Tethys R Client

CONTENTS

1 Introduction 3
2 R Package Requirements 3
3 Using the Tethys R Client 4
3.1 Executing queries 4
3.1.1 Simple declarative query language 5
3.1.2 Executing XQuery 9
3.2 Extracting data from the results of a query.... 10
3.2.1 Extracting data from specific fields 10
3.2.2 Converting XML returned from a query to a data frame. 10
4 Working with XML.... 14
4.1 The XML/xml2 Libraries 14

1 Introduction

Analysis of Tethys data in R begins with accessing the data. There are several ways to do this:

- Use the RClient Tethys class to execute queries with the R programming language (chapter 3). Results are returned as a named list. The name xml contains the raw XML text returned by the server. In most cases, the routines will parse most of the XML into dataframes. If there is information that was not parsed into a dataframe, it can be extracted using XML libraries (see chapter 4).
- Run a query in the web client interface (e.g., http://mytethys.domain:9779/Client[1]). Results may be saved as XML and then imported. Use the Save to functionality to select the format and save it. See chapter 4 for details on working with XML.
- Save output from the Data Explorer program as a comma separated value file and load it using R's read_csv function. Data Explorer provides a simplified view of the Tethys data and reduces everything to a table. This reduces the complexity of working with the data, but does not offer as much functionality.

2 R Package Requirements

The following Comprehensive R Archive Network (CRAN) library packages should be installed:

- xml2 – A library to work with XML files. This package is recommended over the XML package due to speed and memory efficiency.
- XML – A library to parse XML files
- dplyr – Tidyverse library for data manipulation
- stringr – Tidyverse library for string manipulation
- jsonlite – Library for building Javascript object notation (JSON)
- parsedate – Library for parsing timestamps.
- httr2 – Library for http:// and https:// communication which is how the package communicated with the Tethys server.

Packages can be installed using the install.packages command, e.g.

```
# Install packages for communicating with a Tethys server

# and manipulating resultant XML

install.packages(c("xml2", "XML", "stringr", "jsonlite", "parsedate", "httr2", "dplyr"))
```

or if you use RStudio you may use the Tools/Install Packages menu option which should appear after opening the tethys.r file. Most of these packages do not need to be accessed directly, but provide support for the Tethys R client. If you need to access a specific package such as xml2, use the library command:

[1] Update this URL for the Tethys server that you are using.

```
library(xml2)
```

3 Using the Tethys R Client

Code for accessing Tethys data from R is available in the RClient subfolder of the Tethys distribution. The R client is preliminary and is not currently set up as an R package. To use it, you must source the tethys.r. Throughout this manual, we will write commands that can be executed in R `in this font and color`. The R prompt (>) will be omitted to make it easier to copy and paste examples.

```
source("path/to/tethys/RClient/tethys.r")
```

This will make the tethys class available. Tethys is a reference class, so methods are separated from the class instance by a dollar sign ($). Start by creating an instance of the Tethys object. You must know the machine identifier for the Tethys server, the port on which it is running, and whether or not the server is running in encrypted mode. Most Tethys users do not turn on encryption as it requires a signed security certificate to work well.

```
t <- tethys$new("my.server.gov", port=9779, secure=FALSE)
```

If you are running the server on the same machine as the R process and you have not changed the port or security settings, you can use the default arguments:

```
t <- tethys$new()  # Shortcut when server/client on same machine
```

To test communication, you can use the ping() method which returns TRUE if we can communicate with the server:

```
t$ping()
[1] TRUE
```

Should ping return FALSE, there are several likely causes:

- Tethys may be down. Try to connect from another machine if possible. The easiest way to do this is with the web client as it does not require software installation. Navigate to http://server.machine.name:9779/Client (use https:// if the server was configured with security enabled). If you can load the page on your machine, you may have an R issue or the R program may need a firewall exemption. If you cannot, try the same test on a machine that has been able to connect to the server in the past.
- Verify that Tethys is running on the server. Your administrator should be able to check this.

3.1 Executing Queries

The client supports two forms of query, simple declarative query language that is specifies a set of text-based selection constraints and data to return, and XQuery, a standard mechanism for querying XML databases. Simple query language is only capable of a subset of XQuery's functionality, but it is suitable for most users and does not require learning XQuery.

3.1.1 Simple declarative query language

The Tethys class supports the following methods that are designed to enable users to write queries without having to learn a complicated query language:

- getDeployments – Retrieve information about instrument deployments.
- getCalibration – Retrieve information about instrument calibrations.
- getDetectionEffort – Retrieve information about which taxa we have been looking for, where we have been looking, and how we have been looking.
- getDetections – Retrieve information about what we have actually found.
- getLocalizationEffort – Retrieve information about where and how we have attempted to localize sounds.
- getLocalizations – Retrieve localization information.

All of these rely on syntax that indicates a set of criteria that must be met and optionally a set of information that will be returned in the query. For each of the query functions, there is a default set of values that are returned, but these can be customized to contain specific types of data.

The first argument of all of these uses the same syntax to specify the criteria being selected. For example, if we wanted to see all information about instruments deployed at 800 m depth or deeper that had a sample rate of greater than or equal to 192 kHz, we could provide the following string to getDeployments:

```
# Throughout these examples, we will assume that variable t contains
# an instance of the Tethys object.
xml <- t$getDeployments("DeploymentDetails/ElevationInstrument_m <= -800 and
    SampleRate_kHz >= 192")
```

The results could then be processed using the methods described in the section 3.2 (p. 10) on working with XML. In order to know how to construct a query you first need to have an understanding of the data that are being queried. These can be found by asking to have the list of elements associated with a collection displayed:

```
# Open a schema in a web browser.  Argument should be the root element of a
# collection document:
# Deployment, Calibration, Ensemble, Detections, or Localizations
t$schema("Deployment")
```

This will open a table in a web browser that lists the names of fields[2] in the schema, their type, the minimum and maximum number of times they can appear as a child of their parent, and a field description. Fields that we consider to be self-describing may not have a description. Valid arguments to the schema method can be found in

Table I.

[2] In XML, fields are called elements or element attributes, we use the name field here as it is familiar to more people.

Table I – Collections and the root element used with the schema method.

Collection	Root element	Collection purpose
Calibrations	Calibration	Results of transducer (microphone, hydrophone) calibration.
Deployments	Deployment	Details about instrumentation that has been deployed.
Detections	Detections	Details about passive acoustic monitoring detection effort and results.
Localizations	Localize	Details about localization effort and results.
Ensembles	Ensemble	Virtual instruments consisting of multiple deployments.

Comparisons of fields and values are based on operators that are common in many languages (Table II) and the **field name must appear on the left-hand side of the comparison**.

Table II – Simple query language comparison operators.

Operator	Meaning
=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Multiple comparisons must be joined by the word "**and**," indicating that all conditions must be true. The "or" keyword is not available in the simple query language[3].

Field names consist of a path of field names separated by forward slashes (/). For example, the deployment sample rate field path is:

Deployment/SamplingDetails/Channel/Sampling/Regimen/SampleRate_kHz.

When a field only appears once in the schema, the path can be omitted and one can just write SampleRate_kHz. This is not always the case. For example, in the deployments collection, Longitude fields exist in multiple places (Figure 1).

[3] Disjunction (or) is permitted in XQuery, the query language to which the simple query language is translated. However, disjunction is only allowed when a pair of fields share the same common ancestor. Allowing this would require additional complexity that we do not wish to introduce into simple queries. If this is really needed, one can learn XQuery or generate an XQuery from simple query language or the web client and modify it. To generate an XQuery, add plan=2 to any of the query functions and the result will contain the XQuery that would have been executed.

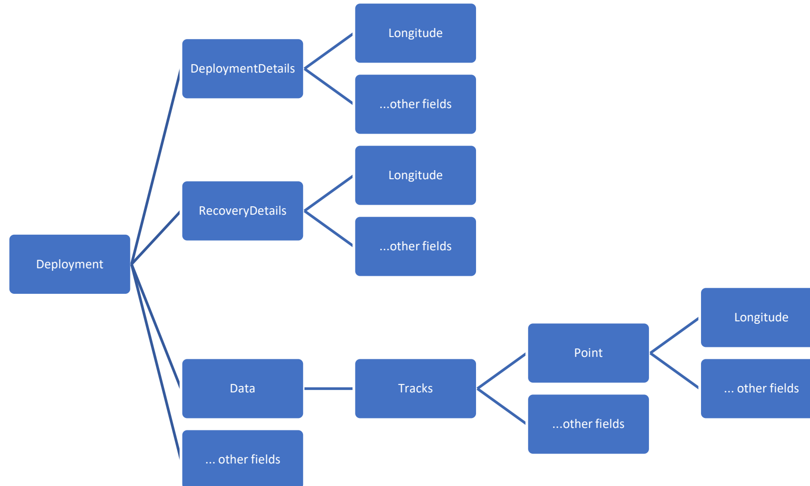

Figure 1 – Field names may be repeated as children of different fields.

Writing selection criteria such as "Longitude > 30" are not sufficient for the parser to determine which Longitude is desired. Either use the entire path, or a distinct subset of it, e.g., "DeploymentDetails/Longitude > 30". Failure to do so will produce an error except in rare cases where it is reasonable to assume a preference for one over the other.

It is possible to use constraints across schemata. For instance, one might be querying detections and wish to add a constraint that they be restricted to a specific deployment Site. The Site field can be used in the query. If a field appears in both schemata, it will default to the one in the schema most associated with the query. For example, in the deployments collection, Deployment/DeploymentId is a number associated with the Nth deployment at a site or the Nth set of deployments. In the detections schema, Detections/DataSource/DeploymentId indicates the identifier for the deployment the detections are associated, that is the deployment where Detections/DataSource/DeploymentId = Deployment/Id. Consequently, if one wishes to filter by deployment's DeploymentId, one must use "Deployment/DeploymentId" in the query. When fields are referenced from multiple collections, constraints are automatically added to the query to ensure that related documents are matched when possible.

If a user simply writes a set of selection criteria, a default set of return values appropriate to the type of query will be returned. However, this can be overridden by using the **return** keyword and a comma-separated list of fields to be returned. For example, suppose we were interested in a list of all deployments in water less than 100 m (instrument elevation > -100), their sample rate, the number of bits per sample, and deployment depth. The following query will accomplish this:

```
dep <- t$getDeployments('DeploymentDetails/ElevationInstrument_m > -100 return I
   d, SampleRate_kHz, SampleBits, DeploymentDetails/ElevationInstrument_m')
```

As a final example, we ask for which species in the Northern hemisphere did we have effort where detections were reported as binned presence (presence/absence over a fixed time bin):

```
eff <- t$getDetectionEffort('Granularity = "binned" and DeploymentDetails/Latitu
   de > 0')
```

Note that even though we queried on detections, we were able to reference deployment fields to filter where we had detection effort. When executed on the demonstration database, the result looks like this:

```
cat(eff$xml)
<Result>
  <Record>
    <Detections>
      <Id>CINMS04C_automatic_UO</Id>
      <DeploymentId>CINMS04-C</DeploymentId>
      <Start>2008-10-15T00:00:00Z</Start>
      <End>2008-12-04T01:02:30Z</End>
      <Kind>
        <DetectionsGroup>1</DetectionsGroup>
        <SpeciesId>Odontoceti</SpeciesId>
        <Call>Clicks</Call>
        <Granularity BinSize_min="1.25">binned</Granularity>
      </Kind>
    </Detections>
  </Record>
  <Record>
    <Detections>
      <Id>CINMS05B_automatic_UO</Id>
      <DeploymentId>CINMS05-B</DeploymentId>
      <Start>2008-12-04T00:00:00Z</Start>
      <End>2009-02-21T11:14:28.000002Z</End>
      <Kind>
        <DetectionsGroup>2</DetectionsGroup>
        <SpeciesId>Odontoceti</SpeciesId>
        <Call>Clicks</Call>
        <Granularity BinSize_min="1.25">binned</Granularity>
      </Kind>
    </Detections>
  </Record>
 <!-- other records ... -->
</Result>
```

To work with this type of XML data that is returned as a character vector, we need to either extract out specific information or convert some portion of the XML to a dataframe (see section 3.2, p. 10).

In most cases, the simple query language methods will attempt to extract dataframes from queries for you. The function names will show what fields are available in a result:

```
names(eff)
[1] "xml" "effort" "kinds"
```

```
eff$effort
```

3.1.2 Executing XQuery

Most users will not need to use XQuery and we suggest skipping to the next section unless you have a specific need.

The XQuery language is used for querying data from XML databases or documents. The vast majority of queries can be accomplished with the simple query language which writes XQueries for you. There is a brief introduction to XQuery in the main Tethys manual, and for those interested in learning the language we highly recommend Walmsley (2006).

The Tethys class has two methods for executing XQuery:

- XQuery(query, stylesheet)
- XQueryTethys(query, stylesheet)

which both take identical character vector arguments. The mandatory query argument is the XQuery database query. Stylesheet is optional and not needed for most users. When present, it contains an XSLT stylesheet (Kay 2008), a specification for transforming XML. While the description of XSLT is beyond the scope of this manual, it is quite powerful and can be used to rewrite XML or to transform it into other formats (e.g., HTML, JSON). Interested users are referred to one of the many books and web sites that describe XSLT.

The XQueryTethys method performs the same operations as XQuery, but prepends the following lines to the query:

```
declare default element namespace "http://tethys.sdsu.edu/schema/1.0";
import module namespace lib='http://tethys.sdsu.edu/XQueryFns' at 'Tethys.xq';
```

XML supports namespaces, which allow the same field name (XML element) to have multiple meanings, much like the international telephone system's country code allows the same telephone number to be assigned to different people in different countries. The first line states that fields (elements) used in this query will by default be assigned to the Tethys namespace which is generally what is desired.

As an example, on the demonstration database, the following XQuery for the number of deployments in the southern hemisphere:

```
t$XQueryTethys('count(collection("Deployments")/Deployment[DeploymentDetails/Lat
   itude < 0])')
```

[1] "4"

If we run the *same query* without placing the elements in the Tethys namespace explicitly, we do not receive the results that we expect:

```
t$XQuery('count(collection("Deployments")/Deployment[DeploymentDetails/Latitude
   < 0])')
```

[1] "0"

as the elements were not in the Tethys namespace. One can add additional code to the XQuery to place the elements in the Tethys namespace, but it is usually easier to use the XQueryTethys method instead of the XQuery method.

The second line that XQueryTethys prepends (*import module* …) makes available a set of utility functions designed for Tethys that enable utility functions such as mapping between Latin taxonomic names, abbreviations, and Integrated Taxonomic Information System (ITIS) serial numbers that Tethys uses internally to represent taxonomic information. For further details about the utility functions and namespaces, see the Tethys manual.

3.2 EXTRACTING DATA FROM THE RESULTS OF A QUERY

The results returned from queries consist of character vectors. Typically, these are XML documents. To be useful, they need to be converted into R data structures that can be manipulated.

There are two methods for doing this. The first is to apply functions that extract out specific portions of the data, usually specified by a path. The second possibility is to convert portions of the XML to a data frame. This involves using a path to specify a specific field that might be repeated (e.g., Detection in a set of detections). There are caveats to converting to data frames, and in some cases this may not be possible. As an example, if one is interested in retrieving detections, and some of the detections have a measured received level and others do not, this would create problems as all rows of the data frame need to have the same data elements. Note that the data frames are constructed from query results, so querying the same set of detections without returning the received level would allow the construction.

3.2.1 Extracting data from specific fields

This is described in the section on XML2 (4.1, p. 14).

3.2.2 Converting XML returned from a query to a data frame.

For the impatient:

1. If you do not know what your query result will look like, execute the query and inspect the result (cat the result or save it to a file if it is large).
2. Determine what portion of the subtree you want, e.g. all Detection fields (XML elements). Note the fields from the top of the XML down to your field. The list of these elements, which must start with a forward slash (/) and have forward slash separators between each field will be called the path.
   For example: /Result/Record/Detection.
3. Rerun the query with the stylesheet argument. It should be a named vector with the following fields:
   - operation="flatten" – Only operation currently available, makes everything beneath the path the same level. If there is a child field that repeats, you cannot create a flat structure.
   - path="/Result/Record/Detection" – Path to an field that may repeated. Each row of the dataframe will consist of fields and values that are children of the data specified by the path argument.
   - parents=N – Optional value, indicates how many parents should included in the field names. This is only needed when there are fields with the same name that are children of multiple fields. For example, if a deployment query returns DeploymentDetails and RecoveryDetails, these both have several fields with the same name such as Longitude, Latitude. Parents=1 would rewrite the fields as DeploymentDetails.Longitude, RecoveryDetails.Longitude, etc. If more levels are needed the value of parents can be increased.

   Example parameter:

stylesheet=list(operation="flatten", path="/Result/Record/Detections/Detection", parents=1))

4. Use the Tethys$xml2df method to convert your XML to a dataframe. It requires the XML document to transform, and a path to the section of the result to be transformed to a dataframe.

Example that assumes t contains a Tethys object:

```
# ResponsibleParty/organizationName appears in both DeploymentDetails
# and RecoveryDetails.  Hence, we need 2 parents to make organizationName unique
dep = t$getDeployments('Project = "Aleut" return Id, DeploymentDetails, Recovery
   Details', stylesheet=list(operation="flatten", path="/Result/Record", parents
   =2), dataframes=FALSE)

# Convert each Record to a row in the dataframe
# We convert to the appropriate type when possible.
df = t$xml2df(dep$xml, "/Result/Record")
```

CAVEATS: xml2f is designed for extracting a single level of the XML tree. Data that are nested deeper than the xpath specification will be collapsed into a single value. The parents argument can mitigate against this as in the example usage above.

For those who want to understand…

Dataframes are flat structures, meaning that they are tabular without any nested structures. Many times, this does not present significant problems. The default detection query returns the Start, End, and SpeciesId for each detection. As long as all of the detections returned have this, conversion is easy[4].

```
det <- t$getDetections('SpeciesId = "Lagenorhynchus obliquidens"', dataframes=FAL
   SE)
```

The det$xml field contains

```
<Result>
   <Record>
         … many other detections …
         <Detection>
            <Start>2013-03-16T13:00:52.500Z</Start>
            <End>2013-03-16T13:25:37.500Z</End>
            <SpeciesId>Lagenorhynchus obliquidens</SpeciesId>
         </Detection>
         <Detection>
            <Start>2013-03-17T03:22:02.500Z</Start>
            <End>2013-03-17T04:32:37.500Z</End>
            <SpeciesId>Lagenorhynchus obliquidens</SpeciesId>
         </Detection>
      </Detections>
   </Record>
</Result>
```

[4] This query was executed on a different database than the demonstration database that is included with the Tethys distribution, and will not yield the same results.

Consequently, we need to select a portion of the data that repeats in a regular way. By specifying a path to Detection, we can do so and create a data frame with Start, end, and SpeciesId columns.

```
df = t$xml2df(det$xml, path="/Result/Record/Detections/Detection")

df
                  Start                 End                   SpeciesId
1   2011-01-29 23:45:22 2011-01-29 23:50:42 Lagenorhynchus obliquidens
2   2011-02-23 06:41:27 2011-02-23 06:45:02 Lagenorhynchus obliquidens
3   2011-03-01 05:56:52 2011-03-01 06:14:57 Lagenorhynchus obliquidens
4   2011-03-02 02:51:17 2011-03-02 03:40:27 Lagenorhynchus obliquidens
5   2011-03-02 09:29:17 2011-03-02 10:07:57 Lagenorhynchus obliquidens
6   2011-03-03 04:02:52 2011-03-03 04:38:02 Lagenorhynchus obliquidens
7   2011-03-03 10:58:27 2011-03-03 11:14:22 Lagenorhynchus obliquidens
8   2011-03-04 04:41:22 2011-03-04 05:14:37 Lagenorhynchus obliquidens
9   2011-03-05 03:58:27 2011-03-05 04:15:42 Lagenorhynchus obliquidens
10  2011-03-14 03:14:27 2011-03-14 03:37:17 Lagenorhynchus obliquidens
…
```

Currently, the xml2df does not support cases where elements are not present in every row. If we had requested that Parameters/Subtype be returned:

```
# Similar to above request, but returns a Subtype field for detections
# that have one.
xml <- t$getDetections('SpeciesId = "Lagenorhynchus obliquidens" return Detectio
   n/Start, Detection/End, Detection/SpeciesId, Parameters/Subtype')
# As only some of the detections have a Subtype, this will generate an error.
df = t$xml2df(xml, path="/Result/Record/Detection")
Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,  :
  arguments imply differing number of rows: 3384, 1989
```

Things become a little more complicated in examples where the items we want to use to form rows contain nested values. Consider this query for deployments in the southern hemisphere:

```
dep <- t$getDeployments('DeploymentDetails/Latitude < 0 return Id, DeploymentDet
   ails, RecoveryDetails', dataframes=FALSE)
```

which returns the following XML:

```
<Result>
   <Record>
      <Id>ANTARC01-EI</Id>
      <DeploymentDetails>
         <Longitude>304.046033</Longitude>
         <Latitude>-60.886900</Latitude>
         <ElevationInstrument_m>-762</ElevationInstrument_m>
         <TimeStamp>2014-03-05T00:00:00Z</TimeStamp>
         <AudioTimeStamp>2014-03-05T00:00:00Z</AudioTimeStamp>
         <Vessel>SB15-Tango</Vessel>
         <ResponsibleParty>
            <organizationName>Scripps Whale Acoustics Lab</organizationName>
         </ResponsibleParty>
      </DeploymentDetails>
      <RecoveryDetails>
         <Longitude>304.046033</Longitude>
         <Latitude>-60.886900</Latitude>
         <TimeStamp>2014-07-14T12:03:50Z</TimeStamp>
         <AudioTimeStamp>2014-07-14T12:03:50Z</AudioTimeStamp>
         <Vessel>SB15 Tango</Vessel>
         <ResponsibleParty>
            <organizationName>Scripps Whale Acoustics Lab</organizationName>
         </ResponsibleParty>
      </RecoveryDetails>
   </Record>
   <Record>
      <Id>ANTARC_SSI_01</Id>
      <DeploymentDetails>
         …
      <RecoveryDetails>
         …
      </RecoveryDetails>
   </Record>
   …
</Result>
```

We can use the a flatten stylesheet specification in the query to make everything under /Result/Record be part of the same row. However, there are many fields that have different parents. For example, Latitude occurs in both DeploymentDetails and RecoveryDetails. Adding one parent to every nested field would help some as it would let us distinguish our Laittudes, but it would not let us distinguish organizationName as organizationName has ResponsibleParty as the parent in both the deployment and recovery details.

By specifying that we want /Result/Record to form the rows of our dataframe and that we want to keep the names of up to two parents, we can generate XML that is more amenable to a constructing a dataframe.

```
dep <- t$getDeployments('DeploymentDetails/Latitude < 0 return Id, DeploymentDetails, RecoveryDetails', stylesheet=c(operation="flatten", path="/Result/Record/", parents=2), dataframes=FALSE)
```

The flattened records in dep$xml look like this:


```
<Result>
   <Record>
      <Id>ANTARC01-EI</Id>
      <DeploymentDetails.Longitude>304.046033</DeploymentDetails.Longitude>
      <DeploymentDetails.Latitude>-60.886900</DeploymentDetails.Latitude>
      <DeploymentDetails.ElevationInstrument_m>-762</DeploymentDetails.ElevationInstrument_m>
      <DeploymentDetails.TimeStamp>2014-03-05T00:00:00Z</DeploymentDetails.TimeStamp>
      <DeploymentDetails.AudioTimeStamp>2014-03-05T00:00:00Z</DeploymentDetails.AudioTimeStamp>
      <DeploymentDetails.Vessel>SB15-Tango</DeploymentDetails.Vessel>
      <DeploymentDetails.ResponsibleParty.organizationName>Scripps Whale Acoustics
Lab</DeploymentDetails.ResponsibleParty.organizationName>
      <RecoveryDetails.Longitude>304.046033</RecoveryDetails.Longitude>
      <RecoveryDetails.Latitude>-60.886900</RecoveryDetails.Latitude>
      <RecoveryDetails.TimeStamp>2014-07-14T12:03:50Z</RecoveryDetails.TimeStamp>
      <RecoveryDetails.AudioTimeStamp>2014-07-14T12:03:50Z</RecoveryDetails.AudioTimeStamp>
      <RecoveryDetails.Vessel>SB15 Tango</RecoveryDetails.Vessel>
      <RecoveryDetails.ResponsibleParty.organizationName>Scripps Whale Acoustics
Lab</RecoveryDetails.ResponsibleParty.organizationName>
   </Record>
   …
</Result>
```


and we can now construct a dataframe which is not shown here as it is a very wide table.

```
df = t$xml2df(dep$xml, "/Result/Record")
```

4 Working with XML

The xml2 package is the most efficient library. The data.tree offers relatively easy navigation, but it is not efficient for large xml data sets and we are also currently seeing issues with retrieving data from the nodes and cannot recommend using it for use with the R client at this time. For now, we recommend using xml2.

4.1 The XML/xml2 Libraries

There exist two packages for parsing XML in R. XML is the older package and requires users to manage memory. The xml2 package is a wrapper for a C library and is generally preferred. While either one can be used, we recommend xml2 and will provide xml2 examples.

Read in an XML file. For example, if the output of an effort query was saved as XML file effort.xml, one might read the XML as follows:

```
# Here, we read XML from a file that is in the current
# working directory.  Instead of having a string with
# a filename, we could also have a string returned from the
# R Tethys client and it would also be parsed.
> xml <- read_xml("effort.xml")
```

This creates a set of lists:

```
> xml
```

```
[1] <Record>\n  <Deployment>\n  <Id>CCES10_PTA</Id>\n     <Project>CCES</Project>\n     <Site>PTA</Site>\n ...
 [2] <Record>\n  <Deployment>\n <Id>CCES12_MOB</Id>\n     <Project>CCES</Project>\n     <Site>MOB</Site>\n ...
 [3] <Record>\n  <Deployment>\n <Id>CCES13_CHI</Id>\n     <Project>CCES</Project>\n     <Site>CHI</Site>\n ...
 [4] <Record>\n  <Deployment>\n <Id>CCES14_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
 [5] <Record>\n  <Deployment>\n <Id>CCES16_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
 [6] <Record>\n  <Deployment>\n <Id>CCES17_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
 [7] <Record>\n  <Deployment>\n <Id>CCES18_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
 [8] <Record>\n  <Deployment>\n <Id>CCES19_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
 [9] <Record>\n  <Deployment>\n <Id>CCES20_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
[10] <Record>\n  <Deployment>\n <Id>CCES21_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
[11] <Record>\n  <Deployment>\n <Id>CCES22_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
[12] <Record>\n  <Deployment>\n <Id>CCES23_BCN</Id>\n     <Project>CCES</Project>\n     <Site>BCN</Site>\n ...
```

XML supports namespaces, a wrapping mechanism that allows us to distinguish the same name when it is defined in different contexts. While Tethys uses namespaces, the RClient removes these by default.

If you overrode the default, they will need to be stripped manually.

```
xml_ns_strip(xml)  # IMPORTANT:  Examples will not function with namespaces
```

To see it as XML, use the XML package:

```
> xml_p <- xmlParse(xml)
> xml_p # display it
```

```
<Result xmlns="http://tethys.sdsu.edu/schema/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Record>
    <Deployment>
      <Id>CCES10_PTA</Id>
      <Project>CCES</Project>
      <Site>PTA</Site>
      <DeploymentId>10</DeploymentId>
      <DeploymentDetails>
        <Longitude>234.9416</Longitude>
        <Latitude>36.7607</Latitude>
        <TimeStamp>2018-08-22T02:11:00Z</TimeStamp>
        <AudioTimeStamp>2018-08-22T02:24:55Z</AudioTimeStamp>
        <Vessel>R/V Ruben Lasker</Vessel>
      </DeploymentDetails>
    </Deployment>
    <Detections>
      <Id>CCES010_BW_Detections</Id>
      <Start>2018-08-22T02:24:00Z</Start>
      <End>2018-10-21T23:48:00Z</End>
      <Kind>
        <SpeciesId>Bb1</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
```

```
      <Kind>
        <SpeciesId>Zc</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
      <Kind>
        <SpeciesId Group="BWC">BWC</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
      … other species effort omitted for brevity …
      <Algorithm>
        <Method>Analyst detections</Method>
        <Software>Pamguard</Software>
        <Version>2.00.16</Version>
        <Parameters>
          <Click_Viewer_plot_time_m>2</Click_Viewer_plot_time_m>
        </Parameters>
      </Algorithm>
    </Detections>
  </Record>
  ... next Record ...
</Result>
```

Data are queried through XPath expressions, a list of names showing the path through the nesting. For example, if we wanted to know which deployments were in this set of detection effort, we could use:

```
> xml_find_all(xml, "./Record/Deployment/Id")
{xml_nodeset (12)}
 [1] <Id>CCES10_PTA</Id>
 [2] <Id>CCES12_MOB</Id>
 [3] <Id>CCES13_CHI</Id>
 [4] <Id>CCES14_BCN</Id>
 [5] <Id>CCES16_BCN</Id>
 [6] <Id>CCES17_BCN</Id>
 [7] <Id>CCES18_BCN</Id>
 [8] <Id>CCES19_BCN</Id>
 [9] <Id>CCES20_BCN</Id>
[10] <Id>CCES21_BCN</Id>
[11] <Id>CCES22_BCN</Id>
[12] <Id>CCES23_BCN</Id>
```

where Record/Deployment/Id specifies the path to the deployment identifiers. If we wanted to extract the text from these nodes, we could wrap the call in xml_text:

```
> xml_text(xml_find_all(xml, "./Record/Deployment/Id"))
 [1] "CCES10_PTA" "CCES12_MOB" "CCES13_CHI" "CCES14_BCN" "CCES16_BCN" "CCES17_BC
N" "CCES18_BCN" "CCES19_BCN"
 [9] "CCES20_BCN" "CCES21_BCN" "CCES22_BCN" "CCES23_BCN"
```

Other useful functions are xml_integer() and xml_double for extracting numeric values. Additional information may be found in the documentation for the xml2 library as well as descriptions of XPath. There are numerous resources for learning XPath, Walmsley (2006) covers XPath well in her introduction to the XQuery language. Many examples on the web tend to use the wildcard search operator (//) which let one search for an XML element at any level (e.g., .//Id instead of ./Record/Deployment/Id). These types of searches are inefficient and should be avoided except on very small data.

The Tethys class also provides some convenience functions. When the XML has the same fields, the Tethys XML to dataframe method (xml2df) can be very useful. See section 3.2.2 (page 10) for information.

References

Kay, M. (2008). XSLT 2.0 and XPath 2.0 : programmer's reference. Indianapolis, IN, Wiley Pub.

Walmsley, P. (2006). XQuery. Farnham, UK, O'Reilly.