NCEAS Data Team Training

Jeanette Clark, Jesse Goldstein, Dominic Mullen, Bryce Mecum, Matt Jones, Peter Slaughter

2018-03-19

Contents

Chapter 1

Welcome to NCEAS!

1.1 First day to-dos

- Get a tour of the office from Ginger, Jeanette, or Jesse
- Fill out required paperwork from Michelle
- Have Ana take your picture (room 309)
- Set up the remainder of your accounts

1.2 Account information

- LDAP - NCEAS - Jeanette or Jesse should have set this up prior to your start date to help get other accounts set up by the first day. This account and password control your access to:
  - arcticdata RT queue
  - GitHub - arctic-data and sasap-data
- Datateam server - follow instructions in email from Nick at NCEAS to reset your datateam password in the terminal
- ORCiD - create an account
- NCEAS Slack - get an invite from slack.nceas.ucsb.edu
- Arctic Data Center Team - after creation of ORCiD and sign-in to both arcticdata.io and test.arcticdata.io, request an add from Chris on Slack
- Schedule - fill out anticipated quarterly schedule

1.3 NCEAS Events

NCEAS hosts a number of events that you are encouraged to attend. Keep an eye on your email but the weekly events are :

- Roundtable
  - weekly presentation and discussion of research by a visiting or local scientist
  - Wednesdays at 12:15 in the lounge
- Coffee Klatch
  - coffee, socializing, and news updates for NCEAS
  - Tuesdays at 10:30 in the lounge
- Salad Potluck
  - potluck salad and socializing, bring a topping or side to share!

– second Tuesday of the month, 12:15 in the lounge

Chapter 2

Introduction to open science

These materials are meant to introduce you to the principles of open science, effective data management, and data archival with the DataONE data repository.

New data team members should complete Exercise 1 as part of training.

2.1 Open science background reading

Read the content on the Arctic Data Center (ADC) webpage to learn more about data submission, preservation, and the history of the ADC. We encourage you to follow the links within these pages to gain a deeper understanding.

- about
- submission
- preservation
- history

2.2 Effective data management

Read Matt Jones' paper on effective data management to learn how we will be organizing datasets prior to archival.

2.3 Using DataONE

The **Data Observation Network for Earth** (DataONE) is a community driven project providing access to data across multiple member repositories, supporting enhanced search and discovery of Earth and environmental data.

Watch the first 38 minutes of this video explaining how DataONE works. This video is pretty technical, and you may not understand it all at first. Please feel free to ask Jesse or Jeanette questions.

2.4 Working on a remote server

All of the work that we do at NCEAS is done on our remote server, datateam.nceas.ucsb.edu. If you have never worked on a remote server before, you can think of it like working on a different computer via the internet.

We access RStudio for our server through this link. To transfer files on and off of the server, you'll need to use either bash commands in the terminal, or an FTP client. We use a client called Cyberduck.

2.4.1 Cyberduck instructions

To use Cyberduck to transfer local files onto the Datateam server:

1) Open Cyberduck.
2) Check that you have the latest version (Cyberduck -> Check for Update…). If not, download and install the latest (you may need Jesse or Jeanette to enter a pw).
3) Click "Open Connection".
4) From the drop-down, choose "SFTP (Secure File Transfer Protocol)".
5) Enter "datateam.nceas.ucsb.edu" for Server.
6) Enter your username and password.
7) Connect.

From here, you can drag and drop files to and from the server.

2.4.2 A note on paths

On the servers, paths to files in your folder always start with `/home/yourusername/...`.

When you write scripts, try to avoid writing relative paths (which rely on what you have set your working directory to) as much as possible. Instead, write out the entire path as shown above, so that if another data team member needs to run your script, it is not dependent on a working directory.

2.4.3 A note on scripts

To make it easy to follow the flow of your work, it may help to number your scripts. For example, `01_clean_data.R`, `02_edit_EML.R`, `03_publish.R`.

2.4.4 Exercise 1

- Download the csv of Table 1 from this paper.
- Reformat the table using R under the guidelines in the journal article on effective data management.
  - If you need an R refresher, take as much time as you need to go over the data carpentry guide..
  - You may also find the data carpentry lesson on `dplyr` and OHI's data wrangling chapters helpful.
- Go to test.arcticdata.io and submit your reformatted file with appropriate metadata that you derive from the text of the paper:
  - list yourself as the first 'Creator' so your test submission can easily be found,
  - for the purposes of this training exercise, not every single author needs to be listed with full contact details, listing the first two authors is fine,
  - directly copying and pasting sections from the paper (abstract, methods, etc.) is also fine,
  - attributes (column names) should be defined, including correct units and missing value codes.

Chapter 3

Using arcticdatautils

New data team members should complete exercises two and three as part of training.

3.1 Uploading packages using R

We will be using the `arcticdatautils` package and the `dataone` package to connect to the NSF Arctic Data Center (ADC) data repository and push and pull edits. To identify yourself as an admin you will need to pass a 'token' into R. Do this by signing in to the ADC with your ORCiD, hovering over your name and clicking on "My profile", then navigating to "Setings" and "Authentication Token", copying the "Token for DataONE R", and pasting and running it in your R console.

This token is your identity on these sites, please treat it as you would a password (ie. don't paste into scripts that will be shared). The easiest way to do this is to always run the token in the *console*. There's no need to keep it in your script since it's temporary anyway. Sometimes you'll see a placeholder in scripts to remind users to get their token, such as:

```
options(dataone_test_token = "...")
```

Next, please be sure these packages are loaded:

```
library(devtools)
library(dataone)
library(remotes)
library(arcticdatautils)
```

Install any missing packages using `install.packages('PACKAGE_NAME')` in your R console. For `arcticdatautils`, you'll have to run `remotes::install_github("nceas/arcticdatautils")` because it's not on CRAN.

3.1.1 Set environment

Once we have our libraries loaded, we need to tell R with which repo we want to work. If we are working on the production site, you can set the coordinating node (`cn`) and member node (`mn`) this way:

```
cn <- CNode('PROD')
mn <- getMNode(cn,'urn:node:ARCTIC')
```

If we are working in the test environment, we set it this way:

```
cn <- CNode('STAGING')
mn <- getMNode(cn,'urn:node:mnTestARCTIC')
```

A note on nodes - be very careful about what you publish on the production node (PROD, or arcticdata.io). This node should NEVER be used to publish test or training datasets. While going through training exercises, you should be using the test environment (STAGING, or test.articdata.io). The CN is DataONE.

3.1.2 What is in a package?

A data package generally consists of at least 3 "objects" or files:

- metadata
- data
- resource map

The **metadata** file is in XML format, and has the extension `.xml` (extensible markup language). We often refer to this file as the EML (Ecological Metadata Language), which is the metadata standard that it uses.

Other objects are the **data** files themselves. Most commonly these are data tables (`.csv`), but they can also be audio files, netCDF files, plain text files, pdf documents, image files, etc.

The final object is the **resource map**. This object is a text file that defines the relationships between all of the other objects in the data package. It says things like "this metadata file describes this data file," and is critical to making a data package render correctly on the website with the metadata file and all of the data files together in the correct place. Fortunately, we rarely, if ever, have to actually look at the contents of resource maps; they are generated for us using `arcticdatautils`.

Each of these objects are also associated with **system metadata**, which contains information such as file name, file type, and version. Don't worry too much about it now; we'll go into more detail in the later chapters.

3.1.3 About identifiers

Each object (metadata files, data files, resource maps) on the ADC or KNB (another repo) has a unique identifier, also sometimes called a "PID" (persistent identifier). When you look at the landing page for a dataset, for example here, you can find the resource map identifier in the URL (`resource_map_doi:10.18739/A2836Z`), the metadata identifer in the "General > Identifier" section of the metadata record (`doi:10.18739/A2836Z`), and the data identifier by clicking the "more info" link next to the data object, and looking at the "Online Distribution Info" section (`arctic-data.9546.1`).

Different versions of a package are linked together by what we call the "version chain" or "obsolescence chain". Making an update to a data package, such as replacing a data file, changing a metadata record, etc, will result in a new identifier for the new version of the updated object.

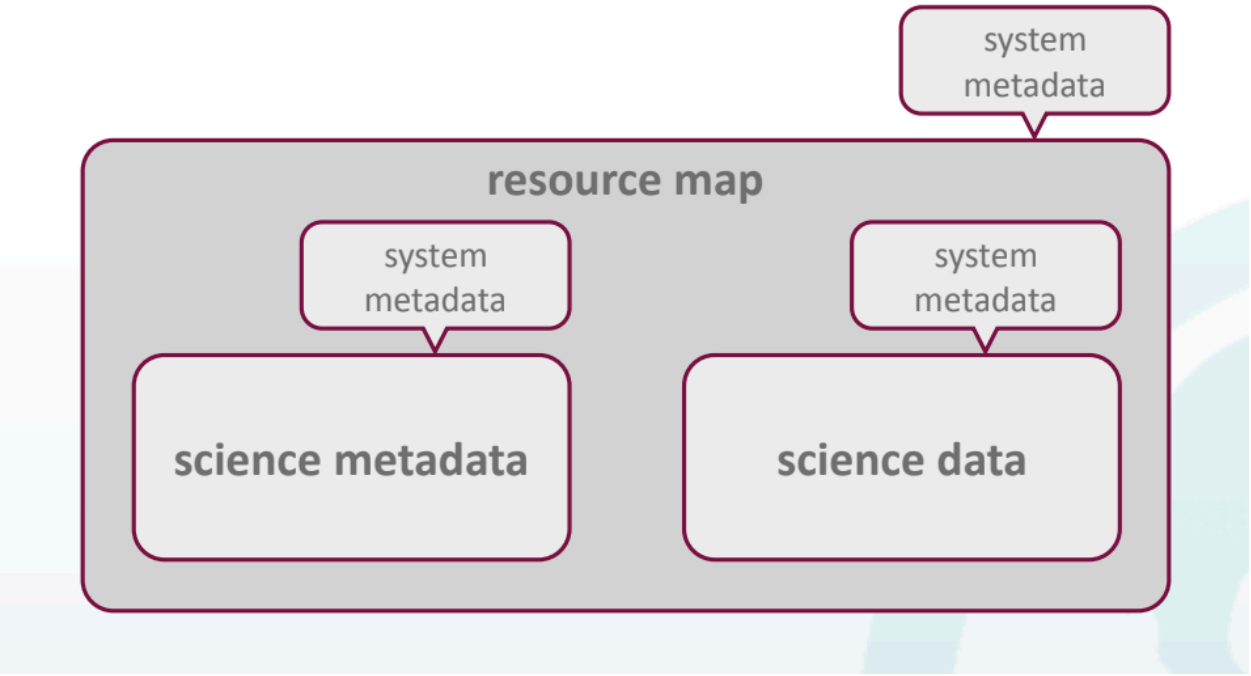

system
metadata
resource map
system
metadata
system
metadata
science metadata
science data

Figure 3.1:

Chapter 4

System metadata

New data team members should complete the exercise in this section as part of training.

4.1 Introduction

Every object on the ADC (or on the KNB (the 'Knowledge Network for Biocomplexity')) has "system metadata". An object's system metadata has information about the file itself, such as the name of the file, the format, who the rights holder is, and what the access policy is (amongst other things). Sometimes we will need to edit system metadata in order to make sure that things on the webpage display correctly, or to ensure a file downloads from the website with the correct file name and extension.

Although the majority of system metadata changes that need to be made are done with the functions in `arcticdatautils`, sometimes we need to change aspects of the system metadata (or 'sysmeta' for short) outside of those functions. This markdown explains how to do this using the functions `getSystemMetadata` and `updateSystemMetadata`.

4.2 Editing sysmeta

First we need to load in some packages, and set up our environment. Note that typically, if you are editing sysmeta, you will need to set a token just like if you were updating a package.

```
library(arcticdatautils)
library(dataone)

cn <- CNode('STAGING')
mn <- getMNode(cn,'urn:node:mnTestARCTIC') #Loading in the test environment for this example
```

Next, we input the `mn` instance and PID of interest into the `getSystemMetadata` function. Note that the PID should be of the object of interest, not just the package of interest. Each individual object (metadata file, resource map, data file) will have its own system metadata.

```
pid <- 'urn:uuid:9a1b02a8-713e-4682-aafb-95c854a4c24a'
sysmeta <- getSystemMetadata(mn, pid)
```

This returns an S4 object (more on these objects in Chapter 5), called `sysmeta`. The sysmeta object has the following "slots":
* serialVersion

* identifier
* formatId
* size
* checksum
* checksumAlgorithm
* submitter
* rightsHolder
* accesPolicy
* replicationAllowed
* numberReplicas
* preferredNodes
* blockedNodes
* obsoletes
* obsoletedBy
* archived
* dateUploaded
* dateSysMetadataModified
* originMemberNode
* authoritativeMemberNode
* seriesId
* mediaType
* fileName
* mediaTypeProperty

You can view and edit slots using the `@` functionality.

```
sysmeta@fileName [1] "WELTS_flatfile.csv"
```

If you want to change a slot, you can simply do the following:

```
sysmeta@fileName <- 'AlaskaWells.csv'
```

Note that some slots cannot be changed by simple text replace (particularly the `accessPolicy`). There are various helper functions for changing the `accessPolicy` and `rightsHolder` such as `addAccessRule` (which takes the sysmeta as an input) or `set_access`, which only requires a PID. In general, you most frequently need to use `getSystemMetadata` to change either the `formatId` or `fileName` slots.

After you have changed the necessary slot, you can update the system metadata using the `updateSystemMetadata` function, which takes the `mn` instance, your PID of interest, and the edited sysmeta object as arguments.

```
updateSystemMetadata(mn, pid, sysmeta)
```

4.2.1 Exercise

- Read the system metadata in from the data file you uploaded in the previous chapter.
- Check to make sure the `fileName` and `formatId` are set correctly.
- Update the system metadata if necessary.

4.3 Editing sysmeta on lots of files

If you have a lot of files that all need a similar change, you can use a for loop or the apply functions to efficiently make all of your changes. Say you need to change all of the `formatId`s of data objects in a package so that they are `text/csv`. You could do this:

```
PID <- 'some_pid'
ids <- get_package(mn, PID) #get all pids in a package
data_pids <- unlist(ids$data)
for (i in 1:length(data_pids)){    #run for loop over all of the data ids
  sysmeta <- getSystemMetadata(mn, data_pids[i])
  sysmeta@formatId <- "text/csv"
  updateSystemMetadata(mn, data_pids[i], sysmeta)
}
```

4.4 Identifiers and sysmeta

Importantly, changing the system metadata does NOT necessitate a change in the PID of an object. This is because changes to the system metadata do not change the object itself, they are only changing the description of the object (although ideally the system metadata is accurate when an object is first published).

Chapter 5

Editing EML in R

New data team members should complete the exercise in this section as part of training.

5.1 Introduction

This chapter is a practical tutorial for using the EML package to read, edit, write, and validate EML documents. Much of this information can also be found in the vignettes for the EML package.

First, load in some packages.

```
library(EML)
library(XML)
library(digest)
library(arcticdatautils)
```

Set a path to a local copy of the EML that you would like to edit, and read the EML into R.

```
path1 <- 'data/glacier_metadata.xml'
eml <- read_eml(path1)
```

When using the EML R package, you will usually be working with objects of class `eml`. For example, we can find out what type of object we just created when we ran the `read_eml` function:

```
class(eml)
```

```
## [1] "eml"
## attr(,"package")
## [1] "EML"
```

This `eml` object represents a complete EML document (as in: the .xml file) and all of the information inside the `.xml` file can be viewed and edited with the EML package.

The `eml` object has what are called "slots", each slot representing an element of the EML document such as "Title" or "Creator" and you will use these "slots" when working with your `eml` object. (Sometimes we call the slots "elements" or "tags".) You can find out what slots you have available with the `slotNames` function:

```
sort(slotNames(eml))
```

```
##  [1] ".Data"              "access"             "additionalMetadata"
##  [4] "citation"           "dataset"            "lang"
##  [7] "packageId"          "protocol"           "schemaLocation"
## [10] "scope"              "slot_order"         "software"
```

```
## [13] "system"
```

You can access information in one of these slots by adding an `@` symbol at the end of the variable you want to view the slots of and hitting TAB (e.g., `eml@<TAB>`):

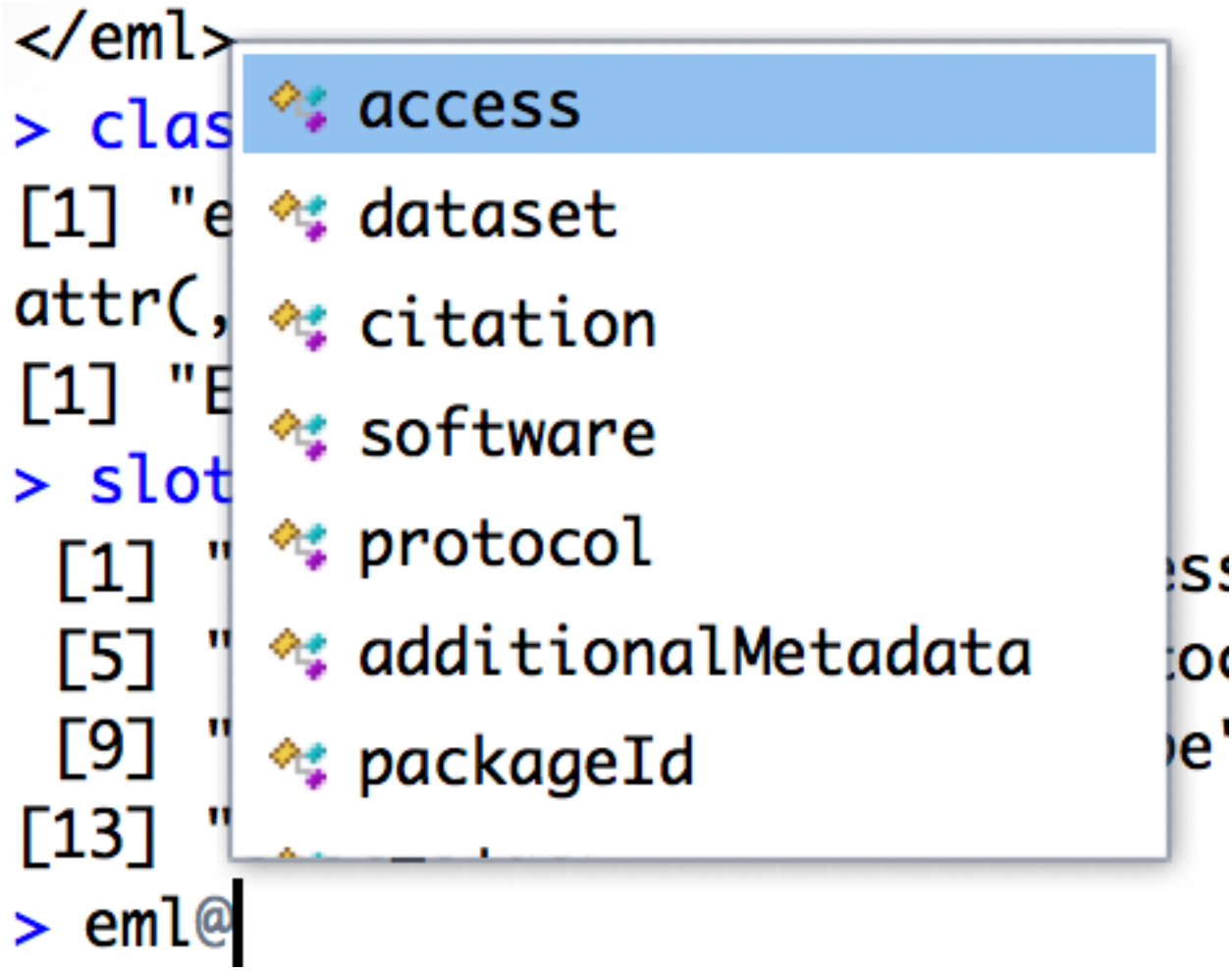


Figure 5.1: RStudio Autocompletion

You will notice that the list of slots on the `eml` object doesn't include common EML things such as the dataset's title or the creators. This is because EML (the XML) and the EML R package are both hierarchical. The (incomplete) EML XML for a dataset with a title would look something like this:

```
<eml>
  <dataset>
    <title>My title goes here...</title>
  </dataset>
</eml
```

You can see that the `title` is nested inside the `dataset` and the `dataset` is nested inside the root `eml` element of the document.

Because the `title` is nested inside the `dataset` element, it will be a slot of `eml@dataset` instead of a slot of the main `eml` object:

```
sort(slotNames(eml@dataset))
```

```
##  [1] ".Data"               "abstract"            "additionalInfo"
##  [4] "alternateIdentifier" "associatedParty"     "contact"
##  [7] "coverage"            "creator"             "dataTable"
## [10] "distribution"        "id"                  "intellectualRights"
## [13] "keywordSet"          "lang"                "language"
## [16] "maintenance"         "metadataProvider"    "methods"
## [19] "otherEntity"         "project"             "pubDate"
## [22] "publisher"           "pubPlace"            "purpose"
## [25] "references"          "schemaLocation"      "scope"
## [28] "series"              "shortName"           "slot_order"
## [31] "spatialRaster"       "spatialVector"       "storedProcedure"
## [34] "system"              "title"               "view"
```

Remember, you can just type `eml@dataset@` into your console and hit to see this list of slot names. The `slotNames` function is used for demonstration purposes here.

Slots can be nested in each other and are all based on the EML schema.

Typing in the name of your `eml` object (in this case, `eml`, the name of the result of `read_eml`) and hitting `<RETURN>` in the console will print the entire EML onto the screen:

```
eml
```

Similarly, you can also view different elements of the EML by drilling down into the structure using the `@` functionality. This would just print the dataset portion of the EML:

```
eml@dataset
```

Going even deeper, this command will print the `title` element, which is within the `dataset` element, of the EML document:

```
eml@dataset@title
```

```
## An object of class "ListOftitle"
## [[1]]
## <title>Subglacial conduit fluid dynamics simulation, Svalbard, Norway</title>
```

Notice than when the last line prints, it doesn't just print a string - it returns "An object of class `ListOftitle`." The EML is composed of different classes of objects that make up these slots. If a slot can have multiple items, it may require an input of a list of a class. You can continue digging down into the EML using subsetting techniques for the S4 object class, with syntax that looks like this:

```
eml@dataset@title@.Data[[1]]@.Data
```

```
## [1] "Subglacial conduit fluid dynamics simulation, Svalbard, Norway"
```

Notice that this actually prints a character string. If you want to change the title, you could just assign this a new character string value, like so:

```
eml@dataset@title@.Data[[1]]@.Data <- 'This is my new title'
```

However, this isn't the best method to edit the EML unless you are an expert both in S4 objects and in the EML schema, since the nesting and lists of elements can get very complex. Instead, the EML package is used to create new objects of particular classes to put into the document, using the function `new`.

The `new` function has a format that looks like `newobject <- new('class', arguments...)`. It can be a difficult function to use because "class" has to be set to a specific name and the argument structure will vary depending on what the slots are in that class. Because the function is so general, the `?new` help is not very helpful. In the EML package, a good guess to what the name of the class will be is a slot name (such as "title"). You can explore what slots are available within an object class by creating a new, empty object like this:

```
test_title <- new('title')
```

and using the R autocomplete functionality on `test_title@`. Note that the slots `lang`, `slot_order`, `schemaLocation`, and `.Data` will always be present, and are set by the EML package automatically according to the required EML and XML schema. The other options will tell you what the arguments following the class name should be. In the case of the title class, the only option is `value`.

So, to change the title, you might at first try do something like this:

```
eml@dataset@title<- new('title', .Data = 'This is my new title')
```

Unfortunately it returns an error. The error reads:

```
Error in (function (cl, name, valueClass)  :assignment of an object of class "title" is
not valid for @'title' in an object of class "dataset"  ; is(value, "ListOftitle") is not
TRUE
```

The error says it cannot assign an object of class "title" to `eml@dataset@title`, and that there is not a value for `ListOftitle`. So this means that you have to assign to `@title` an object of class `ListOftitle`, which is composed of objects of class `title`. Even though `ListOftitle` is it's own kind of class, we can avoid using the `new` function again here by simply putting the `title` object in a vector using the `c()` function.

```
title <- new('title', .Data = 'This is my new title')
eml@dataset@title <- c(title)
```

This seems kind of cumbersome, creating first a new title object, and then a list of title, especially since in most cases an EML will only have one title. However, say for some reason you need two titles - you can then do this:

```
title_second <- new('title', .Data = 'This dataset has two titles')
eml@dataset@title <- c(title, title_second)
eml@dataset@title
```

```
## An object of class "ListOftitle"
## [[1]]
## <title>This is my new title</title>
##
## [[2]]
## <title>This dataset has two titles</title>
```

This functionality will prove very useful with other elements (such as `dataTable`), where there is usually more than one of the same element.

5.2 Making edits

5.2.1 Setting attributes

Since attribute information has to be added to the metadata, we'll cover attributes first.

5.2.1.1 Building the attribute table

First you need to generate a dataframe with attribute information. This dataframe has rows that are attributes, and the following columns:

- **attributeName**: The name of the attribute as listed in the csv. Required. eg: "c_temp"

- **attributeLabel**: A descriptive label that can be used to display the name of an attribute. It is not constrained by system limitations on length or special characters. Optional. eg: "Temperature (Celsius)"
- **attributeDefinition**: Longer description of the attribute, including the required context for interpreting the attributeName. Required. eg: "The nearshore water temperature in the upper intertidal zone, measured in degrees celsius."
- **measurementScale**: One of: nominal, ordinal, dateTime, ratio, interval. Required.
  - *nominal*: unordered categories or text. eg: (Male, Female) or (Yukon River, Kuskokwim River)
  - *ordinal*: ordered categories. eg: Low, Medium, High
  - *dateTime*: date or time values from the Gregorian calendar. eg: 01-01-2001
  - *ratio*: measurement scale with a meaningful zero point in nature. Ratios are proportional to the measured variable. e.g.: 0 Kelvin represents a complete absence of heat. 200 Kelvin is half as hot as 400 Kelvin. 1.2 meters per second is twice as fast as 0.6 meters per second.
  - *interval*: values from a scale with equidistant points, where the zero point is arbitrary. This is usually reserved for degrees Celsius or Fahrenheit, or latitude and longitude coordinates, or any other human-constructed scale. e.g.: there is still heat at 0° Celsius; 12° Celsius is NOT half as hot as 24° Celsius
- **domain**: One of: textDomain, enumeratedDomain, numericDomain, dateTimeDomain. Required.
  - *textDomain*: text that is free-form, or matches a pattern
  - *enumeratedDomain*: text that belongs to a defined list of codes and definitions. eg: CASC = Cascade Lake, HEAR = Heart Lake
  - *dateTimeDomain*: dateTime attributes
  - *numericDomain*: attributes that are numbers (either ratio or interval)
- **formatString**: Required for dateTimeDomain, NA otherwise. Format string for dates, eg "MM/DD/YYYY".
- **definition**: Required for textDomain, NA otherwise. Definition for attributes that are a character string, matches attribute definition in most cases.
- **unit**: Required for numericDomain, NA otherwise. Unit string. If the unit is not a standard unit, a warning will appear when you create the attribute list, saying that it has been forced into a custom unit. Use caution here to make sure the unit really needs to be a custom unit. A list of standard units can be found here: https://knb.ecoinformatics.org/#external//emlparser/docs/eml-2.1.1/./eml-unitTypeDefinitions.html#StandardUnitDictionary
- **numberType**: Required for numericDomain, NA otherwise. Options are "real", "natural", "whole", "integer".
  - *real*: positive and negative fractions and non fractions (…-1,-0.25,0,0.25,1…)
  - *natural*: non-zero positive counting numbers (1,2,3…)
  - *whole*: positive counting numbers and zero (0,1,2,3…)
  - *integer*: positive and negative counting numbers and zero (…-2,-1,0,1,2…)
- **missingValueCode**: Code for missing values (eg: '-999', 'NA', 'NaN'). NA otherwise. Note that an NA missing value code should be a string, 'NA', and numbers should also be strings, '-999.'
- **missingValueCodeExplanation**: Explanation for missing values, NA if no missing value code exists.

```
attributes1 <- data.frame(

    attributeName = c('Date', 'Location', 'Region','Sample_No', 'Sample_vol', 'Salinity', 'Temperature'
    attributeDefinition = c('Date sample was taken on', 'Location code representing location where sampl
    measurementScale = c('dateTime', 'nominal','nominal', 'nominal', 'ratio', 'ratio', 'interval', 'nom
    domain = c('dateTimeDomain', 'enumeratedDomain','enumeratedDomain', 'textDomain', 'numericDomain',
    formatString = c('MM-DD-YYYY', NA,NA,NA,NA,NA,NA,NA),
    definition = c(NA,NA,NA,'Sample number', NA, NA, NA, 'comments about sampling process'),
    unit = c(NA, NA, NA, NA,'milliliter', 'dimensionless', 'celsius', NA),
    numberType = c(NA, NA, NA,NA, 'real', 'real', 'real', NA),
    missingValueCode = c(NA, NA, NA,NA, NA, NA, NA, 'NA'),
    missingValueCodeExplanation = c(NA, NA, NA,NA, NA, NA, NA, 'no sampling comments'),
```

```
    stringsAsFactors = FALSE)
```

Typing this out in R can be a bit of a pain, so you can import a table made in another program (such as Excel) as your attribute table - just make sure that rows are attributes and column names match the column names as listed above *exactly* (case is important). `attributes1 <- read.csv('~/arctic-data/docs/training/EML/LakeSampleData.csv', stringsAsFactors = F)`

5.2.1.2 Defining enumerated domains

For attributes that are enumerated domains, a second table is needed with three columns: `attributeName`, `code`, and `definition`. `attributeName` is repeated for all codes belonging to a common attribute. To make things a little easier and less repetitve, coding wise, codes can be defined using named character vectors and then converted to a data frame. In this example, there are two enumerated domains in the attribute list - "Location" and "Region"

```
Location <- c(CASC = 'Cascade Lake', CHIK = 'Chikumunik Lake', HEAR = 'Heart Lake', NISH = 'Nishlik Lak
Region <- c(W_MTN = 'West region, locations West of Eagle Mountain', E_MTN = 'East region, locations Ea
```

The definitions are then written into a dataframe using the names of the named character vectors, and their definitions.

```
factors1 <- rbind(data.frame(attributeName = 'Location', code = names(Location), definition = unname(Lo
                  data.frame(attributeName = 'Region', code = names(Region), definition = unname(Region
factors1
```

```
##   attributeName  code                                    definition
## 1      Location  CASC                                  Cascade Lake
## 2      Location  CHIK                                Chikumunik Lake
## 3      Location  HEAR                                     Heart Lake
## 4      Location  NISH                                   Nishlik Lake
## 5        Region W_MTN West region, locations West of Eagle Mountain
## 6        Region E_MTN East region, locations East of Eagle Mountain
```

This table can also be generated using a different program, such as Excel, and imported to R as a .csv, similar to what can be done with the attribute table.

5.2.1.3 Generating the attribute list and data table

Next the attributeList is generated from the attributes and the factors using the function `set_attributes`. This puts all of the information from the attribute `data.frame` and the factor `data.frame` defining the enumerated domains into the slotted EML schema.

```
attributeList1 <- set_attributes(attributes1, factors = factors1)
```

Now the physical aspects of the data table, like its name, identifier (PID), header lines, and delimiter, need to be described. The function `set_physical` does this. See `?set_physical` for more options on what can be set in the physical element. One of the more important items to set here is the URL, which points to the newest version of the data object using the object's PID.

```
id1 <- 'PID1' #this should be an actual PID
path <- '~/arctic-data/docs/training/EML/LakeSampleData.csv' #path to data table
physical1 <- set_physical('LakeSampleData.csv',
                          id = id1,
                          size = as.character(file.size(path)),
                          sizeUnit = 'bytes',
```

```
                            authentication = digest(path, algo="sha1", serialize=FALSE, file=TRUE),
                            authMethod = 'SHA-1',
                            numHeaderLines = '1',
                            fieldDelimiter = ',',
                            url = paste0('https://cn.dataone.org/cn/v2/resolve/', id1))
```

If the object is already on the ADC, the physical section is very easy to write using its PID and the `pid_to_eml_physical` function:

```
id1 <- 'PID1' #this should be an actual PID
cn <- CNode('PROD')
mn <- getMNode(cn,'urn:node:ARCTIC')
physical1 <- pid_to_eml_physical(mn, id1)
```

REMEMBER, if we are working in the test environment, we set it this way:

```
cn <- CNode('STAGING')
mn <- getMNode(cn,'urn:node:mnTestARCTIC')
```

Reminder: Be very careful about what you publish on the production node (PROD, or arcticdata.io). This node should NEVER be used to publish test or training datasets. While going through training for the first time you should be using the test environment (STAGING, or test.articdata.io).

The `physical1` and `attributeList1` elements are then used to create the `dataTable`, along with the name of the `dataTable` and its description.

```
dataTable1 <- new('dataTable',
                  entityName = 'Lake Sample Data',
                  entityDescription = 'Water sample temperature and salinity from the Eagle Mountain re
                  physical = physical1,
                  attributeList = attributeList1)
```

5.2.1.4 Adding a second dataTable

If the metadata document describes multiple Data Objects, a new set of attributes, `attributeList`, `physical` description, and `dataTable` can be created just as in the example above.

```
attributes2 <- data.frame(attributeName = c('Time', 'Wind_Speed'),
                          attributeDefinition = c('Date and time of wind speed reading', 'Measured wind
                          measurementScale = c('dateTime', 'ratio'),
                          domain = c('dateTimeDomain', 'numericDomain'),
                          formatString = c('YYYY-MM-DD hh:mm:ss', NA),
                          definition = c(NA, NA),
                          unit = c(NA, 'metersPerSecond'),
                          numberType = c(NA, 'real'),
                          missingValueCode = c(NA, NA),
                          codeExplanation = c(NA, NA),
                          stringsAsFactors = FALSE)
attributeList2 <- set_attributes(attributes2)
id2 <- 'PID2'
physical2 <- pid_to_eml_physical(mn, id2)
dataTable2 <- new('dataTable',
                  entityName = 'EagleMtnWindData.csv',
                  entityDescription = 'Wind data from Eagle Mountain',
                  physical = physical2,
                  attributeList = attributeList2)
```

Now both `dataTable1` and `dataTable2` are added to the original EML using the `c()` function.

```
eml@dataset@dataTable <- c(dataTable1, dataTable2)
```

5.2.1.5 Defining custom units

If you get a warning message about your units not being in the standard unit list, a custom unit list needs to be created and added to the `additionalMetadata` slot of the EML. First we create a custom unit data frame with columns `id` (the name of the unit, as it appears in the attribute table), `unitType`, and `parentSI`. Each custom unit has a row in the data frame. To determine what the `unitType` and `parentSI` are for each unit, it may be helpful to reference the list of custom units. This list can easily be viewed in your R console using these commands:

```
standardUnits <- get_unitList()
View(standardUnits$units)
```

The following lines create the standard unit data table, use the function `set_unitList` to slot it into the EML, and then add it as `addionalMetadata` to the EML.

```
custom_units <- data.frame(id = c('horsepower', 'gallonPerMinute'),
                           unitType = c('power', 'volumetricRate'),
                           parentSI = c('watt', 'litersPerSecond'))
unitlist <- set_unitList(custom_units)
eml@additionalMetadata <- c(as(unitlist, "additionalMetadata"))
```

5.2.2 Setting other entities

5.2.2.1 Removing other entities

In cases where the data package was submitted originally via the web form, the original EML usually has the data tables described as "other entity" elements in the EML. This information is now redundant since we created data table elements describing these objects. Remove the other entities by replacing the `otherEntity` element in the EML with a `ListOfotherEntity` object that consists of an empty list.

```
eml@dataset@otherEntity <- new('ListOfotherEntity', list())
```

5.2.2.2 Adding other entities for files that aren't uploaded yet

There are times, however, when it may be necessary to create `otherEntity` tables for objects that are not described using a data table. Examples of these can include: R scripts, large NetCDF file directories, or audio/image files. Adding other entities is easy as long as you have paths to the files on the server and PIDs. This workflow is best if you are uploading files to the ADC yourself.

First get the list of pathnames for the files you are going to upload.

```
paths <- list.files("/home/sjclark/EML_learning/", full.names = TRUE)
```

Then generate PIDs for those files. These will be the data PIDs you use in `publish_update`, but note that you may have other data PIDs for data objects that are not other entities.

```
pids <- vapply(paths, function(x) { paste0("urn:uuid:", uuid::UUIDgenerate()) }, "")
```

This will guess the format ID from the file extension. These should be checked afterwards and potentially changed.

```
format_ids <- guess_format_id(paths)
```

Finally, a data frame is created with all of that information, and that information is added into the EML using `eml_add_entities`.

```
entity_df <- data.frame(type = 'otherEntity',
                        path = paths,
                        pid = pids,
                        format_id = format_ids,
                        stringsAsFactors = FALSE)
eml <- eml_add_entities(eml, entity_df)
```

5.2.2.3 Adding other entities for files that are already uploaded

If you are trying to describe data objects that are already on the ADC, you can utilize `lapply` along with `get_package` to easily write the EML `otherEntity` elements. In this case, there are data tables and other entity type elements mixed in, so the `otherEntity` elements (non-csvs) are picked out by hand using indexes. The numbers in the call `otherEnts <- pkg$data[c(5,6,9,10)]` will change depending on your dataset.

```
pid = 'doi:10.18739/A2408F'
pkg <- get_package(mn, pid, file_names = T)
otherEnts <- pkg$data[c(5,6,9,10)] #select only `otherEntity` PIDs after viewing `pkg$data` contents
eml@dataset@otherEntity <- new("ListOfotherEntity", pid_to_eml_other_entity(mn, otherEnts))
```

5.2.3 Setting coverages

Sometimes EML documents may lack coverage information describing the temporal, geographic, or taxonomic coverage of a dataset. This example shows how to create coverage information from scratch, or replace an existing coverage element with an updated one. You can view the current coverage (if it exists) by entering `eml@dataset@coverage` into the console. Here the coverage, including temporal, taxonomic, and geographic coverages, is defined using `set_coverage`.

```
coverage <- set_coverage(beginDate = '2012-01-01',
                         endDate = '2012-01-10',
                         sci_names = c('exampleGenus exampleSpecies1', 'exampleGenus ExampleSpecies2'),
                         geographicDescription = "The geographic region covers the lake region near Eag
                         west = -154.6192,
                         east = -154.5753,
                         north = 68.3831,
                         south = 68.3619)
eml@dataset@coverage <- coverage
```

You can also set multiple geographic (or temporal) coverages. Here is an example of how you might set two geographic coverages.

```
geocov1 <- new("geographicCoverage", geographicDescription = "The geographich region covers area 1",
               boundingCoordinates = new("boundingCoordinates",
                                         northBoundingCoordinate = new("northBoundingCoordinate", 68),
                                         eastBoundingCoordinate = new("eastBoundingCoordinate", -154),
                                         southBoundingCoordinate = new("southBoundingCoordinate", 67),
                                         westBoundingCoordinate =  new("westBoundingCoordinate", -155)))

geocov2 <- new("geographicCoverage", geographicDescription = "The geographich region covers area 2",
               boundingCoordinates = new("boundingCoordinates",
                                         northBoundingCoordinate = new("northBoundingCoordinate", 65),
                                         eastBoundingCoordinate = new("eastBoundingCoordinate", -155),
```

```
                                        southBoundingCoordinate = new("southBoundingCoordinate", 64),
                                        westBoundingCoordinate =  new("westBoundingCoordinate", -156))

coverage <- set_coverage(beginDate = '2012-01-01',
                         endDate = '2012-01-10',
                         sci_names = c('exampleGenus exampleSpecies1', 'exampleGenus ExampleSpecies2'))
eml@dataset@coverage@geographicCoverage <- c(geocov1, geocov2)
```

5.2.4 Setting methods

The methods tree in the EML section has many different options, visible in the schema. You can create new elements in the methods tree by following the schema and using the "new" command. Remember you can explore possible slots within an element by creating an empty object of the class you are trying to create. For example, `method_step <- new('methodStep')`, and using autocomplete on `method_step@`.

One very simple, and potentially useful way to add methods to an EML that have no methods at all is adding them via a word document. An example is shown below:

```
methods1 <- set_methods('methods_doc.docx')
eml@dataset@methods <- methods1
```

If you want to make minor changes to existing method information that has a lot of nested elements, your best bet may be to edit the EML manually in a text editor (or in RStudio), otherwise there is a risk of accidentally overwriting nested elements with blank object classes, therefore losing method information.

5.2.5 Adding people

To add people, with their addresses, you need to add addresses as their own object class, which you then add to the contact, creator, or associated party classes.

```
NCEASadd <- new('address',
                deliveryPoint = '735 State St #300',
                city = 'Santa Barbara',
                administrativeArea = 'CA',
                postalCode = '93101')
```

The creator, contact, and associated party classes can easily be created using functions from the `arcticdatautils` package. Here, we use `eml_creator` to set our dataset creator.

```
JC_creator <- eml_creator("Jeanette",
                          "Clark",
                          "NCEAS",
                          "jclark@nceas.ucsb.edu",
                          phone = '123-456-7890',
                          userId = 'https://orcid.org/WWWW-XXXX-YYYY-ZZZZ',
                          address = NCEASadd)
eml@dataset@creator <- c(JC_creator)
```

Similarly, we can set the contacts. In this case, there are two, so we set `eml@dataset@contact` as a `ListOfcontact`, which contains both of them.

```
JC_contact <- eml_contact("Jeanette",
                          "Clark",
                          "NCEAS",
                          "jclark@nceas.ucsb.edu",
```

```
                            phone = '805-893-5095',
                            userId = 'https://orcid.org/WWWW-XXXX-YYYY-ZZZZ',
                            address = NCEASadd)
JG_contact <- eml_contact("Jesse",
                            "Goldstein",
                            "NCEAS",
                            "jgoldstein@nceas.ucsb.edu",
                            phone = '123-456-7890',
                            userId = 'https://orcid.org/WWWW-XXXX-YYYY-ZZZZ',
                            address = NCEASadd)
eml@dataset@contact <- c(JC_contact, JG_contact)
```

Finally, the associated parties are set. Note that associated parties MUST have a role defined, unlike creator or contact.

```
JG_ap <- eml_associated_party("Jesse",
                              "Goldstein",
                              "NCEAS",
                              "jclark@nceas.ucsb.edu",
                              phone = '123-456-7890',
                              address = NCEASadd,
                              userId = 'https://orcid.org/WWWW-XXXX-YYYY-ZZZZ',
                              role = "metaataProvider")
eml@dataset@associatedParty <- c(JG_ap)
```

5.3 Validating and writing the EML

The last step is to write and validate the EML. For time-saving purposes, first you write the EML:

`path2 <- '/path/to/new/eml/Lake Physical Properties Data.xml' write_eml(eml, path2)`

Now, you validate the eml. The EML validation step indicates whether the EML that you've created is valid both with regard to EML and XML schema. Hopefully it returns `TRUE`.

`eml_validate(path2)`

If the EML validate returns `FALSE`, it is accompanied by an error that will be in this format:

`69.0: Element 'boundingCoordinates': This element is not expected. Expected is one of ( geographicDescription, references ).`

This error essentially says, the EML validator reached the slot `boundingCoordinates` but did not expect this element. Instead it expected either `geographicDescription` or `references`. Referring to the schema maps (eg: https://knb.ecoinformatics.org/emlparser/docs/eml-2.1.1/eml-coverage.png) you can see that before bounding coordinates, there must be a geographic description. The fix would be to return to your definiton of the geographicCoverage, and insert a `geographicDescription` into the geographicCoverage object (ie:`geocov1 <- new('geographicCoverage', geographicDescription = 'Description here',...)`).

5.4 Dealing with unusual cases

Typically, the biggest issue most datateam members have using the EML package is trying to add multiple values within a slot. In some cases, like adding multiple `dataTable` instances or multiple `creator` instances, this is easy, as is shown above. Other times, particularly if the slot is nested within a slot that we use a

helper function for (like `set_attributes`), it can get a little more challenging. By delving a little deeper into the EML S4 class of objects though, you can resolve most problems relatively easily.

Any slot that allows for a list of objects, as listed in the EML schema, will have a class called the slot name, in addition to a class called `ListOfslotName`. You can use the `new` function to create an object of the slot itself and a list of objects within that slot. For example:

```
m1 <- new('missingValueCode', code = 'NA', codeExplanation = 'No data taken')
m2 <- new('missingValueCode', code = '-999',  codeExplanation = 'Sensor malfunctioned')

codes <- new('ListOfmissingValueCode', list(m1, m2))
```

In this case, this slot is nested within a part of the EML that we typically construct with a helper function - so how do we actually get this list of missing value codes into the attribute table?

One strategy would be to construct the attribute list without the missing value code information for the attribute with mutliple attributes, and then insert the `ListOfmissingValueCode` into it. So, building our attribute list as normal,

```
attributes1 <- read.csv('~/arctic-data/docs/training/EML/LakeSampleData.csv', stringsAsFactors = F)
attlist1 <- set_attributes(attrbutes1)
```

You can then dig down into the individual attributes using the `@` functionality. Note that all of the attributes are stored as a `ListOfattributes` object. You can view a single value within this list by calling `attlist1@attributes@.Data[[n]]` where n represents the index of the list you want to view.

```
attlist1@attribute@.Data[[1]]
<attribute system="uuid">
  <attributeName>Date</attributeName>
  <attributeDefinition>Date sample was taken on</attributeDefinition>
  <measurementScale>
    <dateTime>
      <formatString>MM-DD-YYYY</formatString>
    </dateTime>
  </measurementScale>
</attribute>
```

You can also run commands on `attlist1@attribute@.Data` just like you would any other list. One useful one is `length(attlist1@attribute@.Data)` which will return the number of items in the list.

So to insert the list of missing value codes, you'll need to navigate to the index in the list of the attribute that you need to modify, and then dig even deeper into the slots. Don't forget that the `tab` key is your friend and can help you find slot names! So if you know that you need to change the 6th attribute in the list, you can type the following to see what is in there now:

```
attlist1@attribute@.Data[[6]]@missingValueCode

An object of class "ListOfmissingValueCode"
[[1]]
<missingValueCode/>
```

Now to add the list, you'll have to execute the following (repeating lines for clarity):

```
m1 <- new('missingValueCode', code = 'NA', codeExplanation = 'No data taken')
m2 <- new('missingValueCode', code = '-999',  codeExplanation = 'Sensor malfunctioned')

codes <- new('ListOfmissingValueCode', list(m1, m2))

attlist1@attribute@.Data[[6]]@missingValueCode <- codes
```

Now, if you view this attribute again you should see the mutliple missing value codes:

```
> attlist1@attribute@.Data[[6]]@missingValueCode
An object of class "ListOfmissingValueCode"
[[1]]
<missingValueCode>
  <code>NA</code>
  <codeExplanation>No data taken</codeExplanation>
</missingValueCode>

[[2]]
<missingValueCode>
  <code>-999</code>
  <codeExplanation>Sensor malfunctioned</codeExplanation>
</missingValueCode>
```

What I showed above is a pretty specific example (which conveniently is also one of our more common "rare cases"), but there are some general strategies you can employ to figure out how to insert other objects that need to be deeply nested into an EML.

- **Create an empty test object**: example: `test <- new('abstract')`
- **Use the `@` and `tab` functionality to explore slots in that object**: example: `test@para`
- *What are all these slots??*: Remember that the `schemaLocation`, `lang`, `slot_order`, and `section` slots are related to the schema and are not actual slots which contain values you can edit.
- *`.Data`*: if you run into a `.Data` using the `tab` key, that means that the slot can be part of a list.
- **Use indexing to access individual items in a list**: example: `test@para@.Data[[2]]`
- *Index out of bounds*: If you receive an "index out of bounds" error that means you are trying to access an index beyond what exists in the list. For example, if there are 4 paragraphs in the abstract, and you write `test@para@.Data[[5]]` you will recieve this error.
- **Use the schema!**: https://knb.ecoinformatics.org/#external//emlparser/docs/eml-2.1.1/index.html#moduleDescriptions

5.5 Exercise

- Read in the EML that you updated in the previous exercise into R
- Use the `eml` package to replace the exsiting `dataTable` slot with a new `dataTable` object with an attribute list and physical section you wrote in R
- Write and validate your EML
- Update your package with the new EML using `publish_update`
- Use the checklist to review your submission.
- Make edits where necessary, and publish updates as needed

Chapter 6

Using Git in RStudio

6.1 Introduction

Important! If you have never used git before, or only used it a little, or have no idea what is is, check out this intro to git put together by the ecodatascience group at UCSB. https://github.com/eco-data-science/github-intro-2/blob/master/index.pdf Don't worry too much about the forking and branching sections, as we will primarily be using the basic commit-pull-push commands. After you have read through that presentation, come back to this chapter.

6.1.1 So why do I need to use this again?

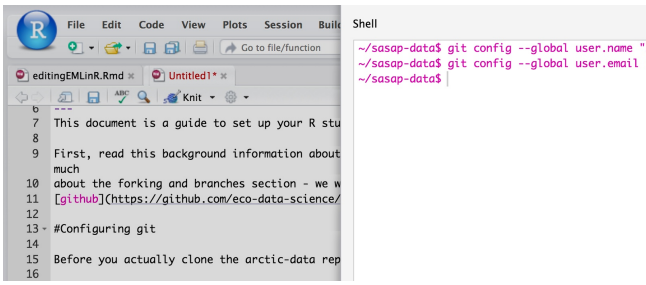
There are several reasons why using the arctic-data GitHub repository is helpful, both for you and for the rest of the data team. Here are a few:

- **Versioning**: Accidentally make a change to your code and can't figure out why it broke? Wish you could go back to that version that worked? If you add your code to the GitHub repo you can do this!
- **Reproducibility**: Being able to reproduce how you acomplished something is incredibly important. We should be able to tell anyone exactly how data have been reformatted, how metadata have been altered, and how packages have been created. As a data center, this is especially important for us with data team members that stay for 6-12 months becasue we may need to go back and figure out how something was done after the intern or fellow who wrote the code left the team.
- **Troubleshooting**: If you are building a particularly complicated EML, or doing some other advanced task, it is much easier for Jesse, Jeanette, or Bryce to troubleshoot your code if it is on the GitHub repo. We can view, troubleshoot, and fix bugs very easily when code is on the GitHub repo, with the added bonus of being able to go back a version if something should break.
- **Solve future problems**: Some of the issues we see in ADC submissions come up over and over again. When all of our code is on GitHub, we can easily reference code built for other submissions, instead of trying to solve the same problems over and over again from scratch.

6.2 Setting up git

Now you need to set up your Git global options, and tell it who you are. At the top of your RStudio window, select Tools > Shell. In the prompt, you will need to run two commands, one at a time. The first tells Git what your name is, the second what your email address is. These are the commands:

```
git config --global user.name "My Name" git config --global user.email myemail@domain.com
```

After running these commands, the shell prompt should look like this: 

6.2.1 Cloning the arctic-data repo

Next, you need to clone the arctic-data repository to your RStudio. You do this by adding it as a "project". In your RStudio window, click File > New Project. Then click 'Version Control', and then select the 'Git' option. If you are prompted to save your workspace during this process, make sure all of your work is saved, and you don't need anything in your environment, and then click 'Don't Save'.

You should see a prompt asking you for a URL. Fill it out like this to clone the arctic-data repository into the top level of your home directory. Note that the URL is the same URL you use to view the repository on the web. If you are using the sasap-data repository, the URL is http://github.nceas.ucsb.edu/NCEAS/sasap-data/.

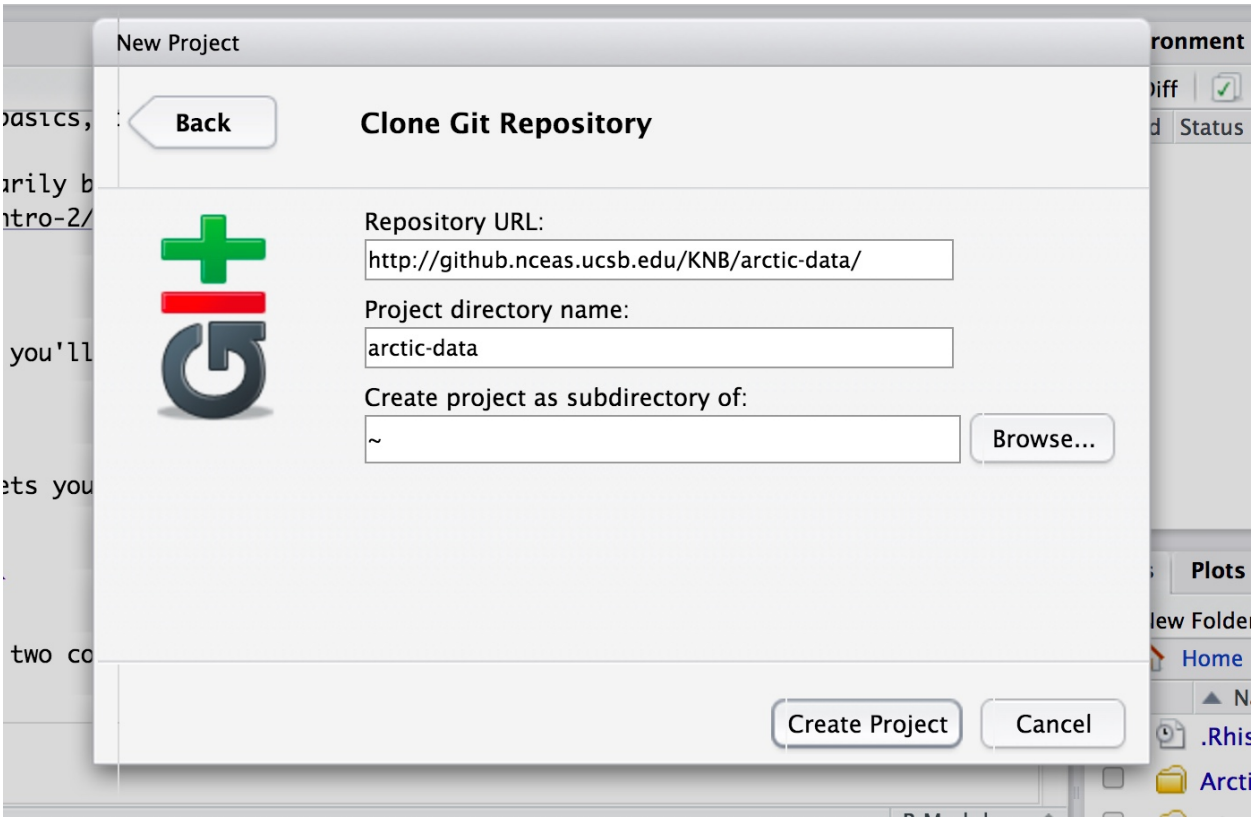


Figure 6.1: clone repo

You will be prompted for your username and password, and then Git will clone the directory. The username/password you use should be the same one you use to log in when you go to http://github.nceas.ucsb.edu/KNB/arctic-data. Now you should have a directory called arctic-data in your RStudio files window.

6.3 Working with the repository

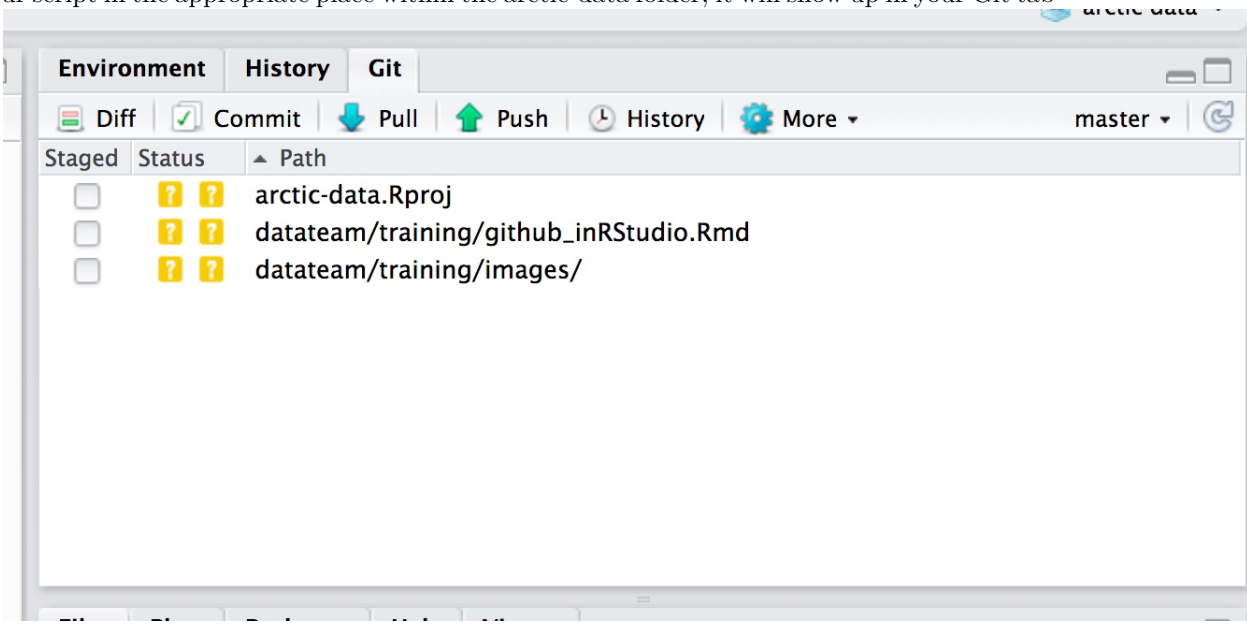
6.3.1 Adding a new script

If you have been working on a script that you want to put in the arctic-data GitHub repo, you need to save it somewhere in the arctic-data folder you made in your account on the server. You can do this by either moving your script into the folder or using the save-as functionality. Note that Git will try and version anything that you save in this folder, so you should be careful about what you save here. For our purposes, things that probably shouldn't be saved in this folder include:

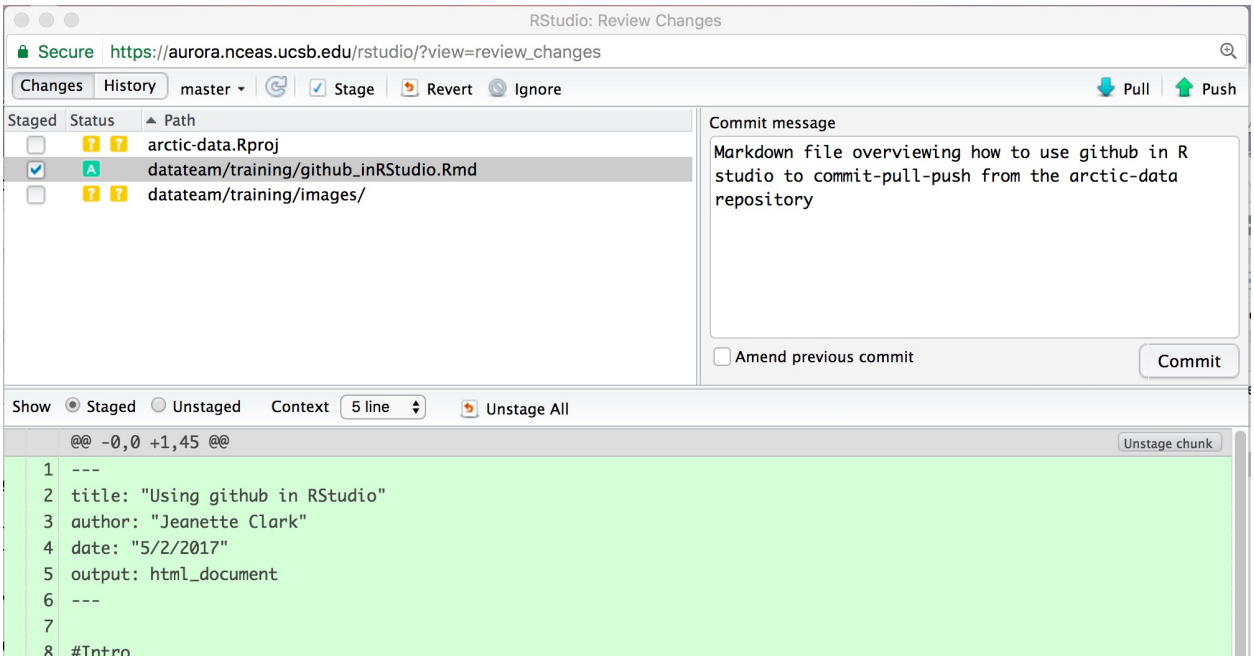
- **Tokens**: Any token file or script with a token in it should NOT be saved in the repository. Others could steal your login credentials if you put a token in GitHub.
- **Data files**: Git does not version data files very well. You shouldn't save any .csv files or any other data files (including metadata).
- **Workspaces/.RData**: If you are in the habit of saving your R workspace, you shouldn't save it in this directory.
- **Plots/Graphics**: For the same reasons as data files

Note: Do not EVER make a commit that you don't understand. If something unexpected (like a file you have never worked on) shows up in your Git tab, ask Jesse or Jeanette before committing.

After you save your script in the appropriate place within the arctic-data folder, it will show up in your Git tab looking like this:



Before you commit your changes, you need to click the little box under "staged." Do not stage or commit any .Rproj file. After clicking the box for your file, click "Commit" to commit your changes. In the window that pops up (you may need to force the browser to allow pop-ups), write your commit message. *Always* include a commit message. Remember that the commit message should be a concise description of the changes being made to a file. Your window should look like this:



Push 'Commit', and your commit will be saved. Now you want to merge the commits you made with the master version of the repository. You do this by using the command "Push." But before you push, you *always* need to pull first to avoid merge conflicts. Click "Pull" and type in your credentials. Then, assuming you don't have a merge conflict, you can push your changes by clicking "Push".

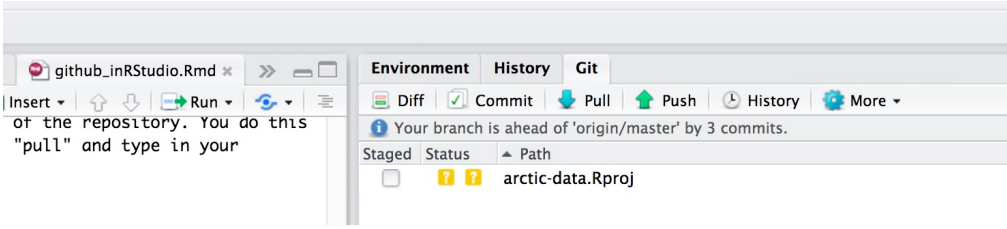
Always remember, the order is **commit-pull-push**

6.3.2 Editing a script

If you want to change a script, the workflow is the same. Just open the script that was saved in the arctic-data folder on your server account, make your changes, save the changes, stage them by clicking the box, commit, pull, then push to merge your version with the main version on the website. Do NOT edit scripts using the website. It is much easier to accidentally overwrite the history of a file this way.

One thing you might be wondering as you are working on a script is, how often should I be committing my changes? It might not make sense to commit-pull-push after every single tiny change - if only because it would slow you way down. Personally, I commit every time I feel that a significant change has happened and that the chunk of code I was working on is "done". Sometimes this is an entire script, other times it is just a few lines within a script. A good sign that you are committing too infrequently might be if many of your commit messages address a wide variety of coding tasks, such as: "wrote for loop to create referenced attribute lists for tables 1:20. also created nesting structure for this package with another package. also created attribute list for data table 40."

And one final note, you can make multiple commits before you push to the repo. If you are making lots of changes to the script, you might want to make several commits before pull-push. You can see how many commits you are ahead of the "origin/master" branch (i.e. what you see on the website) by looking for text in your



git tab in RStudio that looks like this:

6.3.3 Where do I commit?

The default right now is to save data-processing scripts in the `arctic-data/datateam/data-processing/` directory, with subfolders listed by project. Directories can be created as needed but please ask Jesse or Jeanette first so we can try and maintain some semblance of order in the file structure.

6.4 My Git tab disappeared

Sometimes R will crash so hard it loses your project information, causing your git tab to disappear. If this happens *anything you saved, but did not commit or push* in your arctic-data (or sasap-data) folder is no longer being tracked by GitHub. To get the tab back, first, rename your old arctic-data folder to something else (like 'arctic-data_old'). This will ensure that any work that you had in that folder that you didn't push to the master branch is not lost. Next, follow the steps above in "Cloning the arctic-data repo" to re-clone the repository. Then, move whatever scripts or parts of scripts that were not being tracked into the arctic-data repo so that they are tracked again, and merge them back into the master branch.

Ideally, you are committing and pushing your scripts frequently enough that you don't have to resort to this. If you had changes you had committed but not pushed, you can still push these changes from the command line. See Jeanette for more info on how to do this if it is the case.

Remember: *commit, pull, push* frequently (at least once a day).

Chapter 7

Using RT

7.1 Navigating RT

The RT ticketing system is how we communicate with folks interacting with the Arctic Data Center. We use it for managing submissions, accessing issues, etc. It consists of three separate interfaces:
Front Page
All Tickets
Ticket Page

7.1.1 Front Page

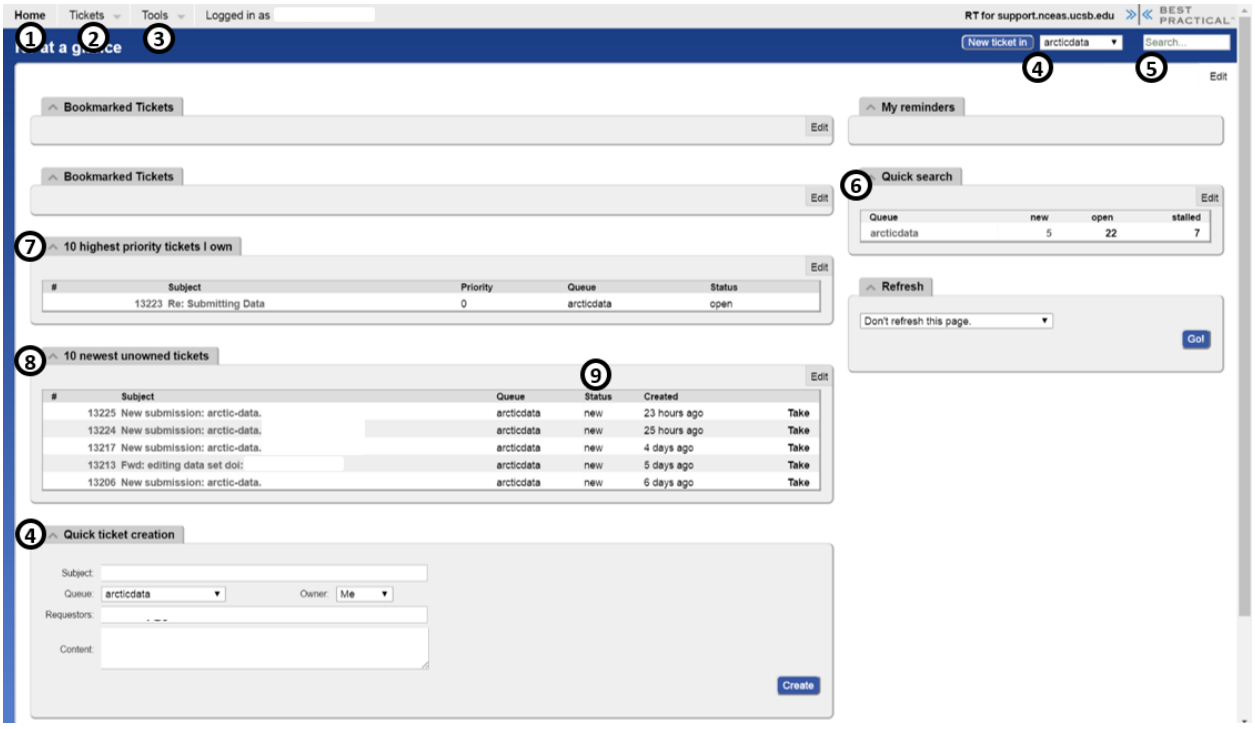


Figure 7.1:

This is what you see first

1. Home - brings you to this homepage
2. Tickets - to search for tickets (also see number 5)
3. Tools - not needed
4. New Ticket - create a new ticket
5. Search - Type in the ticket number to quickly navigate to a ticket
6. Queue - Lists all of the tickets currently in arcticdata and their status

- New = unopened tickets that require attention
- Open = tickets currently open, under investigation by team member
- Stalled = tickets awaiting response from PI/Submitter

7. Tickets I Own - These are the current open tickets that are claimed by me
8. Unowned Tickets - Newest tickets awaiting claim
9. Ticket Status - Status and how long ago it was created
10. Take - claim the ticket as yours

7.1.2 All Tickets

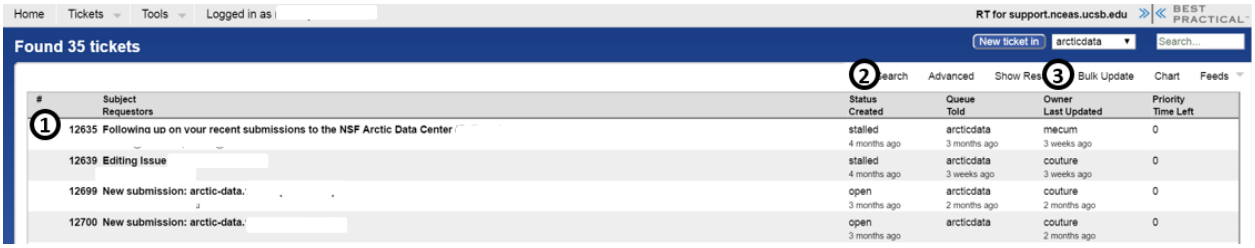


Figure 7.2:

This is the queue interface from number 6 of the Front page
1. Ticket number and title
2. Ticket status
3. Owner - who has claimed the ticket

7.1.3 Example Ticket

1. Title - Include the PI's name for reference
2. Display - homepage of the ticket

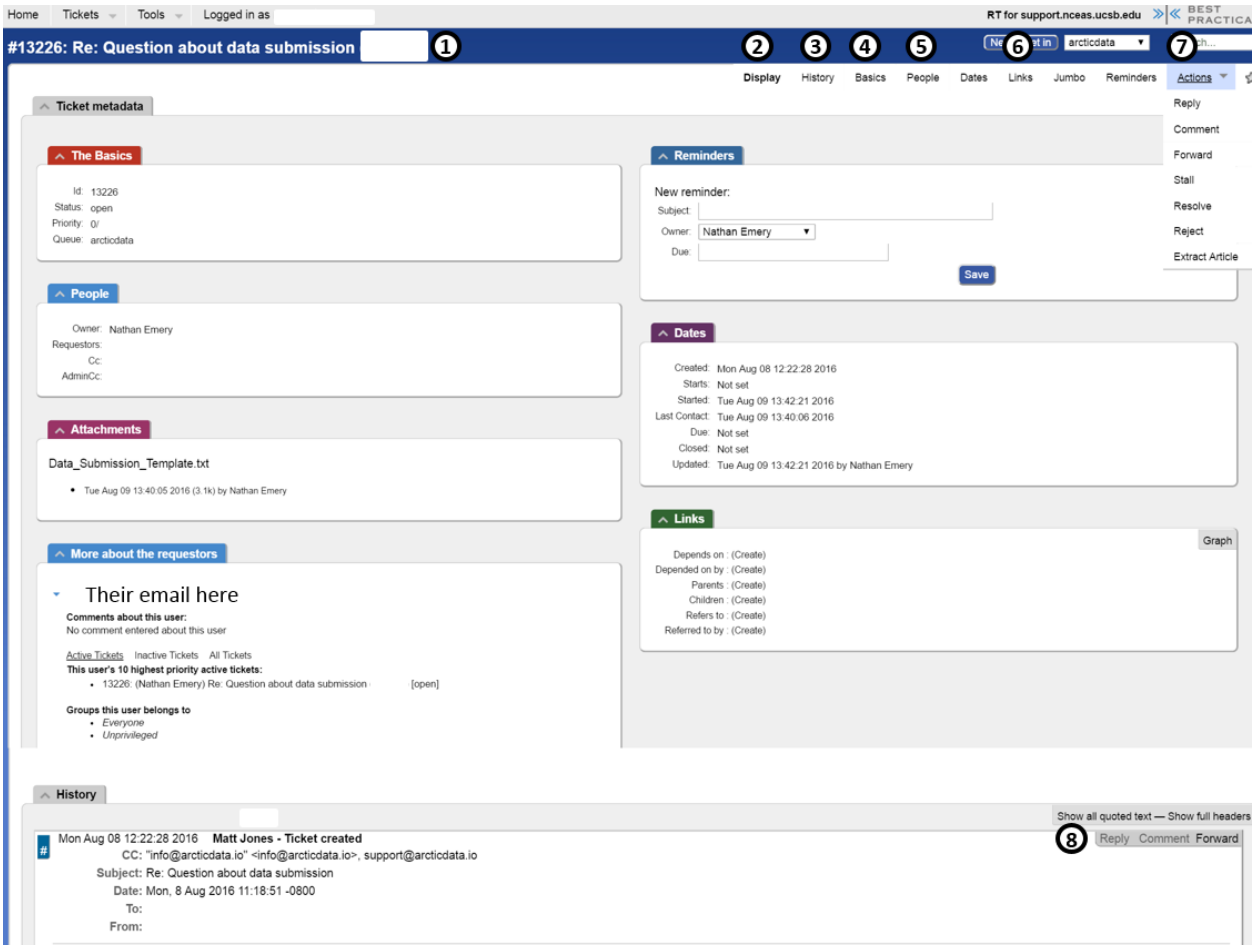

Home
Tickets
Tools
Logged in as
RT for support.nceas.ucsb.edu
#13226: Re: Question about data submission
1
2
3
4
5
6
7
Display
History
Basics
People
Dates
Links
Jumbo
Reminders
Actions
arcticdata
Ticket metadata
The Basics
Id: 13226
Status: open
Priority: 0/
Queue: arcticdata
People
Owner: Nathan Emery
Requestors:
Cc:
AdminCc:
Attachments
Data_Submission_Template.txt
Tue Aug 09 13:40:05 2016 (3.1k) by Nathan Emery
More about the requestors
Their email here
Comments about this user:
No comment entered about this user
Active Tickets Inactive Tickets All Tickets
This user's 10 highest priority active tickets:
13226: (Nathan Emery) Re: Question about data submission [open]
Groups this user belongs to
Everyone
Unprivileged
Reminders
New reminder:
Subject:
Owner: Nathan Emery
Due:
Save
Dates
Created: Mon Aug 08 12:22:28 2016
Starts: Not set
Started: Tue Aug 09 13:42:21 2016
Last Contact: Tue Aug 09 13:40:06 2016
Due: Not set
Closed: Not set
Updated: Tue Aug 09 13:42:21 2016 by Nathan Emery
Links
Graph
Depends on : (Create)
Depended on by : (Create)
Parents : (Create)
Children : (Create)
Refers to : (Create)
Referred to by : (Create)
Reply
Comment
Forward
Stall
Resolve
Reject
Extract Article
History
Show all quoted text — Show full headers
8
Reply Comment Forward
Mon Aug 08 12:22:28 2016 Matt Jones - Ticket created
CC: "info@arcticdata.io" <info@arcticdata.io>, support@arcticdata.io
Subject: Re: Question about data submission
Date: Mon, 8 Aug 2016 11:18:51 -0800
To:
From:

Figure 7.3:

3. History - Comment/Email history, see bottom of Display page

4. Basics - edit the title, status and ownership here

5. People - option to add more people to the watchlist for a given ticket conversation. Note that user/ PI/ submitter email addresses should be listed as "Requestors". Requestors are only emailed on "Replys", not "Comments". Ensure your ticket has a Requestor before attempting to contact users/ PIs/ submitters.

6. Links - option to "Merge into" another ticket number if this is part of a larger conversation. Also option to add a reference to other ticket number

7. Actions

- Reply - message the submitter/ PI/ all watchers
- Comment - attach internal message (no submitters, only Data Teamers)
- Open It - Open the ticket
- Stall - submitter has not responded in greater than 1 month
- Resolve - ticket completed

8. History - message history and option to reply (to submitter and beyond) or comment (internal message)

7.2 Initial review

7.2.1 Checklist

- Title
  - Provides the what, where, and when of the data
  - Does not use acronyms
- Abstract
  - Describes the DATA
- Data
  - At least one data file
  - No xls/xlsx files (or other proprietary files)
  - File contents and relationships among files are clear
  - All attributes are clearly defined
  - Column names do not use spaces or special characters
  - NA's are defined.
- Contacts
  - At least one contact with email and ORCiD
- Temporal/geographic coverage
  - Includes coverage that makes sense.
- Funding
  - At least one funding number
  - If there are multiple submissions with the same funding number/creators, check about nesting
- Methods
- Enough detail is provided in metadata

7.2.2 Starting email text

> Hello _____, Thank you for your recent submission to the NSF Arctic Data Center! We will review your dataset and get back to you with any suggestions.
>
> From my preliminary examination of your submission I have noticed a few items that I would like to bring to your attention. We are here to help you publish your submission, but your continued assistance is needed to do so. See comments below:
>
> [COMMENTS HERE]
>
> Best,
>
> [YOUR NAME]

7.2.3 Deadlines

If the PI is checking about dates/timing:

> We process submissions in the order in which they are received, and yours still has a few ahead of it in our queue. Are you facing any deadlines? If so, we may be able to expedite publication of your submission.

7.2.4 Pre-assigned DOI

If the PI needs a DOI right away:

> We can provide you with a pre-assigned DOI that you can reference in your paper. However, please note that it will not become active until after we have finished processing your submission and the package is published.

7.2.5 Title

7.2.5.1 Provides the *what, where, and when* of the data

> We would like to add some more context to your data package title. A descriptive title that provides context about the what, where, and when of a data package is often far more useful in search results. Something like: 'OUR SUGGESTION HERE, WHERE, WHEN' might be an option.

7.2.5.2 Does not use acronyms

> We noticed that your title contains several acronyms or abbreviations. In order to increase discoverability of your title, please spell out all acronyms and abbreviations.

7.2.6 Abstract

7.2.6.1 Describes DATA in package (ideally > 100 words)

> Your abstract, while informative, appears to be missing some information. We suggest that the abstract be sufficiently descriptive for a general scientific audience. It should provide an overview of the scientific context/ project/ hypotheses, how this data package fits into the larger project, a synopsis of the experimental or sampling design, and a summary of the data contents. If you

prefer and it is appropriate, we could add language from the abstract in the NSF Award found here: [NSF AWARD URL].

7.2.7 Data

7.2.7.1 At least one data file

We noticed that no data files were submitted. With the exception of sensitive social science data, NSF requires the submission of all data products prior to publication. Do you intend to submit data files?

7.2.7.2 No xls/xlsx files (or other proprietary files)

We noticed that the data files submitted are in xlsx format. Please convert your files to a plain text/csv (or other open source format) in order to facilitate an accurate transfer of information to users and to ensure preservation of the data in perpetuity.

We noticed you submitted your data as a .mat file. While the Arctic Data Center supports the upload of any data file format, sharing data can be greatly enhanced if you use ubiquitous, easy-to-read formats. We recommend conversion of your data to txt or csv (or other open) formats for better archival.

7.2.7.3 File contents and relationships among files are clear

Could you provide a short description of the files submitted? Information about how each file was generated (what software, source files, etc.) allows other scientists to reproduce your work.

7.2.7.4 All attributes are clearly defined

Check for descriptions of abbreviations and units:

Could you describe _____?

Please define “X”.

Please define “XYZ”, including the unit of measure.

What are the units of measurement for the columns labeled “ABC” and “XYZ”?

7.2.7.5 Column names do not use spaces or special characters

Data files should include column headers without special characters or spaces (i.e., Exp_no, NO3_M, Organic_M, as per “Some Simple Guidelines for Effective Data Management” - http://onlinelibrary.wiley.com/doi/10.1890/0012-9623-90.2.205/full).

7.2.7.6 NA’s are defined.

What do the NA’s in your measurements represent? (instrument failure, site not found, etc.)

We noticed that the data files contain blank cells. What do these represent?

7.2.8 Funding

- At least one funding number
- If there are multiple submissions with the same funding number/creators, check about nesting

  Thank you for your submission to the Arctic Data Center! Based on the NSF awards, your most recent submission and the [PACKAGE NAME] package appear to be related. Would you like your submission(s) grouped with the other data packages funded by the same award? If so, we are happy to do all this grouping for you.

  Thank you for your submission to the Arctic Data Center! Based on the NSF awards, your most recent submission and the [PACKAGE NAME] parent package seem like disparate projects, is that correct?

  Nesting data packages under a common "parent" is beneficial so that all packages funded by the same award can be found in one place. This way additional packages can be added to the group without altering existing ones. Once we process all the "child" data pacakges you upload, we can group them under the same "parent". The parent does not need to contain any data files itself, but will be populated with metadata only.

7.2.9 Methods

7.2.9.1 Enough detail is provided in metadata

Submissions should:

- provide instrument names
- specify how sampling locations were chosen
- provide citations for sampling methods that are not explained in detail

  Your methods, while informative, appear to be missing some information. Enough detail should be included so that a reasonable scientist can interpret the study and data for reuse without consulting you nor any other resources. This should hold true today, or even decades or a century from now. Users need to understand how the data were collected, how to interpret the values, and potentially how to use the data in the case of specialized files. We would be happy to add more content for you. Please provide us with a more robust methods section directly in this email or point us to a document from which we can extract more methods.

  NSF requires that comprehensive methods information be included directly in the metadata record. Pointers or URLs to other sites are unstable and insufficient.

7.3 Additional email templates

7.3.1 Best practices

We noticed that the data files submitted are in _____ format. We recommend conversion of these files into a plain text/csv (OR ANOTHER APPROPRIATE OPEN-SOURCE) format in order to facilitate an accurate transfer of information to future researchers and ensure preservation of the data in perpetuity. We will perform these conversions for you. Below are some linked articles about data science best practices that the NSF Arctic Data Center adheres to:
DataONE - https://www.dataone.org/best-practices "Some Simple Guidelines for Effective Data Management" - http://onlinelibrary.wiley.com/doi/10.1890/0012-9623-90.2.205/full "Good Enough Practices in Scientific Computing" - http://arxiv.org/pdf/1609.00037v1.pdf

7.3.2 DOI and data set finalization comments

Replying to questions about DOIs
We attribute DOIs to data sets as one might give a DOI to a citable publication. Thus, a DOI is permanently associated with a unique and immutable version of a data set. If the data set changes, a new DOI will be created and the old DOI will be preserved with the original version.

DOIs and URLs for previous versions of data sets remain active on the Arctic Data Center (will continue to resolve to the data set landing page for the specific version they are associated with), but a clear message will appear at the top of the page stating that "A newer version of this dataset exists" with a hyperlink to that latest version. With this approach, any past uses of a DOI (such as in a publication) will remain functional and will reference the specific version of the dataset that was cited, while pointing users to the newest version if one exists.

Clarification of updating with a DOI and version control
We definitely support updating a data set that has already been assigned a DOI, but when we do so we mark it as a new revision that replaces the original and give it its own DOI. We do that so that any citations of the original version of the data set remain valid (i.e.: after the update, people still know exactly which data were used in the work citing it).

Sending finalized URL before resolving ticket [NOTE: the URL format is very specific here, please try to follow it exactly (but subsitute in the actual DOI of interest)] Here is the link to your finalized data package:

https://doi.org/10.00000/X00X0X

Please let us know if you need any further assistance.

7.3.3 NSF ARC data submission policy

The NSF ARC program managers check compliance with their data policies when checking annual reports. Generally, NSF ARC requires that fully quality controlled data be uploaded within 6 months of collection for AON projects, or within 2 years of collection (or by end of the grant) for other ARC funded projects. Additionally, complete metadata must be submitted within two years of collection or before the end of the award, whichever comes first. Please find an overview of the NSF ARC policies here and the full policy information here.

Investigators should upload their data to the Arctic Data Center, or, where appropriate, to another community endorsed data archive that ensures the longevity, interpretation, public accessibility, and preservation of the data (e.g., GenBank, NCEI). Local and university web pages generally are not sufficient as an archive. Data preservation should be part of the institutional mission and data must remain accessible even if funding for the archive wanes (i.e., succession plans are in place). We would be happy to discuss the suitability of various archival locations with you further. In order to provide a central location for discovery of ARC-funded data, a metadata record must always be uploaded to the Arctic Data Center even when another community archive is used.

7.3.4 Adding metadata via R

KNB does not support direct uploading of EML metadata files through the website (we have a webform that creates metadata), but you can upload your data and metadata through R!

Here are some training materials we have that use both the `dataone` and `datapack` packages. It explains how to set your authentication token, build a package from metadata and data files, and publish the package to one of our test sites. I definitely recommend practicing on a test site prior to publishing to the production site your first time through. You can point to the KNB test node (dev.nceas.ucsb.edu) using this command:
`d1c <- D1Client("STAGING2", "urn:node:mnTestKNB")`

If you prefer, there are Java, Python, Matlab, and Bash/cURL clients as well.

7.3.5 Nesting

Nesting data sets under a common "parent" is beneficial so that all data sets funded by the same award can be found in one place. This way additional data sets can be added to the group without altering existing ones. Once we process all the "child" data sets you upload, we can group them under the same "parent". The parent will not contain any data files itself, but will be populated with metadata only. Please view an example of a parent data set here. Would you like your submission(s) grouped with the other data sets funded by the same award? If so, we are happy to do all this grouping for you.

7.3.6 Finding multiple data sets

If linking to multiple data sets, you can send a link to the profile associated with the submitter's ORCID ID and it will display all the data sets. Ex.: https://arcticdata.io/catalog/#profile/http://orcid.org/0000-0002-2604-4533

7.3.7 Asking for approval

Hi PI,

I have updated your data set and you can view it here after logging in: [URL]

Please review and approve it for publishing or let us know if you would like anything else changed. For your convenience, if we do not hear from you within a week we will proceed with publishing.

7.3.8 Other FAQs

Q: Can I replace data that has already been uploaded and keep the DOI?
A: Once you have published your data with the Arctic Data Center, it can still be updated by providing an additional version which can replace the original, while still preserving the original and making it available to anyone who might have cited it. To update your data, return to the data submission tool used to submit it, and provide an update.

Any update to a data set qualifies as a **new version** and therefore requires a new DOI. This is because each DOI represents a unique, immutable version, just like for a journal article. DOIs and URLs for previous versions of data sets remain active on the Arctic Data Center (will continue to resolve to the dataset landing page for the specific version they are associated with), but a clear message will appear at the top of the page stating that "A newer version of this dataset exists" with a hyperlink to the latest version. With this approach, any past uses of a DOI (such as in a publication) will remain functional and will reference the specific version of the dataset that was cited, while pointing users to the newest version if one exists.

Q: Why don't I see the data set that I uploaded to the ADC?
Possible Answer #1: The data set is still private because we are awaiting your approval to publish it. Please login with your ORCID ID to view private data sets. Possible Answer #2: The data set is still private and you do not have access because you were not the submitter. If you need access please have the submitter send us a message from his/her email address confirming this, along with your ORCID ID. Once we receive that confirmation we will be happy to grant you permission to view and edit the data set. Possible Answer #3: The data set is still private and we accidentally failed to grant you access. I apologize for the mistake. I have since updated the access policy. Please let us know if you are still having trouble viweing this data set here: (URL). Remember to login with your ORCID ID.

Issue: I would like to display multiple geographic coverages for my data set, but the form only accepts one point or bounding box.
Unfortunatley, the current web form does not allow users to input more than one geographic coverage. We are happy to add multiple coverages (maps) for you. Please provide us with the coordinates. Note that next version of the web form will support the addition of multiple locations by users. We anticipate this release in the coming months.

Issue: MANY files to upload (100s or 1000s).
A: Please consider zipping the files up for transfer. Can you upload the files to a drive we can access, such as G Drive or Dropbox? Alternatively, if you have a publically accesible FTP you can point us to, we could grab the files from there. If needed, we have a secure FTP you can access. Details are available here. Please access our server at datateam.nceas.ucsb.edu with the username "visitor". Let us know if you would like to use our SFTP and we will send you the password and the path to which directory to upload to.

Q: May another person (e.g. my student) submit data using my ORCID ID so that it is linked to me?
A: I would recommend instead that the student set up their own ORCID accounts at https://orcid.org/register and submit data sets from that account. Submissions are processed by our team and, at that point, we can grant you full rights to the metadata and data files even though another person submitted them.

Issue: Web form not cooperating.
A: I apologize that you are experiencing difficulties while attempting to submit your data set. We are happy to attempt to troubleshoot this for you. Which operating system (including the version) and browser (with version #) are you using? At which exact step did the issue arise. What error message did you recieve? Please provide us with any screenshots of the error message.

Q: May I submit a non-NSF funded data set?
A: Yes, you can submit non-NSF-funded Arctic data if you are willing to submit under the licensing terms of the Arctic Data Center (CC-0 or CC-BY), the data are moderately sized (with exact limits open to discussion), and a lot of support time to curate the submission is not required (i.e., you submit a complete metadata record and well formated, open-source data files). For larger data sets, we would likely need to charge a one-time archival fee which amortizes the long-term costs of preservation in a single payment. Also, please note that NSF-funded projects take priority when it comes to processing. Information on best practices for data and metadata organization is available here.

Q: Can I add another data file to an existing submission without having to fill out another metadata form?
A: Yes. Navigate to the data set after being sure to login. Then click the green "Edit" button. The form will populate with the already existing metadata so there is no need to fill it out again. Just scroll to the bottom of the form to "Upload Data", click "Choose File", and browse to the data file you wish to add. Be aware that the DOI will need to change after you add this file (or make any changes to a data set) as, just like for a journal article, a DOI represents a unique and immutable version. The current URL will remain functional, but clearly display a message at the top of that page stating "A newer version of this dataset exists" with a link to the latest version.

Q: Can I add these data anytime or is there some deadline associated with the grant, or some other restriction I'm not aware of?
A: We are happy to accept your submission any time; however, NSF has stricter policies. For all ARC supported projects, the policies state that "Complete metadata must be submitted… within two years of collection or before the end of the award, whichever comes first. All data and derived data products that are appropriate for submission… must be submitted within two years of collection or before the end of the award, whichever comes first." For all AON projects, "Real-time data must be made publicly available immediately. All data must be submitted… within 6 months of collection, and be fully quality controlled. All data sets and derived data products must be accompanied by a metadata profile and full documentation."

Q: Can you clarify what constitutes sensitive information (in relation to social science data and whether it needs to be uploaded)?
A: Sensitive information includes human subjects data and data that are governed by an Institutional Review Board policy. Data that are ethically or legally sensitive or at risk of decontextualization also constitute

sensitive information.

The NSF policy states "NSF realizes that on occasion there are data gathered of a particularly sensitive nature, such as the locations of archaeological sites or nest locations of endangered species. It is not the intention of this policy to reveal such information publicly. Discipline standards, indigenous community cultural rules, and state and federal regulations and laws should be followed for these types of data." The full policies are available here.

Q: Can we submit data as an Excel file?
A: While the Arctic Data Center supports the upload of any data file format, sharing data can be greatly enhanced if you use ubiquitous, easy-to-read formats. For instance, while Microsoft Excel files are commonplace, it is better to export these spreadsheets to Comma Separated Values (CSV) text files, which can be read on any computer without needing to have Microsoft products installed. Data submitted in Excel workbooks will undergo conversion to CSVs by our staff before being made public.

So, yes, you are free to submit an Excel workbook, however we strongly recommend converting each sheet to a CSV. The goal is not only for users to be able to read data files, but to be able to analyze them with software, such as R Studio. Typically, we would extract any plots and include them as separate image files.

[ONLY SAY THIS NEXT PART IF THE PI CONTINUES TO INSIST] I understand that having the plots in the same file as the data they are built from simplifies organization. If you definitely prefer to have the Excel workbook included, we ask that you allow us to document the data in both formats and include a note in the metadata clarifying that the data are indeed duplicated (but in different formats).

Chapter 8

Introduction to Solr

Solr is what's known as an index. More specifically, it's a piece of software that we install with every instance of Metacat which we use to query all of the Objects Metacat stores (metadata, data, resource maps, etc.). Metacat is our underlying metadata catalog and the foundation of our data repositories for both the CN and all MNs. Every Object in Metacat will have a corresponding Solr document for it that contains information about that Object. Each type of Object will have a different set of fields in its Solr document. For example, an EML document will have a `title` field (corresponding to the EML document's `<title>` element), while a CSV will not.

The fields that go into a Solr document are populated by information such as:

- The System Metadata (fileName, accessPolicy, etc)
- The Object itself (e.g. title, creators, etc. for an EML record)
- Additional computed fields (e.g., a geohash for quick spatial search)

Indexing the Object's Metacat stores lets us execute some interesting and very useful queries such as:

- What are the most recently updated datasets?
- What Metadata and Data Objects are in a given Data Package?
- What is the total size (in terms of disk space) of all Objects stored in Metacat?

8.1 Querying Solr

Solr is queried via what's called an HTTP API (Application Programming Interface). Practically, what this means it that you can visit a URL (web address) in a web browser to execute a query. This may be a little bit weird at first but I hope some examples will make it more clear.

So I said you visit a URL to query Solr. But what address do you visit? For the Arctic Data Center (https://arcticdata.io), every Solr query starts with a base URL of https://arcticdata.io/metacat/d1/mn/v2/query/solr. If you visit that URL, you will see a list of fields Solr is storing for the Objects it indexes:

```
<ns2:queryEngineDescription xmlns:ns2="http://ns.dataone.org/service/types/v1.1">
  <queryEngineVersion>3.6.2.2012.12.18.19.52.59</queryEngineVersion>
  <name>solr</name>
  <queryField>
    <name>abstract</name>
    <description>
      The full text of the abstract as provided in the science metadata document.
    </description>
    ...truncated...
```

You can see that there is a large set of queryable fields, though, as I said above, not all types of Objects will have values set for all of the possible fields because some fields do not make sense for some Objects (e.g., `title` for a CSV).

8.1.1 Parts of a Query

Each Solr query is comprised of a number of parameters. These are like arguments to a function in R, but they are entered as parts of a URL.

The most common parameters are:

- `q`: The query. This is like `subset` or `dplyr::filter` in R
- `fl`: What fields are returned for the documents that match your query (`q`). If not set, all fields are returned.
- `rows`: The maximum number of documents to return. Solr will truncate your result if the result size is greater than `rows`.
- `sort`: Sorts the result by the values in the given Solr field (e.g., sort by date uploaded)

To use these parameters, we append to the base URL like this:

`https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q={QUERY}&fl={FIELDS}&rows={ROWS}`

and we replace the text inside `{}` with the value we want for each parameter. Note that the parameters can come in any order so the following is equivalent:

`https://arcticdata.io/metacat/d1/mn/v2/query/solr/?fl={FIELDS}&rows={ROWS}&q={QUERY}`

The first parameter in the URL must have a '?' in front of it and all subsequent parameters must have an '&' between them. It's really easy to get the URL wrong when typing it in manually like this so be sure to double-check your URL and think critically about the result: Solr tries to always return something even if it's not what you intended.

8.1.2 Constructing a Query

The query ('q') parameter uses a syntax that looks like `field:value`, where `field` is one of the Solr fields and `value` is an expression. The expression can match a specific value exactly, e.g.,

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=identifier:arctic-data.7747.1`
- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=identifier:"doi:10.5065/D60P0X4S"`

which finds the Solr document for a specific Object by PID (identifier). Note that in the second example, the DOI PID is surrounded in double quotes. This is because Solr has reserved characters, of which ':' is one, so we have to help Solr by surrounding values with reserved characters in them in quotes (as I did here) or escaping them.

Queries can take on a more advanced form such as a wildcard expression:

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=identifier:arctic-data.*`

finds all the Objects that start with "arctic-data." followed by anything ("*")

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=title:*soil*`

finds all the Objects with the word "soil" somewhere in the title.

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=origin:*Stafford*+AND+title:Bering Strait&fl=title&sort=title+desc&rows=10`

finds 10 Objects where one of the EML `creator`s has a name that contains the substring "Stafford" and the `title` contains the substring "Bering Strait", sorted by `title` (descending order). Note that the `+AND+` between the `origin` and `title` query above specifies that both conditions must be true for a Solr document to be returned. We could've also switched the `+AND+` to `+OR+` and/or added more conditions to the query.

Here's a slightly more advanced one:

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=formatType:METADATA+AND+-obsoletedBy:*&sort=da`

This query is the query MetacatUI uses to fill in the https://arcticdata.io/catalog/ page. Notice the `-obsoletedBy:*`. The '-' before the field inverts the expression so this part of the query means "things that have no `obsoletedBy` value set".

We can also just find everything:

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=*:*`

finds any value in any field.

8.1.3 Faceting

Above we went through querying across Solr documents but we can also summarize what's in Solr with Faceting which lets us group Solr documents together and count them. This is like `table` in R. Faceting can do a query within a query, but more commonly I use it to summarize unique values in a field. For example, we can find the unique format IDs on Data Objects:

- https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=*:*&fq=formatType:DATA&facet=true&facet.field=formatId&rows=0

To facet, we usually do a few things:

- Add the parameter `facet=true`
- Add the parameter `facet.field={FIELD}` with the field we want to facet (group) on
- Set `rows=0` because we don't care about the matched Solr documents
- Optionally specify `fq={expression}` which filters out Solr documents before faceting. In the above example, we have to do this to only count Data Objects. Without it, the facet result would include formatIDs for metadata and resource maps which we don't want.

8.1.4 Stats

With Faceting, we found we could make queries to find the unique values for a Solr field. With Stats, we can have Solr calculate statistics on numerical values (such as `fileSize`).

- https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=formatType:DATA&stats=true&stats.field=size&rows=0

This query calculates a set of summary statistics for the `size` field on Data Objects that Solr has indexed. In this case, Solr's `size` field indexes the `fileSize` field in the System Metadata for each Object in Metacat.

8.2 Querying Solr through R

What if I told you that every time you run the `query` function in the `dataone` R package you are asking R to visit a URL like the ones above, parse the information returned by the page, and present it in an R-friendly way such as a `list` or `data.frame`? Well that's what happens! Then why might we use R in the first place? There are two big advantages:

1. The result is returned in a more useful way to R without extra work on your part

2. We can more easily pass our authentication token with the query

Why does #2 matter? Well by default, all of those URLs above only returned publicly-readable Solr documents. If a private document matched any of those queries, Solr doesn't give you any idea and acts like the non-public-readable documents don't exist. So we must pass an authentication token to access non-public-readable content. This bit is crucial for working with the ADC, so you'll very often want to use R instead of visiting those URLs in a web browser.

And there's good news: all of the URLs you visited above can be turned into an R expression very easily. For example:

- `https://arcticdata.io/metacat/d1/mn/v2/query/solr/?q=title:*soil*`

becomes

```
library(dataone)
cn <- CNode("PROD")
mn <- getMNode(cn, "urn:node:ARCTIC")
# Set your token if you need/want!
query(mn, "q=title:*soil*&fl=title&rows=10")
```

I just deleted the first part of the URL, up to and including the '?', and pasted the rest in as the second argument to `query`. You may have seen an alternative syntax:

```
query(mn, list(q="title:*soil*",
               fl="title",
               rows="10"))
```

this is the same query as above because the `query` function takes either a string (the first form) or a named list (the second form).

By default, `query` returns the result as a `list`. This is definitely useful but a `data.frame` can be a more useful way to work with the result. To get a `data.frame` instead, just set the `as` argument to 'data.frame' to get a `data.frame`:

```
query(mn, list(q="title:*soil*",
               fl="title",
               rows="10"),
      as = "data.frame")
```

```
1  Daily Average Soil, Air and Ground Temperatures - Council Forest Site [Romanovsky, V.]
2     Canadian Transect of Soils and Vegetation for the Circumpolar Arctic Vegetation Map
3                                    Soil Temperatures, Toolik Lake, Alaska, 1995 and 1996
4                                         Soil Temperature ARCSS grid Atqasuk, Alaska 2013
5                                                    X-ray fluorescence, Barrow soils
6                                               Toolik Lake, Alaska Soil moisture - 2014
7                           Thermal Soil Properties for Ivotuk and Council [Beringer, J.]
8                                      Soil Temperature NIMS grid Barrow, Alaska 2014
9                                       Ivotuk Soil Data - Station Met2 [Hinzman, L.]
10                                                    thule_tram_soil_temps_2013.csv
```

8.3 Key takeaways

- Find out what you can query at https://arcticdata.io/metacat/d1/mn/v2/query/solr
- The Solr HTTP API is what the R `dataone` package uses when you run `query`
- Pass a token if you want to include non-public-readable objects in the results (you often do!)

These skills are highly transferable. Note the syntax of the URL for a Google search.

8.4 More resources

- Solr's The Standard Query Parser docs (high level of detail)
- Another quick reference: https://wiki.apache.org/solr/SolrQuerySyntax
- http://www.solrtutorial.com/

Chapter 9

Nesting a data package

9.1 Introduction

This document is a tutorial for creating nesting relationships between data packages on the DataONE member nodes (MNs).

Data packages on MNs can exist as independent packages or in groups (nested data packages). Much like we can group multiple data files together with a common metadata file, we can group related data packages together with a common "parent" data package.

The structure of nested data packages resembles a pyramid. There is one top level, or "parent", with one or more data packages, or "children", nested beneath it. There is no limit to how many nested levels can be created, but packages do not generally exceed 3 levels. This example has three levels. The link will take you to the top level, or "grandparent". This top level has 5 children (nested datasets). If you click on any one of these children you can see that they have children of their own.

Some common uses for nesting:

- collected data vary by year
- an NSF award funds several related projects
- data collection is still ongoing
- data files exceed the 1000 file limit per data package

9.2 Exercise

We will create two data packages on the test node to nest beneath a parent. These data packages contain measurements taken from Lake E1 in Alaska in 2013 and 2014.

First, load the Arctic Data Center Test Node and libraries.

```
library(dataone)
library(arcticdatautils)
library(EML)
cnTest <- CNode('STAGING')
mnTest <- getMNode(cnTest,'urn:node:mnTestARCTIC')
```

- We can publish several data files at once using the `dir` and `sapply` functions. `dir` returns the paths to every file located in a folder.

```
dir("data/2013", full.names = T)
```

```
## [1] "data/2013/2012_2013_winter_E1_temperature.csv"
## [2] "data/2013/2013_summer_E1_temperature.csv"
```

- We can use `sapply` to apply a function over a list or vector. In this instance we will use it to apply `publish_object` over the vector of files paths (paths_2013) in order to publish them.

```
paths_2013 <- dir("/home/dmullen/Nesting_Training/2013", full.names = T)
data_2013 <- sapply(paths_2013, function(path) { publish_object(mnTest, path, format = "text/csv") })
```

- Publish the metadata file to the test node using `publish_object`.

```
metadata2013 <- publish_object(mnTest,
                               path = "/home/dmullen/Nesting_Training/E1_2013metadata.xml",
                               format = format_eml())
```

- Create a resource map to associate the data with the metadata. Here we specify the metadata and data identifiers (PIDs) that we would like the resource map to associate together. These are the identifiers of the objects we just published to the test node.

```
resourceMap2013 <- create_resource_map(mnTest,
                                       metadata_pid = metadata2013,
                                       data_pids = data2013)
```

- Repeat the same steps as above for the 2014 data package.

```
paths_2014 <- dir("/home/dmullen/Nesting_Training/2014", full.names = T)
data_2014 <- sapply(paths_2014, function(path) { publish_object(mnTest, path, format = "text/csv") })
metadata2014 <- publish_object(mnTest, path = "/home/dmullen/Nesting_Training/E1_2014metadata.xml", for
resourceMap2014 <- create_resource_map(mnTest, metadata_pid = metadata2014, data_pids = data2014)
```

These two packages correspond to data from the same study, varying only by year; however, they currently exist on the test node as independent entities. We will associate them with each other by nesting them underneath a parent. *Note:* Data packages are usually not entirely uploaded through the R client. Typically a PI will submit the metadata and data, and the resource map is automatically generated by the node.

9.3 Creating a Parent

In some cases a parent package already exists. Search the ADC for the NSF award number to see if there are already exisiting packages. Parents usually have a `UUID` rather than a `DOI` and often start with a title like "Collaborative research:", but not always. More typically, you will need to create a new parent by editing the existing metadata. The parent package should contain a generalized summary for the metadata of each of its children. It typically does not contain any data itself. We will now generalize a metadata file from one of the children in order to create the parent.

- Read in one of the children's metadata files (EML), in this case 2013.

```
eml_parent <- read_eml("data/E1_2013metadata.xml")
## Use this code when you are reading in the parent
# eml_parent <- read_eml("/home/dmullen/Nesting_Training/E1_2013metadata.xml")
## View the title
eml_parent@dataset@title
```

```
## An object of class "ListOftitle"
## [[1]]
## <title>Time series of water temperature, specific conductance and oxygen from Lake E1, North Slope, A
```

- The title of this child contains "2012-2013". This is too specific for the parent, as the temporal range of both children is 2012-2014. The parent should encompass this larger time range.

```
eml_parent@dataset@title <- c(new("title",
                                  .Data = "Time series of water temperature, specific conductance, and
```

- Like the title, the temporal coverage elements in this EML need to be adjusted. This field depends on whether or not the project has been completed.
  +If the project is ongoing and the PI plans on submitting data in the future, set the Temporal Coverage to the start date of the project using a `singleDateTime` element. We leave it open-ended and do not include an ending date.
  +If the project has been completed, set the Temporal Coverage to the start and end dates of the entire study. *In this case lets assume the project has been completed. We will update the temporal coverage to reflect the entire duration of the study. Since we are using the metadata from the first year of the project, we only need to update the `endDate` element.

```
eml_parent@dataset@coverage@temporalCoverage@.Data[[1]]@rangeOfDates@endDate@calendarDate <- new("calen
                                                                                            "2014-
```

- Remove `dataTable`s and `otherEntity`s from the metadata. If you recall from previous chapters, `dataTable`s contain metadata associated with data files (generally CSVs) and `otherEntity`s contain metadata about any other files in the data package (for instance a readMe or coding script). Because the parent does not contain any data objects, we want to remove `dataTables` and `otherEntities` from the metdata file. In this instance, the E1 2013 metadata only contains `dataTable`s. We can remove these by setting the `dataTable` element in the EML to a new blank object.

```
eml_parent@dataset@dataTable <- new("ListOfdataTable")
```

- In most cases elements like the abstract, contacts, creators, geographic description, and methods are already generalized and do not require changes.

- Before writing your parent EML make sure that it validates. This is just a check to make sure everything is in the correct format.

```
eml_validate(eml_parent)
```

- After your EML validates we need to save, or "write", it as a new file. Write your parent EML to a directory in your home folder. You can view this process like using "Save as" in Microsoft Word. We opened a file (E1_2013.xml), made some changes, and "saved it as" a new file called eml_parent.xml.

```
## This will create the file "eml_parent.xml" at the location specified by path
path = "/home/YOURUSERNAME/eml_parent.xml"
write_eml(eml_parent, path)
```

- Next, we will publish the parent metadata to the test node.

```
metadata_parent <- publish_object(mnTest, path = path, format_id = format_eml())
```

Finally, we create a resource map for the parent package. We nest the two child data packages using the `child_pids` argument in `create_resource_map`. Note that these `child_pids` are pids for the resource maps of the child packages, NOT the metadata pids.

```
resourceMap_parent <- create_resource_map(mnTest,
                                          metadata_pid = metadata_parent,
                                          child_pids = c(resourceMap2013, resourceMap2014))
```

The child packages are now nested underneath the parent.

9.4 Adding a child to a pre-existing parent

In some cases you will need to nest a new child underneath an existing parent. In this case we will nest the 2015 data for Lake E1 with our existing data package.

- First, create the 2015 child package.

```
paths_2015 <- dir("/home/dmullen/Nesting_Training/2015", full.names = T)
data_2015 <- sapply(paths_2015, function(path) { publish_object(mnTest, path, format = "text/csv") })
metadata2015 <- publish_object(mnTest, path = "/home/dmullen/Nesting_Training/E1_2015metadata.xml", for
resourceMap2015 <- create_resource_map(mnTest, metadata_pid = metadata2015, data_pids = data2015)
```

- Nest the new child using the `child_pids` argument in `update_resource_map()`. Note: it's important to include the *resource map PIDs of ALL the children* in the `child_pids` argument, otherwise the nesting relationships between any omitted children and the parent will be deleted.

```
resourceMap_parent2 <- update_resource_map(mnTest,
                                           resource_map_pid = resourceMap_parent,
                                           metadata_pid = metadata_parent,
                                           child_pids = c(resourceMap2013, resourceMap2014, resourceMap
```

- Nesting relationships can also be modified using the `child_pids` argument in `publish_update` in the same manner.

Chapter 10

Building Provenance

10.1 Introduction

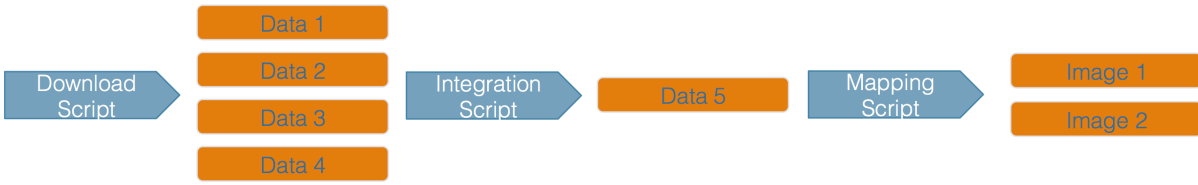
The provenance chain describes the origin and processing history of data. Provenance can exist on a continuum, ranging from prose descriptions of the history, to formal provenance traces, to fully executable environments. In this section we will describe how to build provenance using formal provenance traces in DataONE.

Provenance is becoming increasingly important in the face of what is being called a reproducibility crisis in science. J. P. A. Ioannidis (2005) wrote that "Most Research Findings Are False for Most Research Designs and for Most Fields". Ioannidis outlined ways in which the research process has lead to inflated effect sizes and hypothesis tests that codify existing biases.

The first step towards addressing these issues is to be able to evaluate the data, analyses, and models on which conclusions are drawn. Under current practice, this can be difficult because data are typically unavailable, the method sections of papers do not detail the computational approaches used, and analyses and models are often conducted in graphical programs, or, when scripted analyses are employed, the code is not available.

And yet, this is easily remedied. Researchers can achieve computational reproducibility through open science approaches, including straightforward steps for archiving data and code openly along with the scientific workflows describing the provenance of scientific results (e.g., Hampton et al. (2015), Munafò et al. (2017)).

At NCEAS and in the datateam, not only do we archive data and code openly, but we also describe the workflows that involve that data and code using provenance, formalizing the provenance trace for a workflow that



might look like this
into an easily understandable trace including archived data objects, such as what is shown here.

There are two ways that we add provenance in the datateam - the prov editor and the R `datapack` package.

10.2 The Prov Editor

Provenance can easily be added to production Arctic Data Center packages using the provenance editor on beta.arcticdata.io. On the landing page of a data package within beta, in the `dataTable` or `otherEntity` section where you would like to add a provenance relationship, you can choose to add either a "source" or a "derivation", to the left or right of the object pane, respectively.

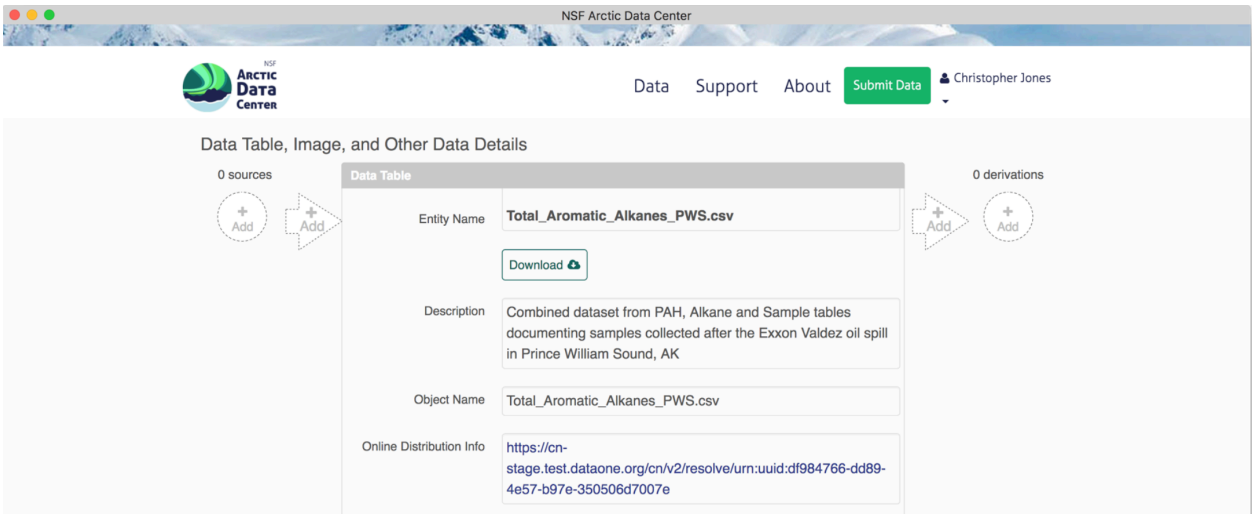
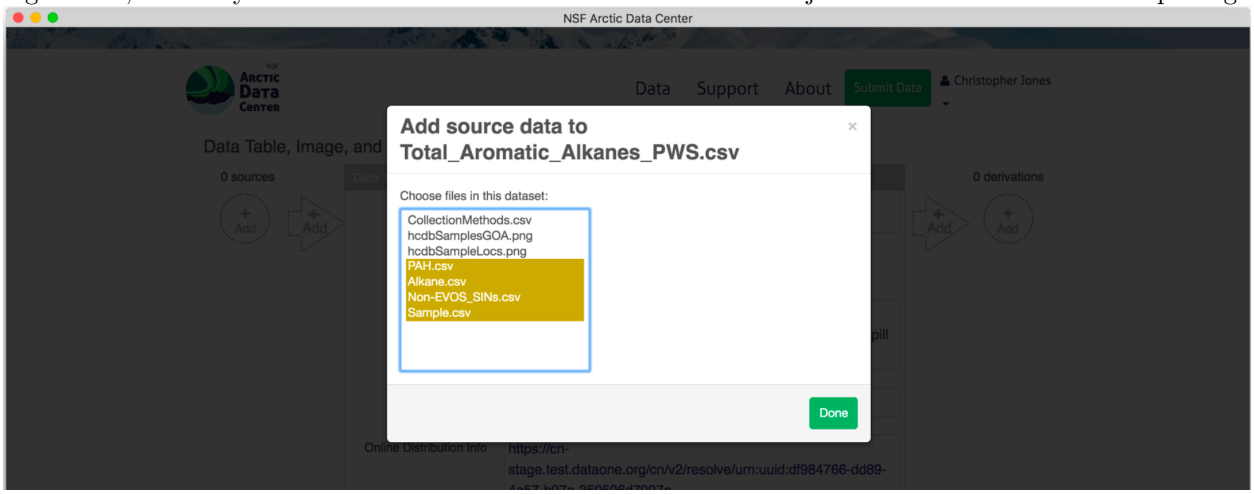
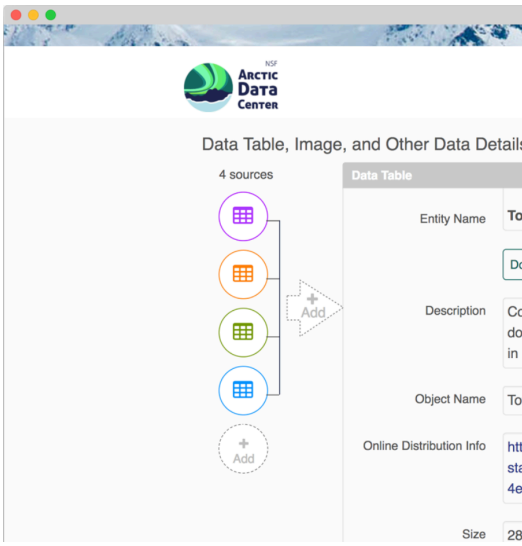


Figure 10.1: add prov

To add a source data file, click on the circle with the "+ add" text. Similarly, a source script would be added by selecting the arrow. Selecting the circle to add a source file pulls up the following screen, where you can select the source from other data objects within the same data package.





A data package with an object that has multiple sources added will look like this.

For simple packages on the Arctic Data Center, adding prov through the prov editor at beta.arcticdata.io is super easy!

10.3 Understanding resource maps

Before we dive further into constructing prov in R, we need to talk more about resource maps.

All Data Packages have a single Resource Map. But what is a Resource Map and how do we use one to find out what Objects are in a particular Data Package? This document is a short introduction but a more complete guide can be found here.

A Resource Map is a special kind of XML document that describes (amongst other things) an Aggregation. The Aggregation describes the members of a Data Package (metadata and data, usually). We can use the `dataone` R package to download a Resource Map if we know its PID:

```
library(dataone)

mn <- MNode("https://test.arcticdata.io/metacat/d1/mn/v2")
pid <- "urn:uuid:82bd7d7f-9e18-4fd2-8bda-99b1fddab556" # A Resource Map PID

path <- tempfile(fileext = ".xml") # We're saving to a temporary file but you can save elsewhere
writeLines(rawToChar(getObject(mn, pid)), path) # Write the object to `path`
```

If we open that file up on a text editor, we see this:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:cito="http://purl.org/spar/cito/" xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:dcte
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A61b48e72-ea29-4ba5-8131
    <cito:isDocumentedBy rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda
    <rdf:type rdf:resource="http://www.openarchives.org/ore/terms/Aggregation"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-bc82
    <cito:isDocumentedBy rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-bc82
```

```
    <cito:documents rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A61b48e72-ea29-4ba5-8
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-bc82-
    <cito:documents rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A61b48e72-ea29-4ba5-8131-
    <ore:isAggregatedBy rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <dc:title>DataONE Aggregation</dc:title>
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <dcterms:identifier rdf:datatype="http://www.w3.org/2001/XMLSchema#string">urn:uuid:82bd7d7f-9e18-4
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <ore:describes rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-bc82-
    <ore:isAggregatedBy rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <ore:aggregates rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A61b48e72-ea29-4ba5-8
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <ore:aggregates rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-
    <rdf:type rdf:resource="http://www.openarchives.org/ore/terms/ResourceMap"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A61b48e72-ea29-4ba5-8131-
    <dcterms:identifier rdf:datatype="http://www.w3.org/2001/XMLSchema#string">urn:uuid:61b48e72-ea29-4
  </rdf:Description>
  <rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-bc82-
    <dcterms:identifier rdf:datatype="http://www.w3.org/2001/XMLSchema#string">urn:uuid:c59b7505-39e6-4
  </rdf:Description>
</rdf:RDF>
```

Whoa. What is this thing and how do you read it? The short of it is that, if you want to find the members of the Data Package, you want to look for lines like this:

```
<rdf:Description rdf:about="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3A82bd7d7f-9e18-4fd2-8bda-9
    <ore:aggregates rdf:resource="https://cn.dataone.org/cn/v2/resolve/urn%3Auuid%3Ac59b7505-39e6-4def-
```

This line says "The Aggregation aggregates `urn:uuid:c59b7505-39e6-4def-bc82-b67a8d117ce5`" so that means `urn:uuid:c59b7505-39e6-4def-bc82-b67a8d117ce5` is in our Data Package! The key bit is the `<rdf:Description rdf:about="...#aggregation` part. If you look for another similar statement, you'll also see that `urn:uuid:61b48e72-ea29-4ba5-8131-4f59a9ebcd27` is part of our Data Package.

Now we know *which* Objects are in our Data Package but we don't know which one is metadata and which one is data. For that, we need to get a copy of the System Metadata for each object:

```
library(dataone)

mn <- MNode("https://test.arcticdata.io/metacat/d1/mn/v2")
getSystemMetadata(mn, "urn:uuid:c59b7505-39e6-4def-bc82-b67a8d117ce5")@formatId
[1] "eml://ecoinformatics.org/eml-2.1.1"
```

```
getSystemMetadata(mn, "urn:uuid:61b48e72-ea29-4ba5-8131-4f59a9ebcd27")@formatId
[1] "application/octet-stream"
```

From the format IDs, we can see the first PID is metadata and the second PID is data. Now we know enough to know what's in the Data Package:

- Resource Map: urn:uuid:82bd7d7f-9e18-4fd2-8bda-99b1fddab556
- Metadata: urn:uuid:c59b7505-39e6-4def-bc82-b67a8d117ce5
- Data: urn:uuid:61b48e72-ea29-4ba5-8131-4f59a9ebcd27

Now that you've actually seen a resource map, we can dive further into prov.

10.4 datapack

For packages not on the ADC, or packages that are extremely complicated, it may be best to upload prov relationships using R. The `datapack` package has several functions which help add relationships in a very simple way. These relationships are stored in the resource map. When you update a package just by adding prov, the package will not get any new identifiers with the exception of the resource map.

First, we set the environment, in a similar, but slightly different way than what you may be used to. Here the function `D1Client` sets the DataONE client with the coordinating node instance as the first argument, and membernode as the second argument.

```
library(dataone)
library(datapack)
d1c <- D1Client("STAGING2", "urn:node:mnTestKNB")
```

Next, get the pid of the resource map of the data package you are adding prov to, and load that package into R using the `getDataPackage` function.

```
resmapId <- "urn:uuid:8f501606-2c13-4454-b22d-050a4176a97b"
pkg <- getDataPackage(d1c, id=resmapId, lazyLoad=TRUE, limit="0MB", quiet=FALSE)
```

Printing `pkg` in your console shows you the contents of the data package, including all of the objects and their names:

```
> pkg
Members:

filename   format         mediaType  size     identifier                                  modified lo
esc...er.R application/R  NA         888      knb.92049.1                                 n        n
PWS....csv text/csv       NA         1871469  knb.92050.1                                 n        n
PWS....csv text/csv       NA         1508128  knb.92051.1                                 n        n
NA         eml:/...-2.1.1 NA         15658    urn:uuid:8f501606-2c13-4454-b22d-050a4176a97b n        y

Package identifier: resource_map_urn:uuid:8f501606-2c13-4454-b22d-050a4176a97b
RightsHolder: http://orcid.org/0000-0002-2192-403X
```

It will also show the existing relationships in the resource map, which in this case are mostly the "documents" relationships that specify that the metadata record is describing all of these data files.

```
Relationships:
                                  subject           predicate                                object
2          esc_reformatting_PWSweirTower.R cito:isDocumentedBy urn:uuid:8f501606-...4-b22d-050a4176a97b
4                      PWS_weirTower.csv cito:isDocumentedBy urn:uuid:8f501606-...4-b22d-050a4176a97b
1               PWS_Weir_Tower_export.csv cito:isDocumentedBy urn:uuid:8f501606-...4-b22d-050a4176a97b
```

```
3 urn:uuid:8f501606-...4-b22d-050a4176a97b     dcterms:creator                     _r1515542097r415842r1
5 urn:uuid:8f501606-...4-b22d-050a4176a97b      cito:documents          esc_reformatting_PWSweirTower.R
6 urn:uuid:8f501606-...4-b22d-050a4176a97b      cito:documents                     PWS_weirTower.csv
7 urn:uuid:8f501606-...4-b22d-050a4176a97b      cito:documents                PWS_Weir_Tower_export.csv
8 urn:uuid:8f501606-...4-b22d-050a4176a97b      cito:documents urn:uuid:8f501606-...4-b22d-050a4176a97b
9 urn:uuid:8f501606-...4-b22d-050a4176a97b cito:isDocumentedBy urn:uuid:8f501606-...4-b22d-050a4176a97b
```

In this example above, the data package has two .csv files, with an R script that converts one to the other. To create our provenance trace, first we need to select the source object, and save the pid to a variable. We do this using the `selectMember` function, and we can query part of the system metadata to select the file that we want. This function takes the data package (`pkg`), the name of the sysmeta field to query (in this case we use the `fileName`), and the value that you want to match that field to (in this case, 'PWS_Weir_Tower_export.csv').

```
sourceObjId <- selectMember(pkg, name="sysmeta@fileName", value='PWS_Weir_Tower_export.csv')
```

This returns a list of the source object pids that match the query (in this case only one object matches).

```
> sourceObjId
[1] "knb.92051.1"
```

Now we need to select our output object. Here, we use the `selectMember` function again, and save the result to a new variable.

```
outputObjId <- selectMember(pkg, name="sysmeta@fileName", value='PWS_weirTower.csv')
```

Now we query for the R script. In this case, we query based on the value of the `formatId` as opposed to the filename. This can be useful if you wish to select a large list of pids that are all similar.

```
programObjId <- selectMember(pkg, name="sysmeta@formatId", value="application/R")
```

Next, you use these lists of pids and a function called `describeWorkflow` to add these relationships to the data package. Note that if you do not have a program in the workflow, or a source file, you can simply leave those arguments blank.

```
pkg <- describeWorkflow(pkg, sources=sourceObjId, program=programObjId, derivations=outputObjId)
```

Viewing `pkg` again confirms that these relationships have been inserted into the data package, as shown by the "wasDerivedFrom" and "wasGeneratedBy" statements. It is always a good idea to print `pkg` to confirm that your pid selection process worked as expected, and your prov relationships make sense.

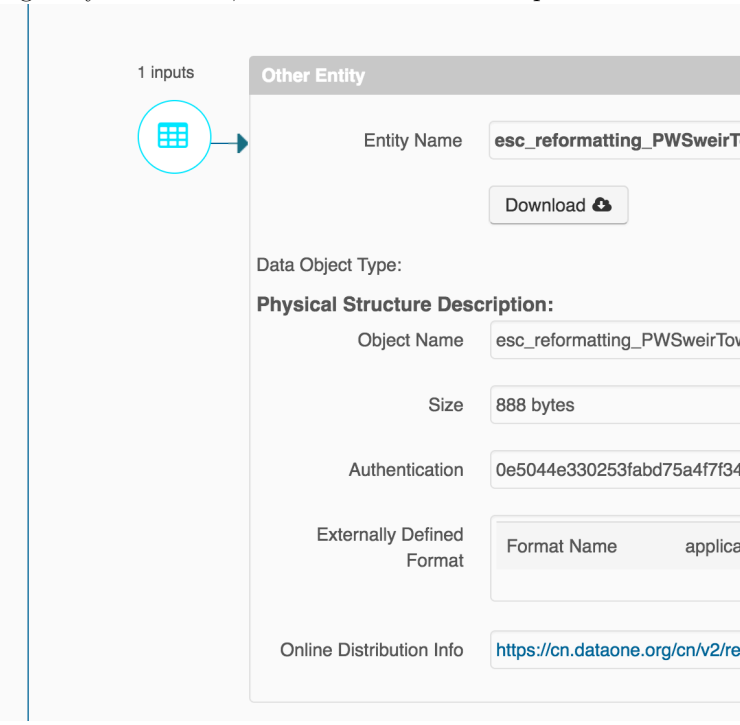
```
Relationships (updated):

                                subject                predicate                                object
15 _1db49d06-ae98-4...9101-39f7c0b45a95                 rdf:type                 prov:Association
14 _1db49d06-ae98-4...9101-39f7c0b45a95              prov:hadPlan      esc_reformatting_PWSweirTower.R
1       esc_reformatting_PWSweirTower.R       cito:isDocumentedBy urn:uuid:8f50160...b22d-050a4176a97b
16      esc_reformatting_PWSweirTower.R                 rdf:type                     provone:Program
8                     PWS_weirTower.csv       cito:isDocumentedBy urn:uuid:8f50160...b22d-050a4176a97b
11                    PWS_weirTower.csv                 rdf:type                         provone:Data
20                    PWS_weirTower.csv       prov:wasDerivedFrom            PWS_Weir_Tower_export.csv
19                    PWS_weirTower.csv       prov:wasGeneratedBy urn:uuid:3dd59b0...bc38-3b5d8fa644ac
6             PWS_Weir_Tower_export.csv       cito:isDocumentedBy urn:uuid:8f50160...b22d-050a4176a97b
10            PWS_Weir_Tower_export.csv                 rdf:type                         provone:Data
9                 _r1515544826r415842r1                foaf:name                      DataONE R Client
17 urn:uuid:3dd59b0...bc38-3b5d8fa644ac        dcterms:identifier urn:uuid:3dd59b0...bc38-3b5d8fa644ac
13 urn:uuid:3dd59b0...bc38-3b5d8fa644ac                 rdf:type                     provone:Execution
12 urn:uuid:3dd59b0...bc38-3b5d8fa644ac prov:qualifiedAssociation _1db49d06-ae98-4...9101-39f7c0b45a95
18 urn:uuid:3dd59b0...bc38-3b5d8fa644ac                 prov:used            PWS_Weir_Tower_export.csv
```

```
5  urn:uuid:8f50160...b22d-050a4176a97b           cito:documents      esc_reformatting_PWSweirTower.R
4  urn:uuid:8f50160...b22d-050a4176a97b           cito:documents                  PWS_weirTower.csv
3  urn:uuid:8f50160...b22d-050a4176a97b           cito:documents        PWS_Weir_Tower_export.csv
2  urn:uuid:8f50160...b22d-050a4176a97b           cito:documents urn:uuid:8f50160...b22d-050a4176a97b
7  urn:uuid:8f50160...b22d-050a4176a97b     cito:isDocumentedBy urn:uuid:8f50160...b22d-050a4176a97b
```

Finally, you can upload the data package using the `uploadDataPackage` function, which takes the DataONE client `d1c` we set in the beginning, the updated `pkg` variable, and some options for public read and whether informational messages are printed during the upload process.

```
resmapId_new <- uploadDataPackage(d1c, pkg, public=TRUE, quiet=FALSE)
```

If successful you should be able to navigate to the landing page of your dataset, and icons should show up where the sources and derivations are, such as in this example



10.4.1 Fixing mistakes

If you messed up updating a datapackage using `datapack`, there unfortunately isn't a great way to undo your work, as the `describeWorkflow` only adds prov relationships, it does not replace them. If you messed up, the best course of action is to update the resource map with a clean version that does not have prov using `update_resource_map`, and then go through the steps outlined above again.

Note: this has not been thorougly tested, and more extreme actions may be necessary to fully nuke the prov relationships. See Jeanette if things do not work as expected.

10.5 References

Ioannidis, John P A. 2005. "Why Most Published Research Findings Are False." PLoS Medicine 2 (8): e124. doi:10.1371/journal.pmed.0020124.

Hampton, Stephanie E, Sean Anderson, Sarah C Bagby, Corinna Gries, Xueying Han, Edmund Hart, Matthew B Jones, et al. 2015. "The Tao of Open Science for Ecology." Ecosphere 6 (July). doi:http:

//dx.doi.org/10.1890/ES14-00402.1.

Munafò, Marcus R., Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. 2017. "A Manifesto for Reproducible Science." Nature Human Behaviour 1 (1): 0021. doi:10.1038/s41562-016-0021.