Tethys RESTful Web Services Interface

A manual for accessing a Tethys server using a representational state transfer (REST) interface



Tethys, Antioch mosaic, 3rd century from Baltimore Museum of Art

Contents

Introduction 2
XQuery Resource 3
POST – Run XQuery or JSON database query 3
XQuery 3
JSON 4
Optional form variables 8
Collection Resources 9
GET – Retrieval of documents 9
POST – Adding/changing documents 10
POST add (deprecated) 10
POST ODBC (deprecated) 10
POST import 11
POST rebuild 13
POST reindex 14
DELETE – Deletion of documents 14
Attach Resource 14
BatchLogs Resource 14
Tethys Resource 15
Get 15
Put 15
Schema Resource 15
Get 16
ODBC Resource 16
GET 16
References 17

Introduction

Tethys provides web services via a representational state transfer (REST, Fielding, 2000) interface. RESTful architectures identify resources via uniform resource locators (URLs) and provide operations to create, remove, update and delete items associated with the resource. It is

assumed that the reader is familiar with Tethys and this work merely documents the interface that can be used by other programs to interact with Tethys.

Tethys provides a series of resources to the user:

- Attach – Document attachments, e.g. a spectrogram or short waveform.
- BatchLogs – Logs for batch operations, primarily used in rebuilding the database from source documents.
- XQuery – A resource to query the Tethys database.
- Collection resources – These resources represent individual Tethys collections. Current collection resources are:
    - Detections
    - Deployments
    - Events
    - Localizations
    - mediator_cache
    - SourceMaps
    - ITIS
    - SourceMaps
    - SpeciesAbbreviations

An example resource for a server running on port 9779 (the default port) accessed from the server machine would be: http://localhost:9779/XQuery. Examples throughout this document will use localhost as the server, these should be updated with the desired server's address.

Clients can use hypertext transfer protocol (HTTP) to operate on these resources via the methods GET, PUT, DELETE, and POST. In general, GET is used to access part of a resource, PUT to add information to a resource, DELETE to remove part of a resource, and POST to either update or add to the resource. PUT and POST can be somewhat confusing, as any message to the server that requires a multi-part form data (e.g. attachments) must use a POST message. In Tethys, POST is used to add documents to the database and to query. POST is used for queries as the XQuery is passed as field "XQuery" in a multipart form.

XQuery Resource

POST – Run XQuery or JSON database query

POST is the only method supported for resource XQuery. One mandatory body attachment, (XQuery or JSON) is expected with the post operation along with a set of optional form variables.

XQuery

The **XQuery** body should contain a valid XQuery which will be executed (see plan for exceptions). See the Tethys manual for information on XQuery and our extensions to support external collections (e.g. physical data such as sea surface temperature). The result is returned in XML.

The following is an example XQuery. XQueries can be generated automatically from the Tethys Web Client interface.

```
<ty:Result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> {
  (: Determine the taxonomic serial number for Lagenorhynchus obliquidens,
     which are abbreviated as Gg in the SIO.SWAL.v1 map :)
  let $tsn0 := lib:abbrev2tsn("Gg", "SIO.SWAL.v1")
  return
      for $Detections0 in collection("Detections")/ty:Detections[
         Effort/Kind/SpeciesID = $tsn0 and OnEffort/Detection/SpeciesID = $tsn0]
      return
        <Detections>{
            for $Detections2 in $Detections0/OnEffort/Detection[SpeciesID = $tsn0]
            return
              <Detection>{
                $Detections2/Call,
                $Detections2/Start,
                $Detections2/End
              }</Detection>
        }</Detections>
} </ty:Result>
```

JSON

A **JSON** body is converted into an XQuery and executed. The JSON specification contains the following keys and values:

- select – A list of selection criteria. Each criterion is a JSON schema, { }, with the following values:
  - op – Specifies a selection operation such as equality, greater than, etc. These should be valid XQuery operators.
  - operands – A list of operands to which the operator will be applied. The query translator relies on the paths that select elements being in the first position.
  - optype – Indicates how the operands are to be combined with the operator. Valid optypes: binary (or infix which is equivalent) and function. Function indicates that the op should be applied as a function to the operands. Example: "op":"foobar", "operands": ["a", "b"], "optype":"function" → foobar(a, b)

  For a concrete example, this entry would allow one to select detection documents with ≥ 1 on-effort detection. "op" : "exists", "operands" : "Detections/OnEffort/Detection", "optype": "function".
- return – A list of values to be returned. Currently, no functions may be applied to these, but we plan to extend this in the future.
- species – This schema is optional and has up to two keywords:
  - query – Used to specify how the query denotes species names. Without query, we assume that all species are encoded as Integrated Taxonomic Information System (ITIS) taxonomic serial numbers (TSNs). The query allows us access Tethys.xq library functions for converting strings to TSNs. The relevant functions are:

- lib:completename2tsn(latin) which converts a Latin species name to TSN.

```
"query":{
        "op":"lib:completename2tsn",
        "operands": ["%s"],
        "optype":"function"
}
```

- lib:vernacular2tsn(language, common) which converts an accepted vernacular name in a given language to a TSN. ITIS has good support for English, and partial support for a wide variety of languages. In general, we do not recommend querying with common names (e.g. English: Blue Whale) without error checking due to the difficulty in constructing appropriate strings which are case sensitive.

```
"query":{
      "op":"lib:vernacular2tsn",
      "optype":"function",
      "operands":["%s", "English"]
 }
```

- lib:abbrev2tsn(abbrev_map_name, abbreviation). Given an abbreviation map name (see http://your_tethys_server:port/SpeciesAbbreviations or query the SpeciesAbbreviatoins collection for Abbreviations/Name), and an abbreviation, return the TSN.

```
"query":{
       "op" : "lib:abbrev2tsn",
       "optype" : "function",
       "operands": ["%s", "NOAA.NMFS.v1"]
}
```

The translation is specified with op, optype, and operands values similar to those in the select statements. The op key specifies the function, the optype is "function," and the operands list is the set of values on which the function will operate. Species names will be substituted into the function in the position occupied by the special string "%s". The example query below translates Latin species names to TSNs.

- return – Translates TSNs contained in SpeciesId elements contained in the output to human readable formats. As with query, we expect op, operands, and optype. Relevant functions with examples of result entries:
  - lib:tsn2vernacular(tsn, language) – Specify a language other than Latin (e.g., English, Spanish). If there is no entry in the language, the Latin is used.

```
"return":{
  "op":"lib:tsn2vernacular",
     "operands": ["%s", "Spanish"],
     "optype": "function"
}
```

  - lib:tsn2completename(tsn) – Translate to Latin name.

```
"return":{
  "op":"lib:tsn2completename",
```

```
            "operands": ["%s"],
            "optype": "function"
        }
```

- lib:tsn2abbrev(tsn, abbreviation_map_name) – Translate to an abbreviation.

```
        "return":{
          "op":"lib:tsn2abbrev",
            "operands": ["%s", "SIO.SWAL.v1"],
            "optype": "function"
        }
```

The translation is specified with op, optype, and values similar to those in select and the query above.

- Specifies a language (Latin, English, Spanish, …) into which TSNs in the output are translated. If a species is not supported in a language, its Latin name is returned.

- enclose – If present, expects an integer. A value of 1 will cause the generation of enclosing XML elements. Defaults to 0. Enclosing elements are generated for each level of selection criteria that are present.

  At times, this can lead to surprising results. Suppose one wrote a query that included Detections/OnEffort/Detection/Start in the return list but did not introduce criteria on which on which specific on-effort selections were selected. As there are multiple Detection elements, one might expect enclose to produce a <Detection> to wrap each <Start>. This is not the case when there are no selection criteria that involve the Detection element. Enclose only works when there are selection criteria. To force the system to produce an enclosing <Detection> element, add a criterion to Detection, such as the following select clause which just checks for existence:

  {"optype":"function", "op": "exists", "operands": "Detections/OnEffort/Detection"}

- namespaces – If present, expects an integer. A value of 0 removes namespaces in output documents which is good for some clients. Defaults to 1.

- enclose_join – When multiple collections are referenced, it is usually desirable to keep associated records from the join relationship together. By default, an element <Record> is generated that encloses results from each document group, letting users determine things such as what deployment characteristics are associated with what detections, etc. The name can be changed from "Record" by providing a string argument to enclose_join. Use the value "omit" to prevent the production of this element.

  Note that if you are parsing the returned XML using XPath, an or expression can allow one to write a single XPath that handles the optional element. For example, to process /Result/Detections or /Results/Record/Detections, us expression /Result/Record|./Detections.

Example JSON query:

```
{
```

```
  "select": [
    {
       "op": "=",
       "operands": ["Deployment/Project", "SOCAL"],
       "optype": "binary"
    },
    {
       "op": "=",
       "operands": ["Detections/Effort/Kind/SpeciesID", "Bm"],
       "optype": "binary"
    },
    {
       "op": "=",
       "operands": ["Detections/Effort/Kind/Granularity", "binned"],
       "optype": "binary"
    },
    {
       "op": "=",
       "operands": ["Detections/Effort/Kind/Granularity/@BinSize_m", 60],
       "optype": "binary"
     }
  ],
  "return": [
    "Deployment/Project",
    "Deployment/Site",
    "Deployment/DeploymentID",
    "Detections/Effort/Start",
    "Detections/Effort/End",
    "Detections/Effort/Kind",
    "Detections/Algorithm",
    "Deployment/DeploymentDetails"
  ]
"species":{
  "query":{
         "op":"lib:tsn2abbrev",
         "operands": ["%s", "SIO.SWAL.v1"],
         "optype": "function"
   },
  "return":{
          "op":"lib:completename2tsn",
          "operands":"%s",
          "optype":"function"
   }
}
```

This JSON query would be translated into the following XQuery

```
import module namespace lib="http://tethys.sdsu.edu/XQueryFns" at "Tethys.xq";
declare namespace ty = "http://tethys.sdsu.edu/schema/1.0";
<ty:Result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> {
  let $tsn0 := lib:abbrev2tsn("Gg", "SIO.SWAL.v1")
  return
      for $Detections0 in collection("Detections")/ty:Detections[
         Effort/Kind/SpeciesID = $tsn0 and OnEffort/Detection/SpeciesID = $tsn0]
      return
        <Detections>{
            for $Detections2 in $Detections0/OnEffort/Detection[SpeciesID = $tsn0]
            return
              <Detection>{
```

```
                $Detections2/Call,
                $Detections2/Start,
                $Detections2/End
              }</Detection>
        }</Detections>
} </ty:Result>
```

The selection operations can reference XQuery functions such as exists or count. For example, to ensure that a Detection element existed, one could use the following as one of the items in the select list.

```
{
   "op": "exists",
   "operands": ["Detections/OnEffort/Detection"],
   "optype": "function"
}
```

The special op value “range” can be used to check for a value between a range:

```
{
   "op": "range",
   "operands": [
      "Deployment/DeploymentDetails/Longitude",
      350.0,
      15.0
      ],
   "optype": "function"
}
```

Range requires three operands: [path, min, max]. It produces XQuery criteria such that returned items have a value lies between the specified min and max. It has a special behavior for Latitudes. If min > max as in the example above, it assumes that the region should cross the prime meridian and constructs criteria such that (Latitude >= min and Latitude <= 360) or (Latitude >= 0 and Latitude <= max), allowing the specification of ranges about the prime meridian.

Optional form variables

In addition to the mandatory XQuery or JSON multipart form variable, one can specify:

- plan – Allows the user to request information related to the query in lieu of execution by specifying a value other than 0. Valid values and their behavior:
    - plan=0 (or omitted) – The query is executed (the same behavior as when plan is omitted).
    - plan=1 – The database manager will return an XML document describing how the query will be executed. This allows one to examine details such as whether or not indices are used. See the chapter in the *Oracle Berkeley db XML Getting started Guide* on verifying indices using query plans for details.

CAVEATS: Does not currently support external XQuery queries using external collections.

  - plan=2 – For JSON queries, returns XQuery that is produced by the JSON specification.
- dataType – Type of data to return. Defaults to XML. *Jeff, please add documentation for plot and save*.
- species – May be a JSON string of the same format described above in the JSON queries. If species is declared both in the JSON query and as a form variable, the value in the JSON query will be used. Specify a language for species input/output: Latin, English, Spanish, or French. This form should only be used for XQuery, there is a separate mechanism to embed this in the JSON queries as described above.
- namespaces –If 0 (default 1), namespaces are removed from the output. Useful for some clients that do not have sophisticated namespace processing.
- XSLT – If present, content must be a string containing a valid extensible stylesheet language transformation (XSLT) that is applied to the result of the query if it is successfully executed.

Collection Resources

GET – Retrieval of documents

There are two ways in which resources can be retrieved, via the XQuery resource (described in its own section) or via the GET operator on the resource.

GET on a resource URL (e.g. http://localhost:9779/Detections) has the following behaviors:

- No arguments – Returns the number of XML documents in the collection

- Path argument – Interprets the path argument as an XPath query. The namespace tethys.sdsu.edu is bound to the abbreviation ty. Example query:
  `http://localhost:9779//Detections/ty:Detections[DataSource%2FProject = "SOCAL" and DataSource%2FSite = "M" and DataSource%2FDeployment= 34]`
  Note that standard URL escapes such as %2F for / must be used for /s and other characters that are for XPath instead of part of the URL.

- Document Identifier parameter – Document identifiers are derived from the basename of documents that are submitted to Tethys. This is specified via ?DocId:
  `http://localhost:9779//Detections/?DocId= SOCAL45H_HF_Gg_Lo_jst`
  where SOCAL45H_HF_Gg_Lo_jst was derived from detection document SOCAL45H_HF_Gg_Lo_jst.xls that was submitted to the database.

- Report indices on collection with the ?Indices parameter. Shows a list of indices that exist on the container. These are database indices that are used to increase the speed of queries.

POST – Adding/changing documents

New documents can be added with the POST operator. POST supports several methods for adding new documents that are specified as part of the path: add, ODBC, rebuild, and reindex. It is assumed that the user is familiar with importing documents in Tethys, details can be found in the Tethys manual.

Note that POST add and POST import are legacy methods and are likely to be removed in a future release. Any new work should use POST import.

POST add (deprecated)

The add method can be used to add Microsoft Office Excel spreadsheets, XML documents, and comma separated value documents. It supports the following variables and multipart body components:

Variables:

- overwrite – Overwrite an existing document: 0 not allowed (default if absent), 1 allowed
- import_map – Specifies the import translation map to be used. This maps from user field names to Tethys names. For Excel spreadsheets only, this can also be specified by adding the text Parser to the first row of any column on the MetaData sheet and writing the import map name in row two of the same column.
- species_map – Specifies a translation map between local abbreviations and Latin names. As with import_map, Excel files can embed this in a column called SpeciesAbbreviation on the MetaData sheet with the map name underneath it. Collections that do not report species or that use taxonomic serial numbers directly need not worry about this variable.

Attachments

- data – The document file that is to be added. Mimetype must be specified.
- Attachment – File that is to be associated with this document, the mimetype should be specified. Each attachment is expected to be referenced by name within the data. Currently, attachments are only available for detections, and are typically used to show an image (e.g. spectrogram) or a short clip of audio data. Attachment may be repeated. If many files are to be added, they can be placed in a zip archive. Image files should be in subfolder image and audio files in subfolder audio.

Resources that expect attachments will verify submissions and report any missing attachments. This can be used as a strategy to avoid having to parse the document being submitted on the client side. In our document submission client, we submit a document, check to see if missing attachments are the only problem, and if they are prepare a new submission with the needed files. This does require the document to be transmitted twice, but saves the additional burden of writing parsing tools for anything but the XML that is returned from the server.

POST ODBC (deprecated)

Open database connectivity (ODBC) is designed to allow data interchange between multiple formats and ODBC libraries let Tethys import data from a wide variety of sources. See the data import section of Tethys for details.

The ODBC method is similar to the add method. Data added by ODBC can either take the form of a file that is transmitted to the Tethys server, or instructions to access a network resource. Both methods share the following variables and body parts:

- import_map – as above
- species_map – as above
- overwrite – as above
- Attachment – as above

ODBC file submission expects a data file attachment as in the add method. The ODBC connection string will be automatically determined for supported types (Excel, XML, CSV, Access). See the Tethys manual for details.

Network submissions expect a ConnectionString parameter that specifies how the data source is to be opened.

When connecting to databases, it is common to import multiple documents at once (e.g. many deployments stored in a separate database). Document names will be automatically generated for each instance. As the number of documents may decrease from one database read to another (e.g. somebody deleted records), it is not recommended to use the overwrite parameter when reading database records. Rather, clear the collection (or at least documents derived from database records within the collection), then import the database.

POST import

Import is a post interface designed to replace POST add and POST ODBC.

POST expects a series of form multipart bodies.

- specification – File containing XML that specifies the how data will be formatted:

```
<?xml version="1.0" encoding="UTF-8"?>
<import>
    <!-- Name that will be used for the associated XML document.
    If more than one document is generated, sequential numbers
    will be assigned, e.g. name1, name2, ...
    If absent, basename of first source will be used, e.g.
    if source is SOCAL32M-HF-SBP.xlsx, use SOCAL32M-HF-SBP.
    -->
    <docname>name</docname>
    <!-- Replace an existing document? -->
    <overwrite>true|false</overwrite>
    <!– Name of mapping between user data and Tethys
        Must exist in SourceMaps collection -->
    <source_map>import_map_name</source_map>
    <!-- For documents that refer to species, how are species
        declared:
            TSN - ITIS taxonomic serial number (default if absent)
            Latin - scientific name of taxon
            abbreviation map name - Name of mapping in collection
                SpeciesAbbreviations
    -->
    <species_map>TSN</species_map>
    <sources>
        <source>
            <!-- What is being sent?
```

```
            file - A file in the multipart body attachment.  It is
            assumed that the file type can be derived from the file
            extension and an appropriate ODBC connection string will
            be generated by the server.
            resource - A resource accessible by ODBC on the remote
            server.
            -->
            <type>file|resource</type>

            <!-- name distinguishes this source from other sources
                 and is only required when multiple sources are present
                 Although it is recommended.
            -->
            <name>sourcename</name>

            <!—- Required under the following conditions:
            Type is file or
            Resource will be establishing an ODBC connection to a
            File, regardless of the file type.  Examples:
            Excel file
            Access database that is not served on the network
            Comma separate value (CSV) file(s) – Each file is considered
              a table. Multiple files can be combined into a zip file
              but the transfer must be done as type resource with a connection
              string.
            -->

            <file>filename</file>

            <!-- ODBC connection string.
            Optional for type file.  If specified, remote system
                will not attempt to guess the connection string.  For
                example, a file ending in .xslx will result in the server
                setting up a connection string to read Excel files.
            Required for type resource.
            -->
            <connectionstring></connectionstring>
            <odbc>
               <param1> … </param1>
               <param2> … </param2>
            </odbc>
        </source>
        <!-- Repeat source as needed.
            Name elements must have unique values -->
    </sources>
</import>
```

The data specification must be enclosed within an element called import. Children include docname, overwrite, sources, and species_map.

The docname specifies the name that the data will have within the collection to which they are added and must be unique. For documents that have a top-level Id element, we strongly recommend that the docname be the same as the Id although this is not mandatory. The collection itself is specified based on the URL to which the data are posted (e.g., http://localhost:9779/Detections would add the data to the set of detections. Data cannot be overwritten unless the optional overwrite element contains the word true.

Documents that contain species information are interpreted as follows and will sometime use the speciesabbr element to specify a coding scheme:

1. If the document is XML, we expect numerical taxonomic serial numbers (TSNs as per the Integrated Taxonomic Information System).
2. When documents are translated to XML, any element that marks a species will accept TSNs or Latin names. If the speciesabbr is populated with anything other than TSN or Latin, we will first look for codings within a SpeciesAbbreviations document where Abbreviations/Name matches the specified speciesabbr.

- SourceFiles – One form variable for each file source specified in the sources. Mimetype must be specified. Example: if a source specified:
  <type>file</type> <name>boobear.xls</name>
  then there should be a form variable boobear.xls that contains the filename, mime media type, and file handle.
- Attachment – File that is to be associated with this document, the mimetype should be specified. Attachments may only be specified when a single document is being added. Each attachment is expected to be referenced by name within the data. Multiple attachments may be handled in one of two ways:
  - Repeat Attachment label. Some clients do not allow this and Attachment1, Attachment2, … is also acceptable.
  - Create a zip archive of file attachments and attach it. Filepaths must match the references.

  Currently, attachments are only available for detection and localization documents, and are typically used to show an image (e.g. spectrogram), a short clip of audio data, or bespoke (custom) data. Resources that expect attachments will verify submissions and report any missing attachments. This can be used as a strategy to avoid having to parse the document being submitted on the client side. In our document submission client, we submit a document, check to see if missing attachments are the only problem, and if they are prepare a new submission with the needed files. This requires the document to be transmitted twice, but saves the additional burden of writing parsing tools for anything but the XML that is returned from the server.

Important notes for disaster recovery. When only simple files (comma separated values, spreadsheets, xml) are sent, the source documents will be saved on the server, and the server's rebuild commands will be able to reconstruct the database if it is lost or corrupted. When more complicated resources are used (e.g. a database), source documents are not stored as multiple snapshots of a database are neither practical nor desirable.

POST rebuild

This method is used to rebuild a method from the source data that was submitted to the Tethys server. This is primarily used for disaster recovery.

Note that this does not include database records. We do not store these as we assume that users backup their databases and can simply re-run a query. Two variables can be passed to this resource method:

- clear: 0 – Do not clear before rebuilding (default), 1 – clear before rebuilding.
- update: 0 – Only update if a document does not exist (default), 1 – Always update.

Collection rebuild jobs can be quite long when there is a large database. As a consequence, a batch log identifier is assigned and returned as the result of the operation. To check the result of a rebuild, access the BatchLogs resource.

POST reindex

Sets indices to defaults and reindexes the container. Backup first and do not use this lightly. Reindexing can take a long time. Smaller collections such as instrument deployments and calibrations are typically very fast and reindexed in tens of seconds. In contrast, very large containers, such as a detections container with tens of millions of individual detections might take an hour to reindex.

DELETE – Deletion of documents

Documents may be deleted by invoking the DELETE method on a collection. A mandatory keyword argument DocID must be used, which has the same format as for the GET method. To remove all documents, the special DocID `<*clear*>` should be used. Note that this is irreversible.

Attach Resource

The attach resource is used for retrieving attachments associated with collections via a GET operation. It uses a path argument to specify the attachment. The first element of the path is the collection resource (e.g. Detections) followed by the document identifier which is usually composed of the basename of the file that was submitted. The attachment is specified with a keyword argument indicating the attachment type and its name. Currently the only valid attachments are Image and Audio.

Example: If image UknownPhenom.jpg was submitted with an Excel spreadsheet ALEUT02BD_MF_MFAOrca_ajc.xls, one would retrieve the image using a GET on URL:

http://localhost:9779/Attach/Detections/ALEUT02BD_MF_MFAOrca_ajc?Image=UnknownPhenom.jpg

BatchLogs Resource

BatchLogs are used by operations that may take longer than a client can be expected to wait. Instead, an identifier is returned to the user and this is used as the path for a BatchLogs GET request.

Example:
http://localhost:9779//BatchLogs/DetectionsRebuildX32J97.log

Tethys Resource

The Tethys resource is used for controlling the Tethys server itself.

.

Get

The following GET operations are allowed on the Tethys resource:

- Tethys/ping – Returns an XML document if the server can respond:
  ```
  <Tethys>
      <ping>alive</ping>
  </Tethys>
  ```

- Tethys/performance_monitor – Reports status of the query performance monitor (on|off):
  ```
  <Tethys>
      <performance_monitor>on|off</performance_monitor>
  </Tethys>
  ```
- Tethys/cache – Reports if Tethys is currently using the cache to service requests that have been recently executed. Querying a cached document is faster.
  ```
  <Tethys>
    <cache>yes|no</cache>
  </Tethys>
  ```

Put

Several Put commands are allowed on the Tethys resource:

Tethys/shutdown – Exit the server cleanly.

Tethys/cache/ACTION – Controls caching of XQuery/JSONQuery requests. ACTION:
- on – Enables caching.
- off – Disables caching.

Tethys/checkpoint – Create a checkpoint in the transaction journaling system. This should be done preferably before backing up or moving the database. Tethys will automatically do this anytime the server is restarted.

Tethys/performance_monitor/ACTION – Set the performance monitor state. ACTION:
- on – Start or continue monitoring performance.
- off – Disable monitoring performance.
- clear – Resets the performance monitor counts. Starts performance monitor if needed.

Example using curl URL tool: curl -X get http://localhost:9779//Tethys/ping . Graphical user interfaces are available in tools like insomnia and postman. Most languages have libraries for sending http and https packets, e.g. requests library in Python.

Schema Resource

The schema resource lets users query the schema that are used for the various collections. The only supported operation is Get.

Get

The Schema resource requires a path element that is the root element of documents associated with a collection. For example, the root element for all Deployments is Deployment. This returns the XML schema document (XSD) or related information (see below). Two optional query parameters are supported.

- format = JSON | XML (default). Returns the XML specification of the schema document or a representation of it using the Badgerfish convention for converting XML to Javascript object notation (JSON).
- parsetree = true | false (default). This option is only valid when ?format=JSON. This returns a parsed tree representation of the schema document, discarding many things that are not usually useful such as whether an element is a built-in type, simpleType, or complexType. It consists of a set of nested dictionaries. Each element consists of list with six elements:
    - The minimum number of times the element can be present (0|1)
    - The maxmum number of times the element can be present (usually 1 or Infinity)
    - Type information. Not all elements have type information. If there is none, a null value is present. Otherwise, a list of two items will be present. The first contains the type information itself, e.g. xs:string, xs:dateTime, etc. The second item is a dictionary that contains any auxiliary information related to the type. As an example, taxonomic serial numbers (TSNs) are xs:integer types that represent a species. Users frequently specify the names using a string, such as a Latin species name. The SpeciesID element has a dictionary with key special and value ITIS, which indicates that special processing may be needed.
    - Attribute information. A dictionary containing attributes. Each attribute has a list with a type information (as above for the element) and any documentation of the attribute.
    - Documentation of the element.
    - Children. This is a dictionary of child elements, with each element having tuples as described above.

ODBC Resource

Then open database connectivity resource is designed to provide information about Microsoft ODBC capabilities on the server platform. ODBC is used in data import, see the Collections Resource POST import documentation on page 11 for details.

GET

GET and GET drivers will return a plain text list of ODBC drivers supported on the server. These may be used to help generate proper connection strings.

References

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, The University of CA, Irvine, Irvine, CA.