also itemService.createOrUpdateItem

PUT: The following formats are implemented/coded within the PUT:

- catalog/item/ then the item id followed by ?format=

Example: /catalog/item/4f4e478de4b07f02db48951f?format=json

- html - Not implemented
- xml - Not implemented
- json - runs the jsonHandler within put method and also itemService.createOrUpdateItem
- jsonp - runs the jsonHandler within put method and also itemService.createOrUpdateItem

DELETE: The following formats are implemented/coded within the DELETE:

- catalog/item/ then the item id followed by ?format=

Example: /catalog/item/4f4e478de4b07f02db48951f?format=json

- html - Not implemented
- xml - Not implemented
- json - runs the jsonHandler within delete method and also itemService.deleteItem
- jsonp - runs the jsonHandler within delete method and also itemService.deleteItem

REST Services By Example

How to get and set up the Firefox REST Service Client:

Once you have Firefox browser installed and running, do a web search for RESTClient or go to this url: https://addons.mozilla.org/en-US/firefox/addon/restclient/

'Add to Firefox' should be an option (as a button). Once this is done and Firefox has been closed and restarted you can go to the Add-ons Manager > Extensions and verify that RESTClient is enabled and there should be a RESTClient button in the upper right of a browser tab (not the Add-ons Manager tab however). While you are in Add-ons Manager however do a search for an add-on. Type in 'JSON' and add JSONView as an add-on extension. With this installed you will be able to view JSON files opened in your firefox browser with the proper formatting, highlighting, and collapse/expand features.

Setting up your RESTClient to work for these examples you will need 2 Headers:

- Header 1 - Name: Accept, Value: application/json
- Header 2 - Name: Content-Type, Value: application/json

You will also need to use Basic Authorization with your Username and Password that you use to login to ScienceBase.

GET Requests

GET requests are simple, you just send a GET request to the specific URL of the item you are requesting. E.g., https://www.sciencebase.gov/catalog/item/<itemId> where itemId is the ScienceBase item ID of the requested item.

GET specific fields of an item. The user is now able to limit the fields brought back by using the field query syntax added to the URL like this:

<base url><itemId>?format=json&fields=title,subtitle,body,contacts

As of February 2014 contacts and distributedLink fields have further filtering within the field groups like so:

<base url><itemId>?format=json&fields=title,subtitle,body,contacts.name,contacts.type

Using Identifiers in GET requests

You may use item identifiers in the query URL like this: https://www.sciencebase.gov/catalog/item/?scheme=testScheme&type=testType&key=testKey You would do this instead of using an item id field.

Here are some examples of using the GET within the RESTClient:

Fig 1 - a url ending in an item id

*NOTE - if you were to delete the last '4' of the item id and click send what would happen? You would get a Response Headers Status Code of 404 Not Found.

*NOTE - if you view in the Response Body (Highlight) tab, both the field names and case sensitivity are revealed in a much more readable fashion.

Fig 2- RESTClient GET using Fields Filter

Fig 3 - JSONView add-on to view the url json with links, highlighting, expand/collapse

*NOTE - This is also a good example of case sensitivity, the subtitle field is not displaying because it needs to be subTitle.

POST Requests

POST requests create new items within ScienceBase. The simplest POST requires only two fields to return successfully: parentId and title. Here is an example of the most basic POST:

Fig 4 - Basic POST with parentId and title, notice the URL ends with "/item/".

In the Response Body (Hightlight) tab we can see all of the default values that are given to an item, the new item's id and that the title and parentId that we used to create the item are set correctly. You may (or may not) note that this item is inheriting its permissions from an item id that is not the parentId. This is normal as the parent of an item (may be) inheriting its permissions from an item above it; these values are passed down (and dynamic).

Using Identifiers in POST requests

You would not use identifiers in a POST request to 'identify' an item. A POST is used to create new item(s) within ScienceBase. Identifiers should be in the JSON body as input field values only.

More Advanced POST Requests

Adding more fields to your POST request is as simple as understanding the type of request you are sending. In JSON it means adding more key/value pairs that match the key/value pairs you are trying to set. For example, a "subTitle" is a String just like the item title so you would put:

"subTitle":"<subTitle text>"

More advanced types are a little trickier, but an easy way to figure out the correct format is to do a GET on an object that already has these properties and see what type of object the "value" is. alternateTitles, for example, are an array of Strings, so you need to create the array of Strings and then set that as the "value" of the key/value pair you are adding. In REST Client this looks simple:

"alternateTitles":["alt title 1","alt title 2"]

Some values are more complicated than just arrays of Strings. Tags, for example, are represented as an Array of Maps. In REST Client this can be sent as:

"tags":[{"name":"tag1"},{"name":"tag2"}]

A good shortcut is to clip key/value pairs from the results of a GET response and change the values.

*NOTE - The examples above are in addition to the required parentId and title fields.

example: {"parentId":"<parentId>","title":"TEST Title","tags":[{"name":"tag1"},{"name":"tag2"}]}

PUT Requests

PUT requests modify existing items within ScienceBase. A simple PUT uses the itemId at the end of the URL and a key/value pair for a field. Here is an example of a simple PUT:

*NOTE - A PUT overwrites existing data so it is also a way to clear out fields like tags. Be careful as existing tags WILL NOT be kept. Try this:

PUT - {"tags":[{"type":"Label","name":"tag1"},{"type":"Label","name":"tag2"}]}

then

PUT - {“tags”:[{“type”:”Label”,”name”:”tag3”}]}

“tag3” is now the ONLY tag, it does NOT get appended to the list

PUT - {“tags”:””}

this will remove all tags from the response

Fig 5 - A very simple PUT request

Using Identifiers in PUT requests

It is also possible to update a document using one of its identifiers (stored in the identifiers field of the item). The only restriction is that the identifier you are using must be unique within the set of items that your user may update. Otherwise, the REST api will return a 409 (Conflict) HTTP response for more than one matching item.

There are two ways to specify the item identifier. In both cases, you must not include a ScienceBase identifier in the URL. You will simply use the https://www.sciencebase.gov/catalog/item/ base url.

- You may specify the identifier as query parameters. For example, https://www.sciencebase.gov/catalog/item/?scheme=testScheme&type=testType&key=testKey.
- Alternately, if you omit the base ScienceBase identifier and the identifier query parameters, the system will use the first item identifier specified in the JSON data included in the PUT request. For example, with the following item data PUT to https://www.sciencebase.gov/catalog/item/, the system would look for a record with the first identifier. It will completely ignore any subsequent records. WARNING: While it may ignore other records it will acknowledge the second identifier as a PUT item. So the record with the first identifier will now have a second identifier of scheme: "DND", type: "foo", key: "11111" and possibly overwrite another identifier.

{title: "My Updated Title",identifiers: [{scheme: "testScheme", type: "testType", key: "testKey"}],{scheme: "DND", type: "foo", key: "11111"}]}

Fig 6 - A POST using a single identifiers

UPSERT Requests

UPSERT requests will create or update an Item.

An upsert is the combination of "update" and "insert" or POST and PUT. This functionality is useful when you have a identifier for an item, but don't know if it already exists in your user's writeable area of ScienceBase. The upsert API lets you submit a full item document, along with the known identifiers. ScienceBase will look for an existing item that matches the first item identifier included in the document you submit. If it finds it, that item will be updated. Otherwise, a new item is created.

NOTE: you must specify a "parentId" in the item document or ScienceBase will be unable to create the item.

Using Identifiers in UPSERT requests

Upsert requests should be POST-ed to https://www.sciencebase.gov/catalog/item/upsert. You may optionally specify the identifiers as query parameters rather than specifying them in the POST body (e.g. https://www.sciencebase.gov/catalog/item/upsert?scheme=testScheme&type=testType&key=testKey). If the identifiers are not specified as query parameters, the first identifier in the item's data is used. All other identifiers would be ignored.

{parentId: "4f3d3bf1ccf2089542bc925c",title: "My Updated Title",identifiers: [{scheme: "testScheme", type: "testType", key: "testKey"}]}

This example would search for an item matching the testScheme identifier testKey and update or create the item as appropriate.

Exception behaviors:

- If a ScienceBase item id is specified in the POST-ed JSON, that item will be updated if it is writable. Otherwise, 403 or 404 will be returned as appropriate. An item will never be created in this instance.
- If more than one writable item matches the identifier provided, no action is taken and a 409 error is returned.
- If no writable item matches the identifier, but one or more readable items do match the identifier, no action is taken a 403 error is returned.
- If exactly one writable item matches the identifier provided, regardless of the number of readable items that match, the writable item will be updated.

Fig 7 - An UPSERT POST with identifiers in the URL

DELETE Requests

DELETE requests remove an Item from ScienceBase.

A DELETE is pretty simple, although it does carry the requirement that the item must not have any children. To delete just set the Method to DELETE on whatever item you want to delete. You must also be logged in to delete items.

Fig 8 - A DELETE request with the Status Code 200 OK displayed

You may also delete one or more items using the Request Body, in a REST client, specify URL:

https://www.sciencebase.gov/catalog/items

In body, list item ids for deletion:

[{"id": "50a8ff81e4b05de520e7a418"},

{"id": "50a8ff82e4b05de520e7a41c"}]

Using Identifiers in DELETE requests

It is also possible to update a document using one of its identifiers (stored in the identifiers field of the item). The only restriction is that the identifier you are using must be unique within the set of items that your user may update. Otherwise, the REST api will return a 409 (Conflict) HTTP response for more than one matching item.

To use an item identifier for DELETE, you must not include a ScienceBase identifier in the URL. You will simply use the https://www.sciencebase.gov/catalog/item/ base url and attach parameters representing the item identifier. For example, https://www.sciencebase.gov/catalog/item/?scheme=testScheme&type=testType&key=testKey.

In batch mode, using https://www.sciencebase.gov/catalog/items/, you may also use identifiers rather than ScienceBase identifiers. Note that you can mix and match both types.

[{"identifiers": [{scheme: "testScheme",type: "testType",key: "testKey"}]},{"id": "50a8ff82e4b05de520e7a426"}]

Properties of Nested Documents within an Item

You should be able to determine most of the properties of an item by reading the outputted JSON. With attached files however you will only have a link to them. There are many ways that you can save a document into ScienceBase but, luckily, there are a few quick ways to get all of the documents out.

Downloadable Links

Many items will have external links that are added to ScienceBase as webLinks. These webLinks will have the type "download" and the uri will be the link to the download.

Documents Uploaded into ScienceBase

Many times a user will upload data directly into ScienceBase. To download all of the data associated with an item as a zip file (or as the individual file if there is only one) you can go to https://www.sciencebase.gov/catalog/file/get/<itemId>.

Downloading individual items is a little trickier as you must first find the "pathOnDisk." If the file was uploaded directly to ScienceBase and not associated with any types it can be found under the "files" key. Each entry in this will have a pathOnDisk key that contains the necessary path. If the file was associated with a facet (eg, Shapefiles, Rasters, Map Packages) You have to navigate to the facets, find the one that you want (eg. className = gov.sciencebase.catalog.item.facet.ShapeFileFacet for shapefiles) and look under the files key here to find the appropriate "pathOnDisk".

Once you have the pathOnDisk you use the same URL as you would to get the zip, but with the option f=pathOnDisk: https://www.sciencebase.gov/catalog/file/get/<itemId>?f=<pathOnDisk>.

The above pathOnDisk field is still present in the latest iteration of JSON output but a new URL field in both facets > files and files gives a quick link to downloading those individual files. If you are viewing the JSON using a browser with the JSONView add-on these URL links should be clickable.

POST: Multiple Items

Creating multiple items with one REST call is very similar to creating a single item. The first difference is that since we are creating items we will use the items controller: https://www.sciencebase.gov/catalog/items The second difference is that although sbJSON of each individual item is exactly the same you treat each item as an entry in an array:

[{title: "Test item 1", parentId: 5033f109e4b068b9cdc547f8}, {title: "Test item 2", parentId: 5033f109e4b068b9cdc547f8}]

The returned sbJSON will also be in an array and will contain all of the same information that a single call would, for each item:

Fig 9 - A simple Multiple POST request

PUT: Multiple Items

PUT to update multiple items act on the same principals that POST's do. First, the call must be made to the items controller: https://www.sciencebase.gov/catalog/items. For the input sbJSON since we are not calling the item by id in our URL we must put the id of the item you wish to update in the sbJSON for each item:

[{title: "Test item 1.2", id: 504695ede4b0e82aebe2cb0f}, {title: "Test item 2.2", id: 504695ede4b0e82aebe2cb11}]

The returned sbJSON will be of the same type as the sbJSON returned by a mutliple items POST.

Fig 10 - A simple Multiple PUT request

Using Identifiers in Multiple PUT requests

Note that identifiers may also be used within a PUT with multiple items. Read more about using identifiers with PUT in the single-item PUT section above.

[{title:"Test item 1.2",identifiers:[{scheme:"testScheme",type:"testType",key:"testKey1.2"}]},

{title:"Test item 2.2",identifiers:[{scheme:"testScheme",type:"testType",key:"testKey2.2"}]}]

Fig 11 - A simple Multiple PUT request with identifiers

UPSERT: Multiple Items

Upserts for multiple items work very similar to the upserts for single items, but may be batched. The POST should be made to https://www.sciencebase.gov/catalog/items/upsert and will contain an array of JSON objects. The semantics of each individual upsert operation is identical to those described in the section above for single upserts.

[{title:"Test item 1.2",identifiers:[{scheme:"testScheme",type:"testType",key:"testKey1.2"}]},

{title:"Test item 2.2",identifiers:[{scheme:"testScheme",type:"testType",key:"testKey2.2"}]}]

Fig 12 - A simple Multiple UPSERT request with identifiers

DELETE: Multiple Items

You may also delete one or more items using the Request Body, in a REST client, specify URL:

https://www.sciencebase.gov/catalog/items

In body, list item ids for deletion:

[{"id": "50a8ff81e4b05de520e7a418"},{"id": "50a8ff82e4b05de520e7a41c"}]

Asynchronous POST, PUT, UPSERT, and DELETE

If you are doing calls to create, update or delete a large number of items it may be advantageous to not have to wait for all the items to be modified in SB to get your return back. In this case you can use the async parameter which will first return a status page with a 202 accepted header and, when the operations are complete in SB, return the sbJSON of the actions you submitted.

Making an asynchronous call is the same as making a normal call, but with the async parameter set to true. For example if you are doing a POST of multiple items the URL that you are POSTing to will be https://www.sciencebase.gov/catalog/items?async=true.

The response will be a JSON object with the keys requestId and status and a Status Code Header with the value "202 Accepted". The requestId will be the id that you can use to access the requests status at on the async controller. Using this requestId you can continue to check the status of your call at https://www.sciencebase.gov/catalog/async/<requestId>. The status will first return a more direct path to the asynchronous call's page and then the status of the call once it starts to be processed.

Fig 13 - POST request with identifiers and async parameter

Example return:

{"requestId": "504777dae4b010bea98f8231","status": "/catalog/async/53069232e4b0e59bb387a336"}

Uploading Files

Although uploading files to ScienceBase RESTfully technically falls into the POST/PUTs category, it is slightly different and thus deserves its own instructions.

First things first, Headers. As with all REST calls for JSON on ScienceBase we need the correct Accept header of application/json:

Headers:

[name: Accept, value: application/json]

You must also have your JOSSO_SESSIONID from the JOSSO_SESSIONID cookie.

uploadAndCreateItem or uploadAndUpdateItem:

There are two ways to upload files, you can upload files and create a new item with them or you can upload files to an existing item. In both cases you need a ScienceBase Id (sb:id). When you use the uploadAndCreateItem service this sb:id will be the id of the parent item that you want to create this new item under. When you use the uploadAndUpdateItem service the sb:id will be the id of the item you wish to upload files to. Both of these services are located at https://www.sciencebase.gov/catalog/file

The request is made is a multipart/form-data request (you can add this header value with [name: Content-Type, value: multipart/form-data] just to be safe). The parts consist of files, each of which is named "file" and a sbJSON part which is named item. With these parts you can upload as many files as you wish as well as creating (or updating) with sbJSON.

You may also have to manually set your boundary for your files. See example request at the bottom of this page to see all of the headers for the entire request as well as each part.

Files:

At least one file is required (otherwise you should be using the faster item services). Each file's name is "file", its Content-Disposition is "form-data", its filename should be its filename, its Content-Type should be "application/octet-stream", and its Content-Transfer-Encoding should be "binary". You may not have to manually set all of these, but you should be aware of them. As of 8/30/12 you can send as many files as you want, each with the name "file" and they will be iterated through and added.

item sbJSON:

You can also send in sbJSON with your item, this way you do not have to make separate calls creating/updating the item's metadata and uploading files. Its Content-Disposition is "form-data", its name is "item", its Content-Type is "text/plain", its charset is "US-ASCII", and its Content-Transfer-Encoding is "8bit". The content of this should be a string representing the sbJSON you are inputing.

If you are updating a record that already contains files the item's file list will be updated with the new file. If you wish to delete the old files set "files: [ ]" in the sbJSON you submit.

Example Upload Paths:

uploadAndCreateItem service:

https://www.sciencebase.gov/catalog/file/uploadAndCreateItem/[sb:parentId]?josso=[JOSSO_SESSIONID]

uploadAndUpdateItem service:

https://www.sciencebase.gov/catalog/file/uploadAndUpdateItem/[sb:id]?josso=[JOSSO_SESSIONID]

Example Request

This request was caught using reflect.py, which reflects local requests to the command line so you can evaluate them. It is available for download here: https://gist.github.com/raw/814831/fb48dd96467a5e52edeff2010d53f30278926391/reflect.py

```
----- Request Start ----->
/file/uploadAndCreateItem/5033ff1ae4b068b9cdc54853?josso=robb
Accept: application/json
Content-Length: 12747
Content-Type: multipart/form-data; boundary=_txIIVyvBlA6lJIq9-laQOG94lj_hyCS6rKfp4
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.1 (java 1.5)

--_txIIVyvBlA6lJIq9-laQOG94lj_hyCS6rKfp4
Content-Disposition: form-data; name="file"; filename="file1.txt"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

[... file 1 contents ...]

--_txIIVyvBlA6lJIq9-laQOG94lj_hyCS6rKfp4
Content-Disposition: form-data; name="file"; filename="file2.txt"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

[... file 2 contents ...]

--_txIIVyvBlA6lJIq9-laQOG94lj_hyCS6rKfp4
Content-Disposition: form-data; name="item"
Content-Type: text/plain; charset=US-ASCII
Content-Transfer-Encoding: 8bit

{title: 'File Upload Test', provenance:{html: 'File upload from example in the ScienceBase Documentation'}}
--_txIIVyvBlA6lJIq9-laQOG94lj_hyCS6rKfp4--
<----- Request End -----
```

Creating Queries for ScienceBase

Documentation on using the browser URL to do more advanced ScienceBase searches is connected to the topics within this document but has its own document. HERE