PIC-SURE
Simplified R Library
Revision 2

Nick Benik
Harvard Medical School

11/12/2019

The PIC-SURE R library exists in two packages. The connection package ("`picsure::`") which handles connectivity and security processes needed to interact with a PIC-SURE network and, the HPDS package ("`hpds::`") which handles building queries and retrieving or otherwise interacting with data stored within a HPDS-hosted dataset.

PIC-SURE Network Functions & Resource Access

Connect to a PIC-SURE network:

You must establish a connection before you can perform any functions against a PIC-SURE network or any resources hosted on it. The following line shows how to connect to a PIC-SURE network using its endpoint URL and your security access token:

```
# create connection to PIC-SURE network
myconnection ← picsure::connect(url="http://some.url", token="my security token")
```

List all resources hosted on a PIC-SURE network:†

Once a connection is established, you are able to list all UUID identifiers for resources that you have access to on the PIC-SURE network by issuing the following commands:

```
# create connection to PIC-SURE network
myconnection ← picsure::connect(url="http://some.url", token="my security token")

# list accessible resources on the PIC-SURE network
resource.list ← picsure::list.resources(connection=myconnection)
```

Print details about a specific resource hosted on a PIC-SURE network:†

Most times you will need more information to determine which resource UUID contains data that is of interest to your research. To print detailed information about a resource you will pass the UUID that uniquely identifies it to the following function as shown:

```
# create connection to PIC-SURE network
myconnection ← picsure::connect(url="http://some.url", token="my security token")

# list accessible resources on the PIC-SURE network
resource.list ← picsure::list.resources(connection=myconnection)

# print details about the first resource returned by the previous line
picsure::resource.details(connection=myconnection, resource=resource.list[1])
```

†*In some configurations of a PIC-SURE network resource listing functionality is disabled*

Get a connection to a HPDS-hosted resource using a standard connection object:

This is the preferred way to connect to resources as it will allow access to other functionality relevant to a PIC-SURE network and not individual resources hosted within it.

```
# create connection to PIC-SURE network
myconnection ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to a resource
myresource ← hpds::get.resource(connection=myconnection, resourceUUID="some UUID")
```

Get a connection to a HPDS-hosted resource without using a standard connection object:

If you are only connecting to a single resource and you do not require any functionality exposed by the main "picsure" library then you can directly connect to the resource without first creating a connection to the PIC-SURE network.

```
# get a reference connection to a resource by specifying all connection and
authentication information as function parameters
myresource ← hpds::get.resource( url="http://some.url",
                                 token="my security token",
                                 resourceUUID="some UUID")
```

Get connections to multiple HPDS-hosted resources:

It is also possible for the library to manage simultaneous connections to more than one resource hosted within one or more PIC-SURE networks. In the follow example, we are connecting to two resources that are both hosted within the same PIC-SURE network. We can see how the connection object created on the first line is reused in the following two lines to simplify setup and reduce utilization of system resources on the computer.

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# get a reference connection to the Simmons Simplex Collection (SSC) resource
ssc.resource ← hpds::get.resource(connection=myconn, resourceUUID="sscUUID")
```

HPDS Resource Data Dictionary Functions

This section describes how to access and query the data dictionary of a dataset that is hosted by a HPDS-powered resource. All HPDS-related information is implemented in a second package called "hpds". The HPDS package depends upon the "picsure" package and uses it to handle all connectivity, communication and authorization with the PIC-SURE network that hosts any HPDS-powered resource.

Search a resource's ontology for entries which match a given term:

To search for ontology items related to a term you would first get a reference to the dictionary object of your HPDS-hosted dataset and then use that reference to query for a term. The results of that term search and then returned.

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# get a list of ontology results from the NHANES resource
asthma.terms ← hpds::find.in.dictionary(resource=nhanes.resource, term="asthma")
```

Extracting information from returned list of matching ontology results:

Once the list of matching terms is returned you can extract various information aspects from the results object by using the following functions:

```
# get the number of ontology terms from NHANES resource that match "asthma"
number.of.terms ← hpds::extract.count(dictionary.results=asthma.terms)

# get a list query keys that represent the matching
# ontology items within the NHANES resource
query.keys ← hpds::extract.keys(dictionary.results=asthma.terms)

# get a list of entries representing all information about the matching
# ontology items within the NHANES resource
term.details ← hpds::extract.entries(dictionary.results=asthma.terms)

# get a dataframe representation of the ontology terms from
# NHANES resource that match "asthma"
dataframe.for.terms ← hpds::extract.dataframe(dictionary.results=asthma.terms)
```

HPDS Resource Query Functions

This section describes how to access and query the actual data from a dataset that is hosted by a HPDS-powered resource. All HPDS-related information is implemented in a second package called "`hpds`". The HPDS package depends upon the "`picsure`" package and uses it to handle all connectivity, communication and authorization with the PIC-SURE network that hosts any HPDS-powered resource.

The basic process of getting information from a HPDS-powered data resource comprises three steps: 1) create a new instance of a "blank" query for the resource, 2) add selection and filtering criteria to the newly created query instance, 3) issue a query request using the query created in step 1 and modified by step 2 to get a list of matching records.

Step 1 – Create an instance of a blank query:

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# create a new instance of a blank query to run against the NHANES resource
my.nhanes.query ← hpds::new.query(resource=nhanes.resource)
```

Step 2 – Modify the newly created query to return data according to needs:

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# create a new instance of a blank query to run against the NHANES resource
my.query ← hpds::new.query(resource=nhanes.resource)

# modify blank query to return gender information
hpds::query.select.add(query=my.query, keys="\\demographics\\SEX\\")

# modify query to filter results to only records identifying females
hpds::query.filter.add(query=my.query, keys="\\demographics\\SEX\\", values="female")
```

Step 3 – Run the updated query to return matching data:

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# create a new instance of a blank query to run against the NHANES resource
my.query ← hpds::new.query(resource=nhanes.resource)

# modify blank query to return gender information
hpds::query.select.add(query=my.query, keys="\\demographics\\SEX\\")

# modify query to filter results to only records identifying females
hpds::query.filter.add(query=my.query, keys="\\demographics\\SEX\\", values="female")

# run the query and have it return results in a dataframe format
my.results ← hpds::query.run(query=my.query, result.type="dataframe")
```

The variable "`my.results`" now contains all records that match the filter and require criteria that was added to the query prior to running. Executing the query only returns columns specified using the `hpds::query.select.*` functions.

Allowable values for the result.type parameter are:

"count" - return only a count of how many records match the query.

"dataframe" - return a matrix containing columns representing configured "select" criteria and rows representing each matching record.

"crosscount" - return a vector containing columns representing configured "crosscounts" entries and one row containing occurrence counts for each column.

Description of the three elements comprising a query:

All queries against HPDS-powered datasets can consist of several elements: selections, requirements, any-of, cross-counts, and filters. Selections ("select") are used to specify record attributes will be returned by a query when it is run. Requirements ("require") specify which record attributes must be present, having any value, in order for the record to be selected by the query. Any-of criteria ("anyof") specify that matching records satisfy at least one of the criteria in the list in order to be selected. Cross-count entries ("crosscount") are used to generate a vector of counts for all keys entered into the field. Only records that pass all other criteria of the query will be counted. Filters ("filter") are used to only return records that have attributes with a specific value or a value within a specified range.

The `query.select.*` functions:

query.select.add() Adds keys of one or more record attributes to return in the query.

query.select.delete() Deletes keys of one or more record attributes from the query.

The `query.crosscount.*` functions:

query.crosscount.add() Adds keys of record attributes that will return an occurrence count.

query.crosscount .delete() Deletes keys of one or more cross-count record attributes from the query.

The `query.anyof.*` functions:

query.anyof.add() Adds keys of one or more record attributes that all returned records must have at least one match of.

query.anyof.delete() Deletes keys of one or more "any-of" record attributes from query.

The `query.filter.*` functions:

query.filter.add() Adds keys of one or more record attributes to return in the query.
Filter add behavior towards the construction of the raw query JSON depends on the arguments passed to it. Parameters are:

`keys` = string or vector of string with keys to add
`values` = one or more values that the key(s) should match
`min` = a minimum value key(s) should have, not used w/value
`max` = a maximum value key(s) should have, not used w/value

`Keys` is a required field, *`values`* may not be mixed with *`min`* or *`max`*. *`Min`* and *`max`* may me used by themselves or in combination with one another.

query.filter.delete() Deletes keys of one or more record attributes from the query.
Filter delete behavior is singular and consists of deleting any and all found key(s) that were passed to it regardless of the type of value the filter is.

Misc Code Examples

Dumping All Data from NHANES

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# get a list of all ontology results from the NHANES resource
all.terms ← hpds::find.in.dictionary(resource=nhanes.resource, term="")

# get a list of query keys that represent all ontology items within the NHANES resource
all.keys ← hpds::extract.keys(dictionary.results=all.terms)

# create a new instance of a blank query to run against the NHANES resource
my.query ← hpds::new.query(resource=nhanes.resource)

# modify blank query to return all information by adding keys for all ontology items
hpds::query.select.add(query=my.query, keys=all.keys)

# run the query and have it return results in a dataframe format
all.results ← hpds::query.run(query=my.query, result.type="dataframe")
```

Running two queries at the same time (i.e. control / experiment setup)

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# get a list of all ontology results from the NHANES resource
all.terms ← hpds::find.in.dictionary(resource=nhanes.resource, term="")

# get a list of query keys that represent all ontology items within the NHANES resource
all.keys ← hpds::extract.keys(dictionary.results=all.terms)

# create new instances of a blank queries to represent control and experiment groups
my.control ← hpds::new.query(resource=nhanes.resource)
my.experiment ← hpds::new.query(resource=nhanes.resource)

# add all keys to both queries so that all information is returned in the results
hpds::query.select.add(query=my.control, keys=all.keys)
hpds::query.select.add(query=my.experiment, keys=all.keys)

# differentiate the control and experiment groups via filtering on gender
hpds::query.filter.add(query=my.control, keys="\\demographics\\SEX\\", value="female")
hpds::query.filter.add(query=my.experiment, keys="\\demographics\\SEX\\", value="male")


# run the queries and return results in dataframe format
results.control ← hpds::query.run(query=my.control, result.type="dataframe")
results.experiment ← hpds::query.run(query=my.experiement, result.type="dataframe")
```

Showing the parameters (with values) of a given query instance:

```
# create connection to PIC-SURE network
myconn ← picsure::connect(url="http://some.url", token="my security token")

# get a reference connection to the NHANES resource
nhanes.resource ← hpds::get.resource(connection=myconn, resourceUUID="nhanesUUID")

# create new instance of a blank query
my.query ← hpds::new.query(resource=nhanes.resource)

# add some selection and filtering information to the query
hpds::query.select.add(query=my.query, keys="\\demographics\\AGE\\")
hpds::query.filter.add(query=my.query, keys="\\demographics\\SEX\\", values="female")

# show the current selection and retrieval criteria for the query
hpds::query.show(query=my.query)
```

General Function Specifications

Verbose Option

All functions in both packages have a named parameter called *verbose* which has a default value of `false`. If *verbose=true* is used in a function call then the function will output information to the screen as it is executed.

Notes and Miscellaneous

Research Notes

Comparison of various Object-Oriented systems used by R:
https://adv-r.hadley.nz/oo-tradeoffs.html

Getting your R package on CRAN:
https://kbroman.org/pkg_primer/pages/cran.html

Package Submission to Bioconductor:
https://www.bioconductor.org/developers/package-submission/

Bioconductor has strict review process. Bioconductor does not require packages to use S4 but most will because the key data structures are built using S4.