Example workflow of input and output to the Seatrack database

Jens Åström

2018-06-30

Contents

Summary

This is an example of a normal workflow that goes through the normal cycle of logger data. This cycle also describes the order in which the tasks is meant to be performed. There are several checks in the database that might throw an error if this order is not respected, e.g. if deployment info is registered before the logger is started up, or a logger is retrieved before it is deployed etc.

Finally, it is shown how to work with the file archive, that stores the raw output files from the loggers.

1. Connecting to the database
2. Importing logger information
    a. Registering loggers
    b. Starting a logging session
    c. Allocating loggers to a colony and species
3. Importing metadata
    a. Registering deployment data
    b. Registering retrieval data
    c. Registering individual bird info
4. Shutting down loggers
    a. Closing a logging session
    b. Creating file names associated with the logger download procedure
5. Storing download files
    a. Identifying which files the storage place expects
    b. Uploading logger files
6. Retrieving download files from the storage space (this step can be done anytime for already present files)

There are of course other important taskt, but these are not covered here. For example, sometimes it is necessary to update the lookup tables for standard information. This may be a new logger model, a new colony, new people working in the project and so on. Other tasks is to upload the processed positions files.

Firstly

Remember to always work using the lastest version of the R package SeatrackR. This is installed by:

```
devtools::install_github("NINAnor/seatrack-db/seatrackR")
```

As of this time, the current version is 0.0.1.6.

Connecting to the database

Use your name and password, provided to you elsewhere. Remember to change your password! This can be done with the `changePassword()` function, or for example in Pgadmin3.

For this work-through, we use the username: testwriter with password:testwriter. We also use several functions in the tidyverse package universe, so we'll load this as well.

```
library(seatrackR)
connectSeatrack(Username = "testwriter", Password = "testwriter")

## To change the password: changePassword('newpassword')
## connectSeatrack('testwriter', 'newpassword')

# This shouldn't be needed, but I will use dplyr and pipes in
# the following code.
library(tidyverse)
library(stringr)
```

Internally, `connectSeatrack` creates a connection to the database called `con` through the package `DBI` using the driver `Rpostgres::Postgres()`. Most functions used later checks that this connection is active and trows an error if it is not. Although you probably won't ever have to, you check the connection and also disconnect manually. Normally, you don't need to disconnect.

```
disconnectSeatrack()

seatrackR:::checkCon()  ##produces error if not connected
#> Error in seatrackR:::checkCon(): No connection, run connectSeatrack()

connectSeatrack(Username = "testwriter", Password = "testwriter")

seatrackR:::checkCon()  ##returns nothing if the connection exists
```

Importing logger info, allocating loggers, and starting logging sessions.

There are two major routes for importing logger data into the database. The first is through the table imports.logger_import, which takes info on logger serial numbers and models, startup info, allocation info,

and shutdown info. This table is just a pipeline to other tables in the database. It redistributes data to several tables depending on what it is fed but is always itself empty. This is meant as a convenience for the user so that they don't have to interact to more tables than necessary.

As of today, the redistribution rules are:

- If the `logger_serial_number` + the `logger_model` column is not already present in the loggers.logger_info table, this info is added as a new logger and given a new logger_id in the loggers.logger_info table. This means that typoes in this import can result in registering non existent loggers! Make sure the logger serial numbers are correct when importing this data. These colums are then written to the loggers.logger_info table:
  - logger_erial_no
  - producer
  - logger_model
  - production_year
  - project
- If the column `starttime_gmt` is not empty (NULL), the logger is started up. A new logging session is registered in the loggers.logging_session as active, and these columns are moved to the loggers.startup table:
  - logger_id
  - starttime_gmt
  - logging_mode
  - started_by
  - started_where
  - days_delayed
  - programmed_gmt_time
- If the column `intended_species` is not empty (NULL), the allocation data is moved to the loggers.allocation table. These columns are filled in in the table loggers.startup:
  - logger_id
  - intended_species
  - intended_location
  - intended_deployer
- Lastly, if the column `shutdown_session` is True, the logging session is shutdown in the table loggers.logging_session and info about the shutdown is imported into the loggers.shutdown table. If the column `download_type` at the same time either "Successfully downloaded", or "Reconstructed", filenames are also generated in the `loggers.file_archive` table. These columns are imported into the loggers.shutdown:
  - session_id
  - download_type
  - download_date
  - field_status
  - downloaded_by
  - decomissioned

The table `sampleLoggerImport` contains an example of information required to register, startup, and allocate a number of loggers. This is written to the `imports.logger_import` table by the function `writeLoggerImport`. The sample data contains both information on new loggers, their startup, and allocations. Note that we don't have to include info on all these steps in the same go. It is fine to first just send the columns that contain the info on the logger serial numbers, then the ones that starts them, and lastly the ones that allocates them. Remember also that if you also include info on shutdown in the same go (shutdown_session = True), then the session is closed and you won't be able to upload deployment or retrieval data. The order of input matters!

```
sampleLoggerImport
#> # A tibble: 79 x 22
#>    logger_serial_no logger_model producer   production_year
#>    <chr>            <chr>        <chr>                <dbl>
```

```
#>  1 Z231             c65          Migrate T~            2013
#>  2 Z236             c65          Migrate T~            2013
#>  3 Z234             c65          Migrate T~            2013
#>  4 Z232             c65          Migrate T~            2013
#>  5 Y604             f100         Migrate T~            2013
#>  6 Y612             f100         Migrate T~            2013
#>  7 Y614             f100         Migrate T~            2013
#>  8 Y595             f100         Migrate T~            2013
#>  9 Y116             c330         Migrate T~            2013
#> 10 Y099             c330         Migrate T~            2013
#> # ... with 69 more rows, and 18 more variables:
#> #   project <chr>, starttime_gmt <date>,
#> #   logging_mode <dbl>, started_by <chr>,
#> #   started_where <chr>, days_delayed <dbl>,
#> #   programmed_gmt_time <date>, intended_species <chr>,
#> #   intended_location <chr>, intended_deployer <chr>,
#> #   shutdown_session <lgl>, shutdown_date <lgl>,
#> #   field_status <lgl>, downloaded_by <lgl>,
#> #   download_type <lgl>, download_date <lgl>,
#> #   decomissioned <lgl>, comment <lgl>
```

We can check how many of the rows in the table about to be imported that has starttimes, and will result in started logging sessions.

```
noStartups <- sampleLoggerImport %>% summarize(no_startups = sum(!is.na(starttime_gmt)))
noStartups
#> # A tibble: 1 x 1
#>   no_startups
#>         <int>
#> 1          79
```

So, the import of this data should result in 79 active sessions (since we here start with an empty database). Next, we import the logger startup data.

```
writeLoggerImport(sampleLoggerImport)
#> Warning in result_create(conn@ptr, statement): Closing open
#> result set, cancelling previous query
#> [1] TRUE
```

We can use some convenience functions to checkout some of the newly imported data. The `getLoggerInfo` function reads from the loggers.logger_info table, which stores basic information of each registered logger (in use or not).

```
loggerInfo <- getLoggerInfo()  # This reads from the loggers.logger_info table
loggerInfo
#> # A tibble: 77 x 7
#>    id      logger_id logger_serial_no logger_model producer
#>    <chr>       <int> <chr>            <chr>        <chr>
#>  1 9aeb8b~         1 Z231             c65          Migrate~
#>  2 9aecf9~         2 Z236             c65          Migrate~
#>  3 9aed5c~         3 Z234             c65          Migrate~
#>  4 9aedbe~         4 Z232             c65          Migrate~
#>  5 9aee1e~         5 Y604             f100         Migrate~
#>  6 9aee81~         6 Y612             f100         Migrate~
#>  7 9aeee6~         7 Y614             f100         Migrate~
#>  8 9aef0a~         8 Y595             f100         Migrate~
```

```
#>  9 9aef2e~         9 Y116             c330         Migrate~
#> 10 9aef50~        10 Y099             c330         Migrate~
#> # ... with 67 more rows, and 2 more variables:
#> #   production_year <int>, project <chr>
```

We see that we have 77 registered loggers. Next, we can have a look at the current active sessions, most easily through the `getActiveSessions` function.

```
activeSessions <- getActiveSessions()  # This reads from the table loggers.logging_session.
activeSessions
#> # A tibble: 79 x 12
#>    id       session_id logger_id deployment_id retrieval_id
#>    <chr>         <int>     <int>         <int>        <int>
#>  1 9aec586~          1         1            NA           NA
#>  2 9aed202~          2         2            NA           NA
#>  3 9aed833~          3         3            NA           NA
#>  4 9aede23~          4         4            NA           NA
#>  5 9aee427~          5         5            NA           NA
#>  6 9aeeaa4~          6         6            NA           NA
#>  7 9aeef5b~          7         7            NA           NA
#>  8 9aef185~          8         8            NA           NA
#>  9 9aef3b3~          9         9            NA           NA
#> 10 9aef5eb~         10        10            NA           NA
#> # ... with 69 more rows, and 7 more variables:
#> #   active <lgl>, colony <chr>, species <chr>,
#> #   year_tracked <chr>, individ_id <chr>,
#> #   last_updated <dttm>, updated_by <chr>
```

We se that there are 79 open sessions, meaning they have been started up but not shut down. We can se how many of these have been deployed and retrieved by counting the number of rows with deployment id and a retrieval id. Note that we exclude the rows with NAs, which signifies missing data and is read as `NULL` in the database.

```
activeSessions %>% summarise(no_deployed = sum(!is.na(deployment_id)),
    no_retrieved = sum(!is.na(retrieval_id)))
#> # A tibble: 1 x 2
#>   no_deployed no_retrieved
#>         <int>        <int>
#> 1           0            0
```

At this point all 79 loggers are started up, but none is registered as deployed or retrieved.

Importing metadata (field information)

When the loggers are registered and started up, we can upload some metadata. This conforms to the existing metadata sheets used in the field. Note here the correct order of input; first start up a session through the `writeLoggerImport` function, then import deployment info using the `writeMetadata()` function, then do the same with the retrieval data. You can import deployment and retrieval data in the same go if they appear in the right order in the metadata file (sort by date to make it so).

```
sampleMetadata
#> # A tibble: 178 x 39
#>    date       ring_number euring_code color_ring
#>    <date>     <chr>       <chr>       <chr>
```

```
#>  1 2016-01-07 5175137     NOS         <NA>
#>  2 2016-01-07 5175138     NOS         <NA>
#>  3 2016-01-07 5175139     NOS         <NA>
#>  4 2016-01-07 5175140     NOS         <NA>
#>  5 2016-02-07 2000741     NOS         Red SM
#>  6 2016-02-07 2000903     NOS         Red ZY
#>  7 2016-02-07 2001020     NOS         Red SF
#>  8 2016-04-07 5104211     NOS         <NA>
#>  9 2016-04-07 5175141     NOS         <NA>
#> 10 2016-04-07 5175142     NOS         <NA>
#> # ... with 168 more rows, and 35 more variables:
#> #   logger_status <chr>, logger_model_retrieved <chr>,
#> #   logger_id_retrieved <chr>, logger_model_deployed <chr>,
#> #   logger_id_deployed <chr>, species <chr>, morph <chr>,
#> #   subspecies <chr>, age <dbl>, sex <chr>,
#> #   sexing_method <chr>, weight <dbl>, scull <dbl>,
#> #   tarsus <dbl>, wing <dbl>, breeding_stage <chr>,
#> #   eggs <dbl>, chicks <dbl>, hatching_success <lgl>,
#> #   breeding_success <lgl>,
#> #   breeding_success_criterion <chr>, country <chr>,
#> #   colony <chr>, colony_latitude <dbl>,
#> #   colony_longitude <dbl>, nest_id <chr>,
#> #   blood_sample <chr>, feather_sample <chr>,
#> #   other_samples <chr>, data_responsible <chr>,
#> #   back_on_nest <chr>, logger_mount_method <chr>,
#> #   comment <chr>, other <chr>, old_ring_number <lgl>
```

In this test case, the metadata file contains both deployment, retrieval and measurement info. We can see how many deployments and retrievals we have.

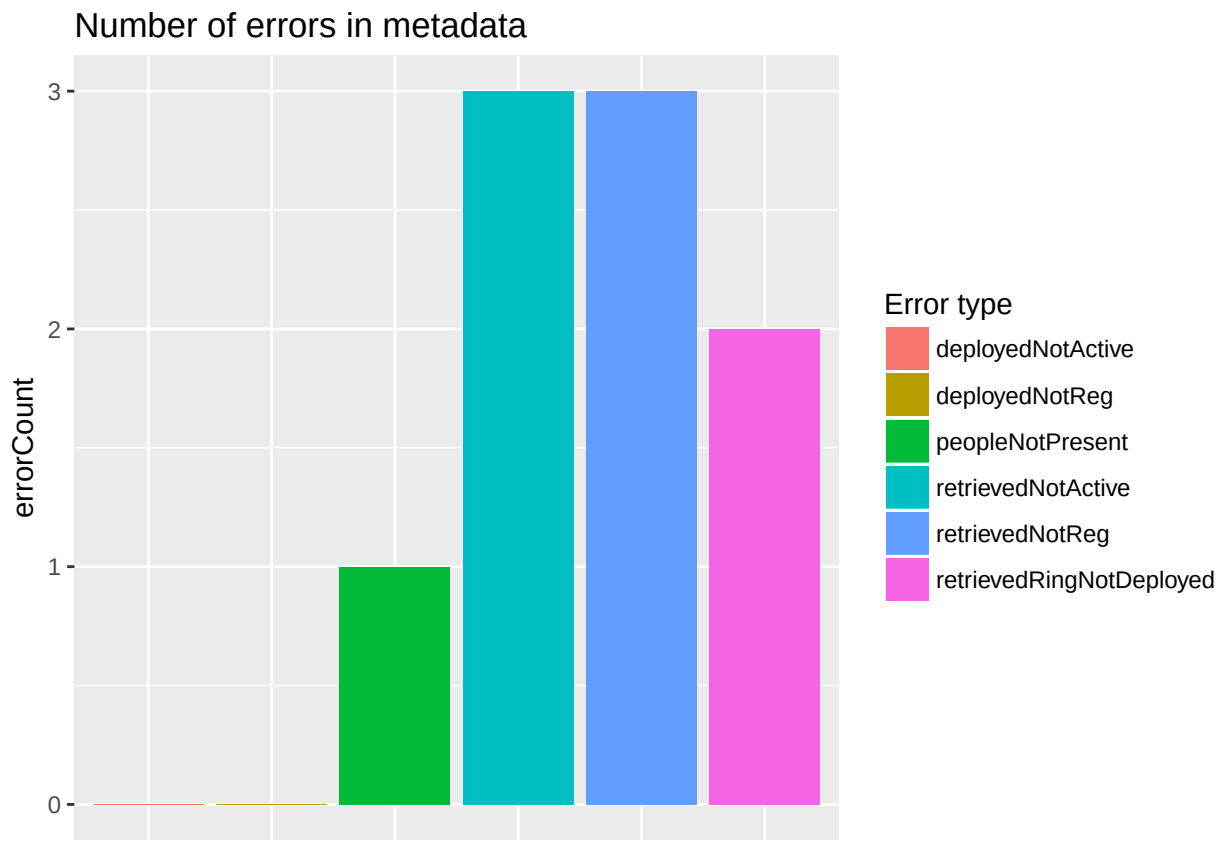
```
noDepRetr <- sampleMetadata %>% summarise(noDeployments = sum(!is.na(logger_id_deployed)),
    noRetrievals = sum(!is.na(logger_id_retrieved)))
noDepRetr
#> # A tibble: 1 x 2
#>   noDeployments noRetrievals
#>           <int>        <int>
#> 1            79           59
```

So, 79 deployment events, and 59 retrieval events are going to be registered in one go, by importing this data. Before we import the data, we can do some quality checks to find common errors. The import routine should stop in the event of important errors, but it can be tedious to step through these problems one by one. The function `checkMetadata` wraps several checking routines, see `?checkMetadata` for a list of all of them.

```
myErrors <- checkMetadata(sampleMetadata)
#> Errors found!
```

Looks like there are errors in our metadata! The object created by the function `checkMetadata` has a special class, with a print and plot function. The plot and summary function can be used to quickly get a quick look, the print function (just typing the object) shows all errors and some hints. Look at the `str(myErrors)` if you want to se the innards of the object.

```
plot(myErrors)
```




```
summary(myErrors)
#> # A tibble: 6 x 2
#>   reason                   errorCount
#>   <chr>                         <int>
#> 1 deployedNotActive                 0
#> 2 retrievedNotActive                3
#> 3 deployedNotReg                    0
#> 4 retrievedNotReg                   3
#> 5 peopleNotPresent                  1
#> 6 retrievedRingNotDeployed          2
```

```
myErrors
#>
#> These loggers are not in an open logging session, but metadata contains retrieval info.
#> Start a new logging session with writeLoggerImport(), and add deployment info before importing retri
#>   row_number logger_serial_no
#> 1        106     scraped off!
#> 2        107     scraped off!
#> 3        108     scraped off!
#>
#> These loggers are not registered in the table loggers.logger_info, but metadata contains retrieval i
#> Register the loggers with writeLoggerImport(), and add deployment info before importing retrieval in
#>   row_number logger_serial_no
#> 1        106     scraped off!
#> 2        107     scraped off!
#> 3        108     scraped off!
```

```
#>
#> These retrieved ring numbers don't match the ring numbers that where deployed on this logger.
#> Individ_id of NA means the deployment data is not yet in database.
#>   individ_id ring_number.y.y euring_code.y.y session_id
#> 1       <NA>         5175137             NOS          1
#> 2       <NA>         5175137             NOS         78
#>
#> These names are not in the table metadata.people. Check spelling and compare with getNames().
#>   row_number               name
#> 1         20 Alkekungen himself
```

In this case, it appears that the logger serial number was not readable for 3 loggers containing info on retrievals, and the field personnel noted this as "scraped off!". Naturally, this "serial number" is not registered in the logger_info table, and these loggers are not registered as in an open logging session.

In addition, someone has been having a bit of fun with the name of the data responsible on a record. This nickname is not registered in the metadata.people table.

We fix these errors and run another check.

```
ringsOfErrors <- sampleMetadata$ring_number[106:108]
sampleMetadata[sampleMetadata$ring_number %in% ringsOfErrors,
    ]
#> # A tibble: 7 x 39
#>   date       ring_number euring_code color_ring
#>   <date>     <chr>       <chr>       <chr>
#> 1 2016-01-07 5175137     NOS         <NA>
#> 2 2016-01-07 5175138     NOS         <NA>
#> 3 2016-04-07 5175141     NOS         <NA>
#> 4 2018-03-04 5175141     NOS         <NA>
#> 5 2017-01-06 5175137     NOS         <NA>
#> 6 2017-01-06 5175138     NOS         <NA>
#> 7 2017-04-07 5175141     NOS         <NA>
#> # ... with 35 more variables: logger_status <chr>,
#> #   logger_model_retrieved <chr>,
#> #   logger_id_retrieved <chr>, logger_model_deployed <chr>,
#> #   logger_id_deployed <chr>, species <chr>, morph <chr>,
#> #   subspecies <chr>, age <dbl>, sex <chr>,
#> #   sexing_method <chr>, weight <dbl>, scull <dbl>,
#> #   tarsus <dbl>, wing <dbl>, breeding_stage <chr>,
#> #   eggs <dbl>, chicks <dbl>, hatching_success <lgl>,
#> #   breeding_success <lgl>,
#> #   breeding_success_criterion <chr>, country <chr>,
#> #   colony <chr>, colony_latitude <dbl>,
#> #   colony_longitude <dbl>, nest_id <chr>,
#> #   blood_sample <chr>, feather_sample <chr>,
#> #   other_samples <chr>, data_responsible <chr>,
#> #   back_on_nest <chr>, logger_mount_method <chr>,
#> #   comment <chr>, other <chr>, old_ring_number <lgl>
```

Going by the data on the deployments, it seems that the missing logger serial numbers where "Z231", "Z236", and "Z234".

```
sampleMetadata$logger_id_retrieved[106:108] <- c("Z231", "Z236",
    "Z234")
```

```
sampleMetadata %>% select(date, colony, species, data_responsible) %>%
    filter(row_number() %in% 18:22)
#> # A tibble: 5 x 4
#>   date       colony  species       data_responsible
#>   <date>     <chr>   <chr>         <chr>
#> 1 2016-05-07 Sklinna European shag Svein-Håkon Lorentsen
#> 2 2016-06-06 Sklinna Herring gull  Svein-Håkon Lorentsen
#> 3 2016-06-06 Sklinna Herring gull  Alkekungen himself
#> 4 2016-06-06 Sklinna Herring gull  Svein-Håkon Lorentsen
#> 5 2016-06-06 Sklinna Herring gull  Svein-Håkon Lorentsen

## Looks like it should be Svein-Håkon
sampleMetadata$data_responsible[20] <- sampleMetadata$data_responsible[19]
```

Time for a new check of the data.

```
myErrors <- checkMetadata(sampleMetadata)
#> Errors found!
```

We still show some errors here because the checking functions can't yet handle multiple open sessions. To be fixed! Also, there is a true error here where the rings don't match. Thats better. Note however that these checks doesn't find every possible error. Please suggest further checks to put into this routine!

We can now import the metadata.

```
writeMetadata(sampleMetadata)
#> Warning in result_create(conn@ptr, statement): Closing open
#> result set, cancelling previous query
#> [1] TRUE
```

And check the new status of the number of deployed and retrieved loggers.

```
activeSessions <- getActiveSessions()


activeSessions %>% summarise(no_deployed = sum(!is.na(deployment_id)),
    no_retrieved = sum(!is.na(retrieval_id)))
#> # A tibble: 1 x 2
#>   no_deployed no_retrieved
#>         <int>        <int>
#> 1          79           56
```

We see that the logger_session table has been filled with data on deployments and retrievals. Data on the colony, species, and individ_id the logger was deployed on is also added to the table. The rows with retrieval data also contains data on the year tracked.

```
activeSessions %>% filter(!is.na(retrieval_id))
#> # A tibble: 56 x 12
#>    id       session_id logger_id deployment_id retrieval_id
#>    <chr>         <int>     <int>         <int>        <int>
#>  1 9af88c7~         78         1            78            1
#>  2 9aed833~          3         3             3            2
#>  3 9aede23~          4         4             4            3
#>  4 9aee427~          5         5             5            4
#>  5 9aeeaa4~          6         6             6            5
#>  6 9aeef5b~          7         7             7            6
```

```
#>  7 9aef185~          8         8             8            7
#>  8 9aef3b3~          9         9             9            8
#>  9 9aef5eb~         10        10            10            9
#> 10 9aef80e~         11        11            11           10
#> # ... with 46 more rows, and 7 more variables:
#> #   active <lgl>, colony <chr>, species <chr>,
#> #   year_tracked <chr>, individ_id <chr>,
#> #   last_updated <dttm>, updated_by <chr>
```

Importing shutdown information

We can now shut down the logging sessions that have been given retrieval data. We could also have shut down these logging sessions before, but we would then not be able to add deployment or retrieval data.

We use the logger_import table again to shut the logging sessions down. For all rows where shutdown_session = True, the corresponding logging sessions will be shut down. Remember that it is usually not a good idea to import startup and shutdown data at the same time, since this will just open and close the session. One way of only importing shutdown info is to blank out all the other columns in the logger import data.

Here the startup and allocation info is empty.

```
sampleLoggerShutdown
#> # A tibble: 59 x 22
#>    logger_serial_no logger_model producer production_year
#>    <chr>            <chr>        <lgl>    <lgl>
#>  1 Z231             c65          NA       NA
#>  2 Z236             c65          NA       NA
#>  3 Z234             c65          NA       NA
#>  4 Z232             c65          NA       NA
#>  5 Y604             f100         NA       NA
#>  6 Y612             f100         NA       NA
#>  7 Y614             f100         NA       NA
#>  8 Y595             f100         NA       NA
#>  9 Y116             c330         NA       NA
#> 10 Y099             c330         NA       NA
#> # ... with 49 more rows, and 18 more variables:
#> #   project <lgl>, starttime_gmt <date>,
#> #   logging_mode <lgl>, started_by <lgl>,
#> #   started_where <lgl>, days_delayed <lgl>,
#> #   programmed_gmt_time <lgl>, intended_species <lgl>,
#> #   intended_location <lgl>, intended_deployer <lgl>,
#> #   shutdown_session <lgl>, shutdown_date <date>,
#> #   field_status <chr>, downloaded_by <chr>,
#> #   download_type <chr>, download_date <date>,
#> #   decomissioned <lgl>, comment <lgl>
```

And we only have shutdown info.

```
sampleLoggerShutdown %>% select(logger_serial_no, logger_model,
    shutdown_session:comment)
#> # A tibble: 59 x 10
#>    logger_serial_no logger_model shutdown_session
#>    <chr>            <chr>        <lgl>
#>  1 Z231             c65          T
```

```
#>  2 Z236             c65          T
#>  3 Z234             c65          T
#>  4 Z232             c65          T
#>  5 Y604             f100         T
#>  6 Y612             f100         T
#>  7 Y614             f100         T
#>  8 Y595             f100         T
#>  9 Y116             c330         T
#> 10 Y099             c330         T
#> # ... with 49 more rows, and 7 more variables:
#> #   shutdown_date <date>, field_status <chr>,
#> #   downloaded_by <chr>, download_type <chr>,
#> #   download_date <date>, decomissioned <lgl>,
#> #   comment <lgl>
```

Remember that filenames will only be produced in the case when the `download_type` is either "Successfully downloaded" or "Reconstructed". Let's have a look at the types of downloads we are about to import.

```
downloadTypes <- sampleLoggerShutdown %>% group_by(download_type) %>%
    tally()
```

This means that we should get filenames from 40 of the 59 loggers.

We import this data similarly as with the startups.

```
writeLoggerImport(sampleLoggerShutdown)
#> Warning in result_create(conn@ptr, statement): Closing open
#> result set, cancelling previous query
#> [1] TRUE
```

This should have closed 59 sessions and so we should now have 20 still active sessions.

```
activeSessions <- getActiveSessions()
activeSessions
#> # A tibble: 20 x 12
#>    id       session_id logger_id deployment_id retrieval_id
#>    <chr>         <int>     <int>         <int>        <int>
#>  1 9af5ce8~         58        58            16           NA
#>  2 9af5f00~         59        59            17           NA
#>  3 9af6136~         60        60            18           NA
#>  4 9af634e~         61        61            24           NA
#>  5 9af6562~         62        62            28           NA
#>  6 9af677a~         63        63            29           NA
#>  7 9af69b5~         64        64            34           NA
#>  8 9af6bcb~         65        65            35           NA
#>  9 9af6dfe~         66        66            36           NA
#> 10 9af7016~         67        67            42           NA
#> 11 9af725b~         68        68            48           NA
#> 12 9af7494~         69        69            49           NA
#> 13 9af76b0~         70        70            59           NA
#> 14 9af78c6~         71        71            60           NA
#> 15 9af7ae0~         72        72            65           NA
#> 16 9af7d19~         73        73            66           NA
#> 17 9af7f39~         74        74            67           NA
#> 18 9af815c~         75        75            72           NA
#> 19 9af837b~         76        76            74           NA
```

```
#> 20 9af8602~         77        77            75           NA
#> # ... with 7 more variables: active <lgl>, colony <chr>,
#> #   species <chr>, year_tracked <chr>, individ_id <chr>,
#> #   last_updated <dttm>, updated_by <chr>
```

Looks good.

The shutdown also creates filenames associated with the session, depending on the make and model of the logger. These end up in the table loggers.file_archive.

Working with the file archive

We can now see what these shutdowns has produced in the file archive table. This table lists the expected filenames produced by the logging sessions that has been shutdown. It is up to the users to manually upload these files to the file archive location. The file archive is an FTP server running on the same machine as the seatrack database. We use the passwords in the database to connect to the FTP server, but this is handled through the functions in this package so that users do not have to enter their credentials once a seatrack connection has been made.

We can take a look at the expected filenames through to functions. Firstly, the function `getFileArchiveSummary` retrieves the info of the expected filenames, together with which logging session they are connected to and some info on the related birds.

```
databaseFileArchive <- getFileArchiveSummary()

databaseFileArchive
#> # A tibble: 257 x 9
#>    file_id session_id colony  ring_number euring_code
#>      <int>      <int> <chr>   <chr>       <chr>
#>  1       1          3 Sklinna 5175139     NOS
#>  2       2          3 Sklinna 5175139     NOS
#>  3       3          3 Sklinna 5175139     NOS
#>  4       4          3 Sklinna 5175139     NOS
#>  5       5          3 Sklinna 5175139     NOS
#>  6       6          3 Sklinna 5175139     NOS
#>  7       7          3 Sklinna 5175139     NOS
#>  8       8          4 Sklinna 5175140     NOS
#>  9       9          4 Sklinna 5175140     NOS
#> 10      10          4 Sklinna 5175140     NOS
#> # ... with 247 more rows, and 4 more variables:
#> #   year_tracked <chr>, logger_serial_no <chr>,
#> #   logger_model <chr>, filename <chr>
```

You could use this table to get some bookkeeping info. Currently, we have shut down 5 different logger models, some of which produces 7 and some that produces 4 files. For example see how the recorded file names group into individual logger models.

```
databaseFileArchive %>% group_by(logger_model, session_id) %>%
    tally() %>% group_by(logger_model) %>% summarise(mean(n))
#> # A tibble: 5 x 2
#>   logger_model `mean(n)`
#>   <chr>            <dbl>
#> 1 c250              7.00
#> 2 c330              7.00
```

```
#> 3 c65               7.00
#> 4 f100              7.00
#> 5 mk4083            4.00
```

Currently, we have shut down 5 different logger models, some of which produces 7 and some that produces 4 files, depending on their make and model. We could also from this table see how many loggers that have been shutdown and are expected to have files associated with them.

```
databaseFileArchive %>% summarise(noShutdownLoggers = n_distinct(logger_serial_no,
    logger_model))
#> # A tibble: 1 x 1
#>   noShutdownLoggers
#>               <int>
#> 1                38
```

So out of the 57 shutdowns we performed, only 40 of them resulted in files in the table loggers.file_archive. This is as predicted since only 40 was successfully downloaded or had their download data reconstructed.

Uploading and downloading files from the file storage.

So far, we have only looked in the database for the expected files connected to each logging session. The actual file storage is located on the FTP server. We can use the `listFileArchive` function to list the files in the file storage on this ftp server. This function also compares the the content of the file storage and to the proposed filenames in the database.

```
fileArchive <- listFileArchive()
fileArchive
#> $filesInArchive
#> # A tibble: 2 x 1
#>   filename
#>   <chr>
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
#>
#> $filesNotInArchive
#> # A tibble: 257 x 1
#>    filename
#>    <chr>
#>  1 Z234_2017_c65.sst
#>  2 Z234_2017_c65driftadj.trn
#>  3 Z234_2017_c65.trn
#>  4 Z234_2017_c65driftadj.lux
#>  5 Z234_2017_c65.lux
#>  6 Z234_2017_c65driftadj.deg
#>  7 Z234_2017_c65.deg
#>  8 Z232_2017_c65.sst
#>  9 Z232_2017_c65driftadj.trn
#> 10 Z232_2017_c65.trn
#> # ... with 247 more rows
#>
#> $filesNotInDatabase
#> # A tibble: 2 x 1
#>   filename
#>   <chr>
```

```
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
```

For the purpose of testing, we have uploaded some dummy files, called F630_2014_c65.sst, and F630_2014_c65driftadj.trn. These are found in the list element `filesInArchive` and since they are not expected by the database, also in the list element `filesNotInDatabase`. The summary also show `filesNotInArchive` which lists the expected files registered in the database, that are not yet sent to the file storage.

We can upload files to the storage, using the function `uploadFiles`. This function grabs the appropriate username and passwords for the ftp connection from the database and uploads the files specified. You need to specify `overwrite = True` to overwrite existing files. Only users that login to the database with write permissions (members of the role group "seatrack_writer") will be able to upload files to the file storage.

Here we will upload two test files that are locally stored in the folder "temp".

```
uploadFiles(c("test.txt", "test2.txt"), originFolder = "../temp")
#> [1] "File uploaded: ../temp/test.txt"
#> [2] "File uploaded: ../temp/test2.txt"
```

Although we get a confirmation, we can double check that the files actually are now stored in the file storage.

```
fileArchive <- listFileArchive()
fileArchive
#> $filesInArchive
#> # A tibble: 4 x 1
#>   filename
#>   <chr>
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
#> 3 test.txt
#> 4 test2.txt
#>
#> $filesNotInArchive
#> # A tibble: 257 x 1
#>    filename
#>    <chr>
#>  1 Z234_2017_c65.sst
#>  2 Z234_2017_c65driftadj.trn
#>  3 Z234_2017_c65.trn
#>  4 Z234_2017_c65driftadj.lux
#>  5 Z234_2017_c65.lux
#>  6 Z234_2017_c65driftadj.deg
#>  7 Z234_2017_c65.deg
#>  8 Z232_2017_c65.sst
#>  9 Z232_2017_c65driftadj.trn
#> 10 Z232_2017_c65.trn
#> # ... with 247 more rows
#>
#> $filesNotInDatabase
#> # A tibble: 4 x 1
#>   filename
#>   <chr>
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
#> 3 test.txt
```

```
#> 4 test2.txt
```

We can download files from the storage using the function `downloadFiles`. This is available for everyone that can login to the database (members of the group "seatrack_reader"). Here, we download all the files.

```
filesToGet = listFileArchive()$filesInArchive
downloadFiles(files = filesToGet, destFolder = "../temp", overwrite = T)
#> [1] "File downloaded: ../temp/F630_2014_c65.sst"
#> [2] "File downloaded: ../temp/F630_2014_c65driftadj.trn"
#> [3] "File downloaded: ../temp/test.txt"
#> [4] "File downloaded: ../temp/test2.txt"
```

More often though you would identify a subset of files to download. Which files you are interested in could be found through a custom SQL query, or some R code that searches through the `getFileArchiveSummary` output.

Deleting files from the FTP archive

In case a wrong file has been uploaded to the file archive, or for testing purposes, we may need to delete files from the file archive. This is done through the `deleteFiles` function, which requires write permissions in the database. This asks for confirmation if you don't specify force = True.

```
filesInArchive <- listFileArchive()$filesInArchive
filesInArchive
#> # A tibble: 4 x 1
#>   filename
#>   <chr>
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
#> 3 test.txt
#> 4 test2.txt

filesToDelete <- filesInArchive %>% filter(str_detect(filename,
    "test"))
filesToDelete
#> # A tibble: 2 x 1
#>   filename
#>   <chr>
#> 1 test.txt
#> 2 test2.txt

deleteFiles(files = filesToDelete, force = T)
#> [1] "File deleted: test.txt"  "File deleted: test2.txt"
```

We can double check that the test files are gone.

```
listFileArchive()$filesInArchive
#> # A tibble: 2 x 1
#>   filename
#>   <chr>
#> 1 F630_2014_c65.sst
#> 2 F630_2014_c65driftadj.trn
```