Data Delivery System v.1

The SciLifeLab Data Centre is currently developing the Data Delivery System[1]. The system is built to enable the simple and secure delivery of data from SciLifeLab facilities to their users. In this document we describe the system, how it currently works, and what features we are planning to add.

Terminology used throughout the document
(unless otherwise stated)

User	Institution, research group etc. using a facility's services. The project/data owner, and recipient.
Facility	SciLifeLab facility. Data producer. Delivers data to the user.

Table of Contents

[1] GitHub: https://github.com/ScilifelabDataCentre/Data-Delivery-System

1. What is the Data Delivery System?

The Data Delivery System uses the Safespring's[2] object-storage service as the delivery medium, thereby keeping the data within the Swedish borders. The Data Delivery System also has built-in encryption and key management, enabling protection of the data without the users being required to perform these steps themselves prior to delivery. The system is not a storage solution – the data will only be stored for a short period of time[3] and the recipients are themselves responsible for the future handling of their data.

The system is comprised of two components – a web interface and a command line interface (CLI). Both of the components will be used for project- and delivery tracking as described below, however they differ in the amount of data they will be able to deliver: the CLI can be used to deliver any number and size and the web interface will only be usable for delivering small data.

NOTE: This is <u>not</u> a finished product and development is ongoing.

2. The current version

This section describes the Data Delivery System and what each component can be used for at this time. Section 4 presents our plan for the system, including the features that will (and will not) be added during further development.

2.1. Web Interface

The web interface can currently be used for project tracking – viewing all projects that the user is involved in which has the Data Delivery System as its delivery option, and listing all uploaded files within that project. It cannot be used for uploading, downloading or deleting any files. A summary of possible actions is listed in the table below.

User type	**Facilities** (data producers)	**Users** (data recipients)
List (projects/data)	✓	✓
Upload	✗	✗
Download	✗	✗
Delete	✗	✗

[2] For more information on Safespring and their storage service: **https://www.safespring.com/en/services/storage/**
[3] The storage time has not yet been decided. At this time, during testing, this is not applicable.

2.2. Command Line Interface, CLI

In the current version of the system, the CLI can be used for uploading and downloading (depending on the type of user) directories or files of any type and size. It cannot be used for listing projects or files, or deleting the data. The table shows a list of possible actions.

User type	**Facilities** (data producers)	**Users** (data recipients)
List (projects/data)	✗	✗
Upload	✓	✗
Download	✗	✓
Delete	✗	✗

3. How do I use it?

This section provides a step-by-step description of how the current version of the Data Delivery System can be used by the different types of users.

3.1. Accounts

To use the Data Delivery System, each user needs an account. Each facility will have one type of account[4], users another. User accounts will be created when an order is made to a SciLifeLab facility, if the user does not already have an account and if the system is chosen as the data delivery method. During the first phase of testing, however, all accounts will be issued manually by the SciLifeLab Data Centre[5]. At this time only one account will be assigned per facility, and there will be only one user account per project. This means that only one account per project can upload data, and only one can download.

[4] Information on how and when facility accounts are created will be provided at a later time.
[5] **Contact: senthilkumar.panneerselvam@scilifelab.se** or **ina.oden.osterbo@scilifelab.uu.se**

3.2. Web Interface

The web interface is accessed by going to **https://dds.dckube.scilifelab.se/.** The home page will be displayed.

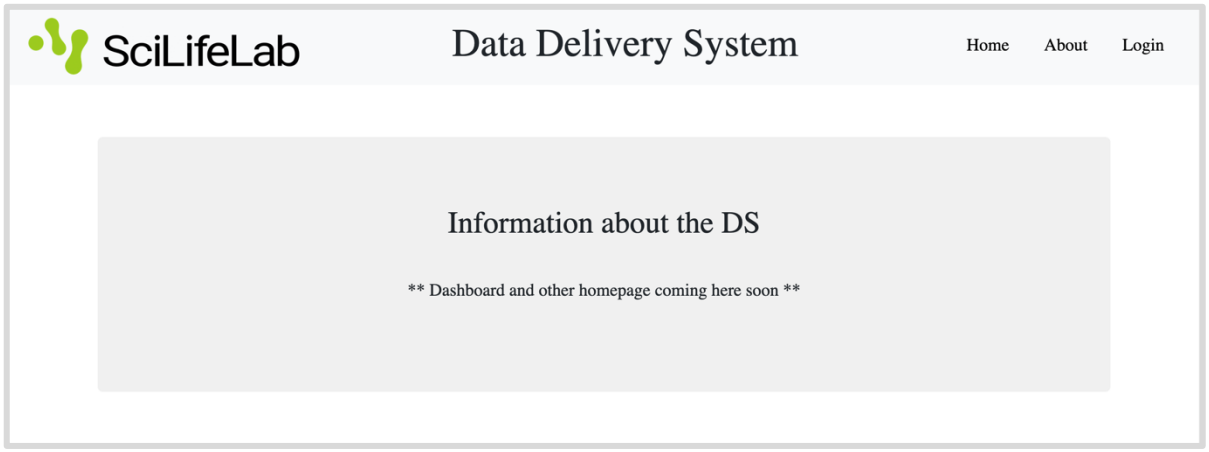


Figure 1. Data Delivery System web interface home page

Go to the login page by clicking “Login” in the top right corner. The following page will be displayed.

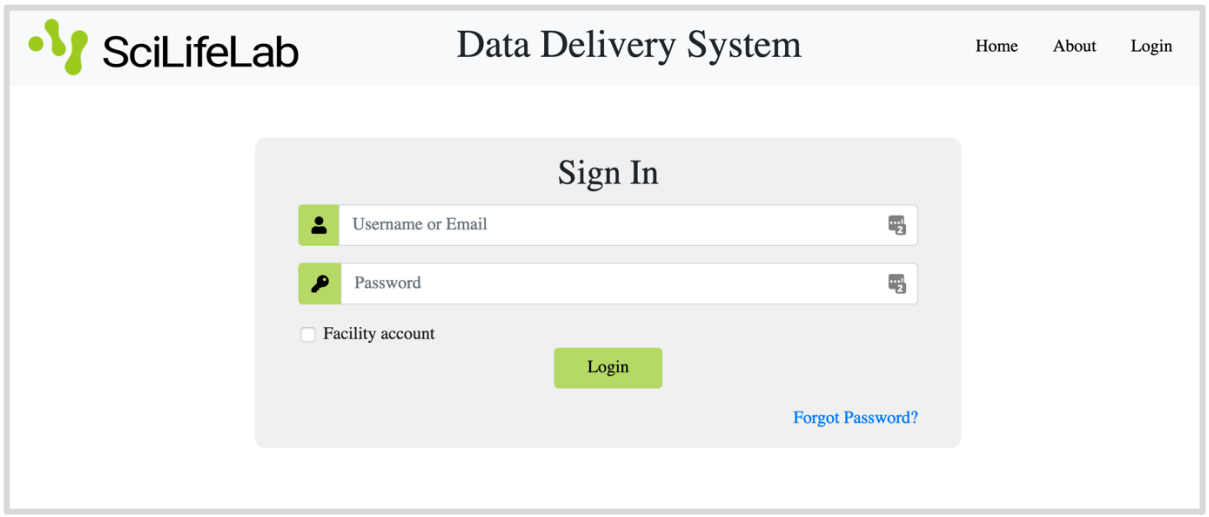


Figure 2. The login form

Users and facilities specify their login credentials and press Login. Facilities also click the “Facility account” box.

Facility (data producer)

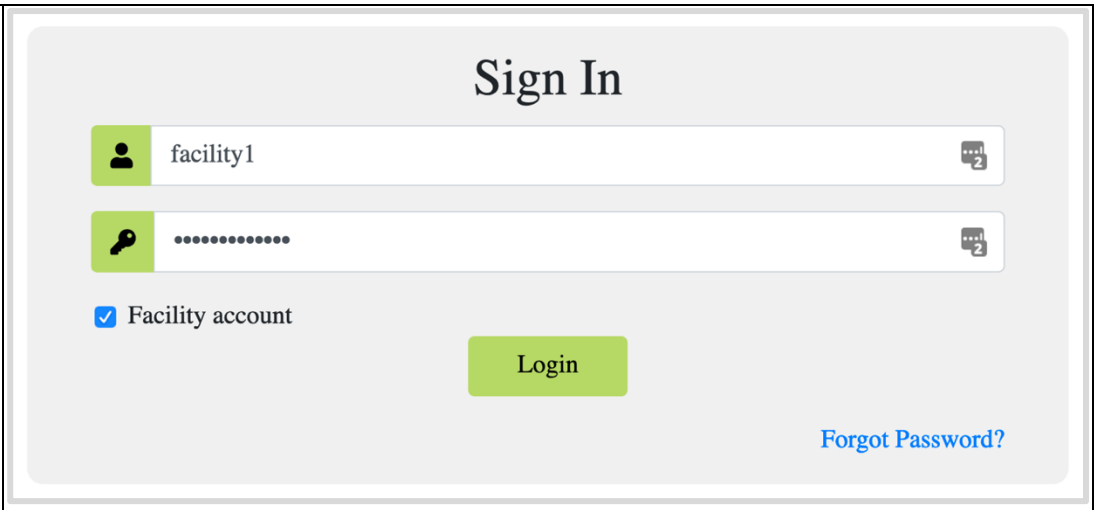


Figure 3. Facility (data producer) signing in

Other user (data recipient)

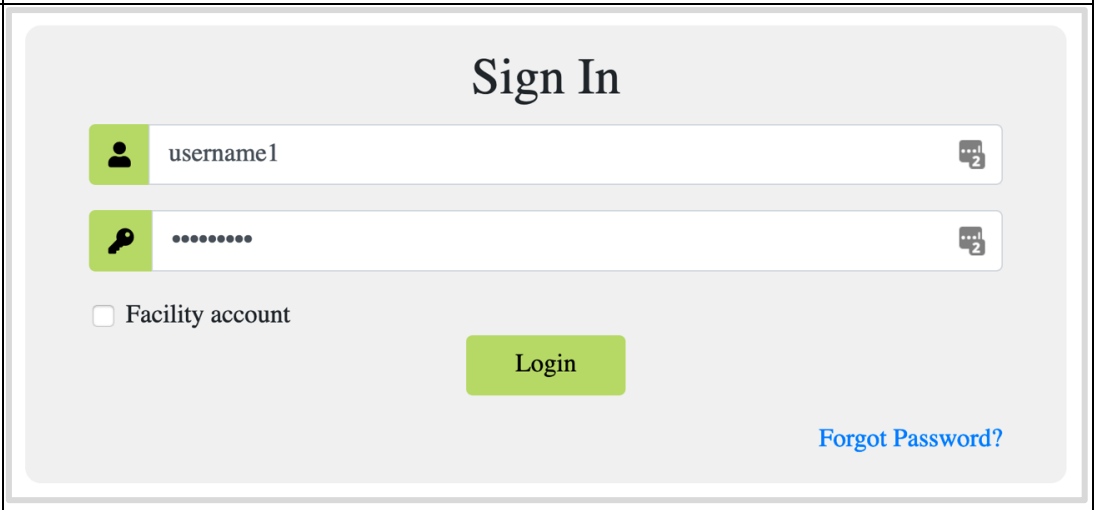


Figure 4. Other user (data owner/recipient) signing in

After logging in, the user will see a list of all projects that the user is currently involved in[6]. The list shows the project ID, the title of the project, what category it falls under (e.g. genomics, imaging etc.), the facility ID and what status the delivery has. The project in the example below has the status "Ongoing"[7], indicating that the facility has begun uploading data within the project.

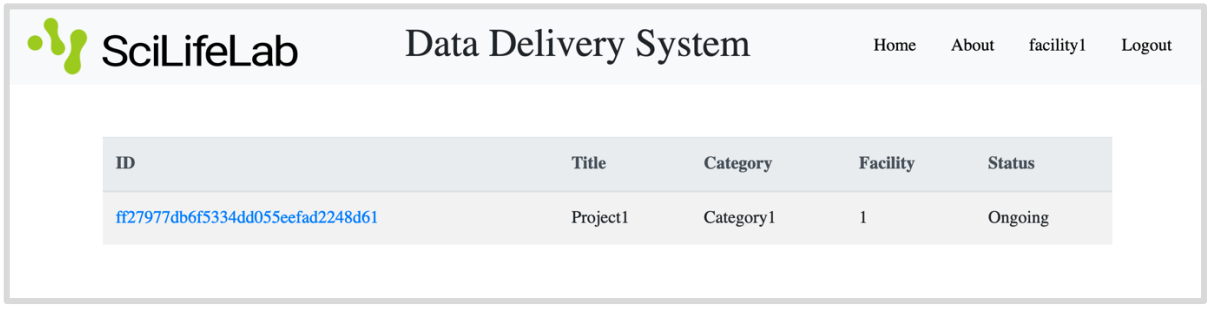


Figure 5. Page showed once logged in

[6] The page currently looks the same for all types of users. This may change during further development.
[7] In later versions, the facility will be able to mark the upload as "Finished", meaning that all data within a specific project has been uploaded.

Klick on the project ID to go to the project's home page. A page with the following structure will be displayed.

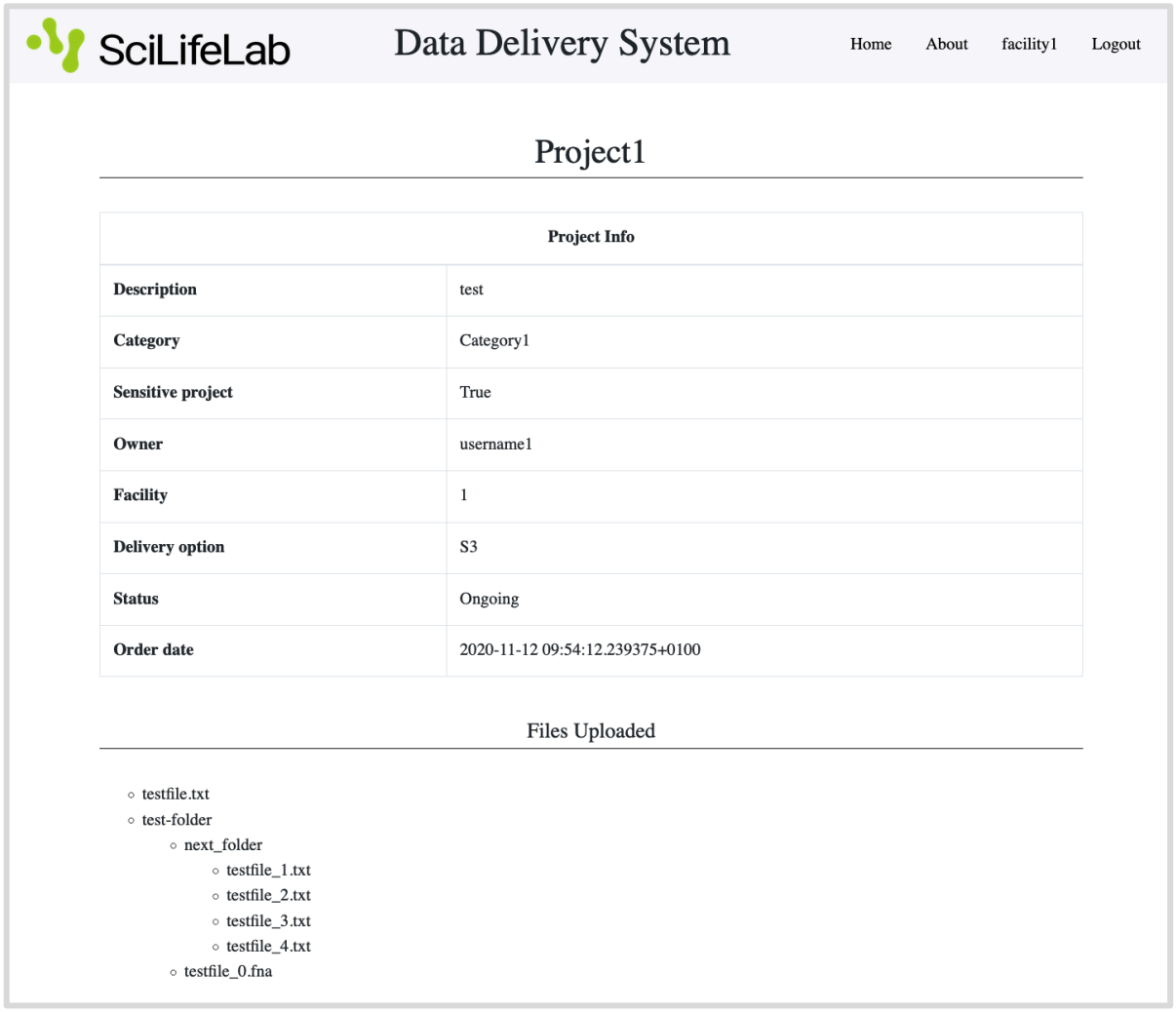


Figure 6. Project page. Information about project and files uploaded within it.

The page shows more detailed information about the project, such as the date on which the order was made, a description of the project, the chosen delivery option[8], etc. At the bottom of the page is a section called "Files Uploaded". This section lists all files which have been uploaded within the project. As seen in the example above, the facility has uploaded the file *testfile.txt* and the folder *test-folder*. The folder structure is maintained, as can be seen from the indented lines indicating subfolders and files within each folder.

Note: The structure and layout of the web interface is not finished. This is an initial version, focusing on the core functionalities. Section 4 describes the future development plans for the Data Delivery System.

[8] At this time, the only delivery option available is "S3". This indicates that the project owner has chosen Safespring Storage as the data delivery medium.

3.3. Command Line Interface, CLI

The CLI command is currently called `ds_deliver`.

3.3.1. Setup

Open a terminal and enter the following command:

```
python -m pip install
git+https://github.com/ScilifelabDataCentre/DS_CLI@
ff633df8e83c14b37eea86955b6db093bc402b3c
```

There should be no new lines. This should result the installation of the CLI and its requirements, and after that you should be able to use the Data Delivery System CLI.

3.3.2. Credentials

To upload or download a file you first need to specify your Data Delivery System user credentials. The table below shows the information required for the two operations.

	put	get
Username	✓	✓
Project ID	✓	✓
Project Owner	✓	✗

Note that the project (/data) owner is not specified by the user when downloading. This is due to that only one account can download, as explained in in *section 3.1*. The owner in this case is therefore the username belonging to the account downloading the data.

Information on the exact usage of the credentials is found in *section 3.3.4* and *3.3.5*.

3.3.3. Help

Get information on how to use the CLI by typing `ds_deliver --help` in the terminal:

```
$ ds_deliver --help
Usage: ds_deliver [OPTIONS] COMMAND [ARGS]...

  Data Delivery System CLI.

  Initiates the required delivery objects: A temporary directory where logs
  and files will be stored, and the CLI logger. Cannot be used independently
  - put or get must be specified as an argument.

Options:
  --help  Show this message and exit.

Commands:
  get  Handles the download of files from the project-specific S3 bucket.
  put  Handles the upload of files to the project-specific S3 bucket.
```

To get help regarding the CLI commands and their specific options, type…

- `ds_deliver put --help`

```
$ ds_deliver put --help
Usage: ds_deliver put [OPTIONS]

  Handles the upload of files to the project-specific S3 bucket.

  Currently only usable by facilities.

Options:
  -c, --creds PATH     Path to file containing user credentials - username,
                       password, project id, project owner.
  -u, --username TEXT  Your Data Delivery System username.
  -p, --project TEXT   Project ID to which the delivery belongs to.
  -o, --owner TEXT     Owner ID - the user to whom you are delivering.
  -f, --pathfile PATH  Path to file containing all files and folders to be
                       delivered.
  -s, --source PATH    Path to file or folder to be delivered.
  --break-on-fail      Failure to deliver one file results in cancellation of
                       all specified files.  [default: False]
  --overwrite          Replace any previously delivered files specified in the
                       current delivery.  [default: False]
  --help               Show this message and exit.
```

or

- `ds_deliver get -help`

```
$ ds_deliver get --help
Usage: ds_deliver get [OPTIONS]

  Handles the download of files from the project-specific S3 bucket.

  Currently not usable by facilities.

Options:
  -c, --creds PATH     Path to file containing user credentials - username,
                       password, project id, project owner.
  -u, --username TEXT  Your Data Delivery System username.
  -p, --project TEXT   Project ID to which the delivery belongs to.
  -f, --pathfile PATH  Path to file containing all files and folders to be
                       delivered.
  -s, --source TEXT    Path to file or folder to be delivered.
  --break-on-fail      Failure to deliver one file results in cancellation of
                       all specified files.  [default: False]
  --help               Show this message and exit.
```

3.3.4. put

In this section, the term ***delivery***[9] refers to the chain of operations from CLI command to the files being located in the Safespring Storage bucket. It does not include the download from the cloud to the data recipient. The reason for using this term as opposed to *upload* is to distinguish the full chain of operations (command → checks → processing → upload → database update) from the individual components. See section 0 for a detailed description of the operations included.

Credentials

You can either…

- …use the `--username`, `--project` and `--owner` options:

```
$ ds_deliver put --username <DDS Username> --project <Project ID> --owner <Project owner>
```

- …or use the `--creds` option and enter the path to a file with the credentials in it:

```
$ ds_deliver put --creds demo/creds_put.json
```

  The accepted format for this file is currently only JSON[10]. The file contents should have the following structure:

```
{
    "username": <DDS Username>,
    "project": <Project ID>,
    "owner": <Project owner>
}
```

The owner specified is the username of the project/data owner (the recipient).

The `--username`, `--project` and `--owner` options take precedence over the `--creds` option. For example, the command shown below will use the username <*ThisIsAnotherUsername*>, not <*DDS Username*>.

```
$ cat demo/creds_put.json
{
    "username": <DDS Username>,
    "project": <Project ID>,
    "owner": <Project owner>
}
$ ds_deliver put --creds demo/creds_put.json --username <ThisIsAnotherUsername>
```

Independent of which of these methods you use, the CLI will prompt you for the password. The password is hidden, you will not see the password as you type.

[9] Including its inflections.
[10] JSON file extension: .json

Data

You can enter the data you wish to deliver using two different options:

- `--source`:

File

```
$ ds_deliver put --creds demo/creds_put.json --source demo/testfile.txt
Password: 🔑
```

Folder

```
$ ds_deliver put --creds demo/creds_put.json --source demo/test-folder/
Password: 🔑
```

`--source` can be used multiple times in the same command.

- `--pathfile`:

```
$ ds_deliver put --creds demo/creds_put.json --pathfile demo/data_to_upload.txt
Password: 🔑
```

The file *data_to_upload.txt* has the following content structure:

```
/Absolute/Path/To/demo/testfile.txt
/Absolute/Path/To/demo/test-folder/testfile_0.fna
/Absolute/Path/To/demo/test-folder/next_folder/testfile_1.txt
/Absolute/Path/To/demo/test-folder/next_folder/testfile_2.txt
/Absolute/Path/To/demo/test-folder/next_folder/testfile_3.txt
/Absolute/Path/To/demo/test-folder/next_folder/testfile_4.txt
```

You can specify data using both options in the same command.

Running the command

When the CLI command is run, a table containing each file name, its current delivery status[11] and upload percentage, is shown. The following example shows the finished delivery of the folder */demo/test-folder/* contents.

```
$ ds_deliver put --creds demo/creds_put.json --source demo/test-folder/
Password:

          File      Status      Upload/Download Progress
 testfile_0.fna        ✅                         100.00%
 testfile_3.txt        ✅                         100.00%
 testfile_2.txt        ✅                         100.00%
 testfile_1.txt        ✅                         100.00%
 testfile_4.txt        ✅                         100.00%
```

If any errors occur, the status is changed to a red cross. An example of when this will happen is mentioned in the *Flags* section below.

When the delivery (either successfully or not) is finished, the CLI produces a "Delivery Report". The full example of delivering the folder */demo/test-folder/* is shown below.

[11] Not to be confused with the *Status* seen in the project page on the web interface.

```
$ ds_deliver put --creds demo/creds_put.json --source demo/test-folder/
Password:

          File      Status      Upload/Download Progress
 testfile_0.fna        ✅                          100.00%
 testfile_3.txt        ✅                          100.00%
 testfile_2.txt        ✅                          100.00%
 testfile_1.txt        ✅                          100.00%
 testfile_4.txt        ✅                          100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *


- - - - - - - - - - - - - Folders - - - - - - - - - - - - -

Folder:                          Absolute/Path/To/                          /demo/test-folder
Files attempted: 5       Files uploaded: 5.


A detailed list of uploaded user-specified data can be found
in the delivery log file, located in the directory:
                    Absolute/Path/To/Current/Working/Directory                    /DataDelivery_2020-11-11_16-16-07/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The Delivery Report shows:

- The user-specified (absolute) path to the folder
- The number of files within that folder (*Files attempted*)
- The number of files successfully uploaded (*Files uploaded*)
- Information on where to find the log file created by the CLI.

The output for files is slightly different. It does show the number of specified files and how many of those were successfully uploaded, and the location of the log file, but it does not show a list of all files. Also, it specifies the location of the files at the destination.

```
$ ds_deliver put --creds demo/creds_put.json --source demo/testfile.txt
Password:

         File      Status      Upload/Download Progress
 testfile.txt        ✅                          100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *

- - - - - - - Files (not located in directory) - - - - - - -

Files attempted: 1      Files uploaded: 1

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Location of uploaded files:      Root directory of the projects Safespring bucket.

A detailed list of uploaded user-specified data can be found
in the delivery log file, located in the directory:
                    Absolute/Path/To/Current/Working/Directory                    /DataDelivery_2020-11-11_15-26-46/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Flags

Note that the two flags can be used in the same CLI call.

--overwrite

Once you have delivered a file, attempting to deliver a file with the same name again will lead to the CLI cancelling the delivery for that file:

```
$ ds_deliver put --creds demo/creds_put.json --source demo/testfile.txt
Password:

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *

- - - - - - - Files (not located in directory) - - - - - - -

Files attempted: 1      Files uploaded: 0
Failed files:
+-------------------------------------------------+-----------+---------------------------------+
| File                                            | Delivered | Error                           |
+-------------------------------------------------+-----------+---------------------------------+
|                  Absolute/Path/To/              |     NO    |  File already exists in database |
| /demo/testfile.txt                              |           |                                 |
|                                                 |           |                                 |
+-------------------------------------------------+-----------+---------------------------------+

A detailed list of uploaded user-specified data can be found
in the delivery log file, located in the directory:
          Absolute/Path/To/Current/Working/Directory          /DataDelivery_2020-11-11_17-15-14/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

To execute the delivery again, and thereby overwrite the file[12] already located in the Safespring Storage, you can specify the `--overwrite` option:

```
$ ds_deliver put --creds demo/creds_put.json --source demo/testfile.txt --overwrite
Password:

         File      Status      Upload/Download Progress
 testfile.txt        ✅                        100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *

- - - - - - - Files (not located in directory) - - - - - - -

Files attempted: 1      Files uploaded: 1

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Location of uploaded files:      Root directory of the projects Safespring bucket.

A detailed list of uploaded user-specified data can be found
in the delivery log file, located in the directory:
          Absolute/Path/To/Current/Working/Directory          /DataDelivery_2020-11-11_17-17-55/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

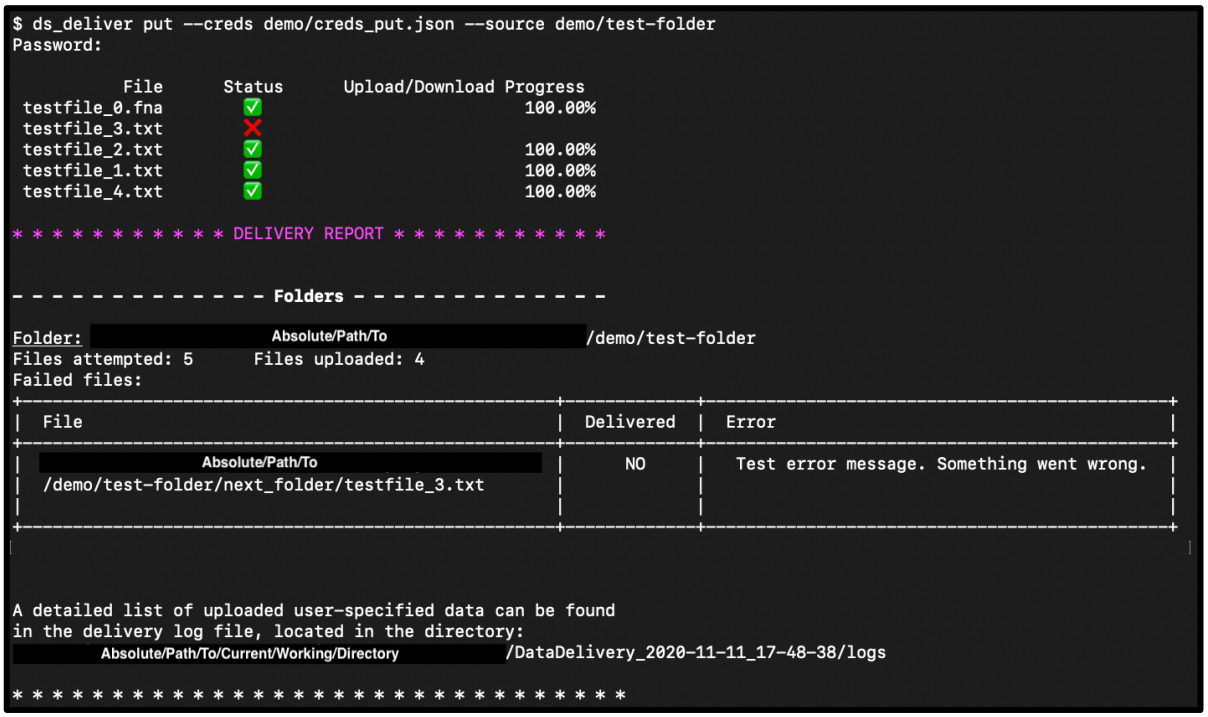
--break-on-fail

In the event of an error during delivery (be it during the initial checks, the processing or the upload), the CLI only cancels the execution for the file(s) which produced the error. If the user has specified multiple files, the upload of the remaining files continues. This may be useful in cases where, for

[12] This function applies to folders as well.

example, a folder has been previously delivered but there are new files within it which have not. The example below shows:

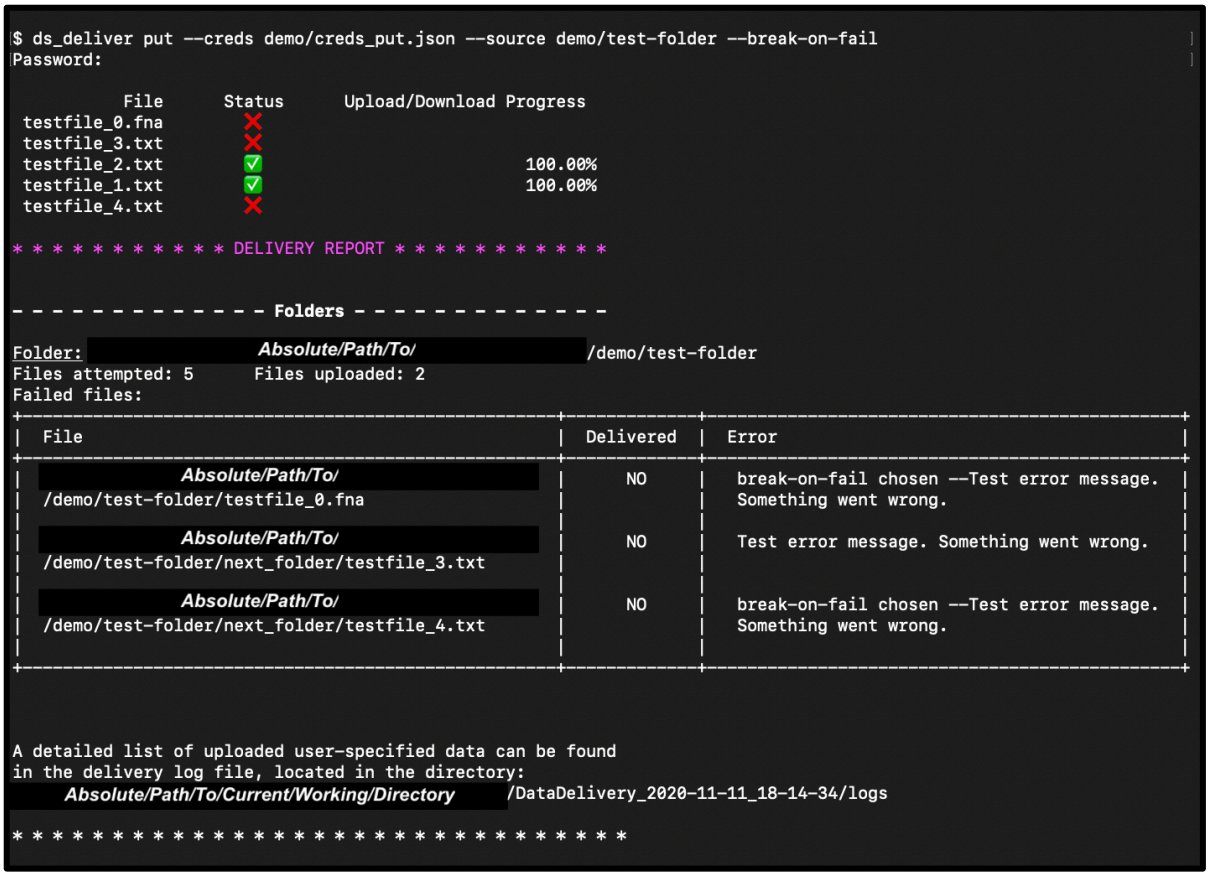
- The file *demo/test-folder/next_folder/testfile_3.txt* fails at some point during the delivery[13].
- All other files continue to be delivered.



The `--break-on-fail` flag, however, allows you to cancel the delivery of all files if one fails. The example below shows:

- The file *demo/test-folder/next_folder/testfile_3.txt* fails at some point during the delivery[13].
- The delivery of two files were cancelled due to the error with *testfile_3.txt*.
- Two files were successfully delivered before *testfile_3.txt* failed[14]. If the upload of these files were schedules to be started after the failure of *testfile_3.txt*, these would have also been cancelled.

[13] The error message displayed is an example. The error messages will show what has gone wrong, not "something".
[14] Cancelling the upload of a file after it has begun poses a problem, partly due to that the upload is handled in separate threads for each file. *See Appendix A. How does the CLI work?* for an explanation on how the system works.



Note: The error messages produced by the Data Delivery System will be improved before a stable version of the system is released.

3.3.5. get [15]

In this section, the term ***data collection***[9] refers to the chain of operations from CLI command to the decrypted files being located in the DataDelivery folder created by the CLI. It does not include the upload from the data recipient to the cloud. The reason for using this term as opposed to *download* is to distinguish the full chain of operations (command → checks → download → processing → database update) from the individual components. See section 0 for a detailed description of the operations included.

Credentials

The project owner is not an option for the get command. In all other ways, the options work in the same manner. You can either…

- …use the `-username` and `--project` options:

```
$ ds_deliver get --username <DDS Username> --project <Project ID>
```

- …or use the `--creds` option and enter the path to a file with the credentials in it:

```
$ ds_deliver get --creds demo/creds_get.json
```

The accepted format for this file is currently only JSON. The file contents should have the following structure:

```
{
    "username": <DDS Username>,
    "project": <Project ID>
}
```

The `-username` and `--project` options take precedence over the `--creds` option. See *section 3.3.4 (subsection Credentials)* for an example of this. Independent of which of these methods you use, the CLI will prompt you for the password. The password is hidden, you will not see the password as you type.

Data

The data you wish to collect is specified in a similar way as with put, however the paths entered are the relative location of the files within the project-specific Safespring bucket. To see a list of files uploaded within a specific project, go to the Data Delivery System web interface. The following image shows a screenshot of the list of files uploaded within a test project, as shown by the web interface.

Figure 7. Screenshot of the files uploaded within a project

[15] This section gives similar information as *section 3.3.4* (put), with some small differences. These differences may however cause confusion in some cases, and therefore put and get are described in different sections.

To collect the file *testfile.txt* you would specify that alone, to collect the file *testfile_0.fna* you would specify the path *test-folder/testfile_0.fna*, to collect the entire *test-folder* you would simply specify *test-folder*, and so on.

You can enter the data you wish to collect using two different options.

- `--source`:

File	`$ ds_deliver get --creds demo/creds_get.json --source testfile.txt` `Password: 🔑`
Folder	`$ ds_deliver get --creds demo/creds_get.json --source test-folder` `Password: 🔑`

  `--source` can be used multiple times in the same command.

- `--pathfile`:

```
$ ds_deliver get --creds demo/creds_get.json --pathfile demo/data_to_download.txt
Password: 🔑
```

  The file *data_to_download.txt* has the following content structure:

```
testfile.txt
test-folder/testfile_0.fna
test-folder/next_folder/testfile_1.txt
test-folder/next_folder/testfile_2.txt
test-folder/next_folder/testfile_3.txt
test-folder/next_folder/testfile_4.txt
```

You can specify data using both options in the same command.

Running the command

When the CLI command is run, a table containing each file name, its current processing- or download status and download percentage, is shown. The following example shows the finished collection of the folder *test-folder* contents.

```
$ ds_deliver get --creds demo/creds_get.json --source test-folder
Password:

          File      Status      Upload/Download Progress
 testfile_1.txt        ✅                        100.00%
 testfile_2.txt        ✅                        100.00%
 testfile_3.txt        ✅                        100.00%
 testfile_4.txt        ✅                        100.00%
 testfile_0.fna        ✅                        100.00%
```

If any errors occur, the status is changed to a red cross. An example of when this will happen is mentioned in the *Flags* section below.

When the data collection (either successfully or not) is finished, the CLI produces a “Delivery Report”. The full example of collecting the folder *test-folder* is shown below.

```
$ ds_deliver get --creds demo/creds_get.json --source test-folder
Password:

          File      Status      Upload/Download Progress
 testfile_1.txt        ✅                        100.00%
 testfile_2.txt        ✅                        100.00%
 testfile_3.txt        ✅                        100.00%
 testfile_4.txt        ✅                        100.00%
 testfile_0.fna        ✅                        100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *


- - - - - - - - - - - - - Folders - - - - - - - - - - - - -

Folder: test-folder
Files attempted: 5       Files downloaded: 5.


A detailed list of downloaded user-specified data can be found
in the delivery log file, located in the directory:
                    Absolute/Path/To/Current/Working/Directory                    /DataDelivery_2020-11-11_21-49-38/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The Delivery Report shows:

- The user-specified path to the folder (relative path within Safespring Storage bucket, as described above)
- The number of files within that folder (*Files attempted*)
- The number of files successfully collected (*Files downloaded*)
- Information on where to find the log file created by the CLI.

The output for files is slightly different. Below is the full example and output from the CLI when collecting the file *testfile.txt*. The Delivery Report does not list all user-specified files. Instead, it shows:

- The number of files specified (*Files attempted*)
- The number of files successfully collected (*Files downloaded*)
- The location of files at the destination[16]. More information on this in *section 4.3.2*.
- Information on where to find the log file created by the CLI.

```
$ ds_deliver get --creds demo/creds_get.json --source testfile.txt
Password:

         File      Status      Upload/Download Progress
 testfile.txt        ✅                        100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * *

- - - - - - - Files (not located in directory) - - - - - - -

Files attempted: 1      Files downloaded: 1

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Location of downloaded files:      Absolute/Path/To/Current/Working/Directory      /DataDeliver
y_2020-11-11_21-46-47/files.

A detailed list of downloaded user-specified data can be found
in the delivery log file, located in the directory:
                    Absolute/Path/To/Current/Working/Directory                    /DataDelivery_2020-11-11_21-46-47/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

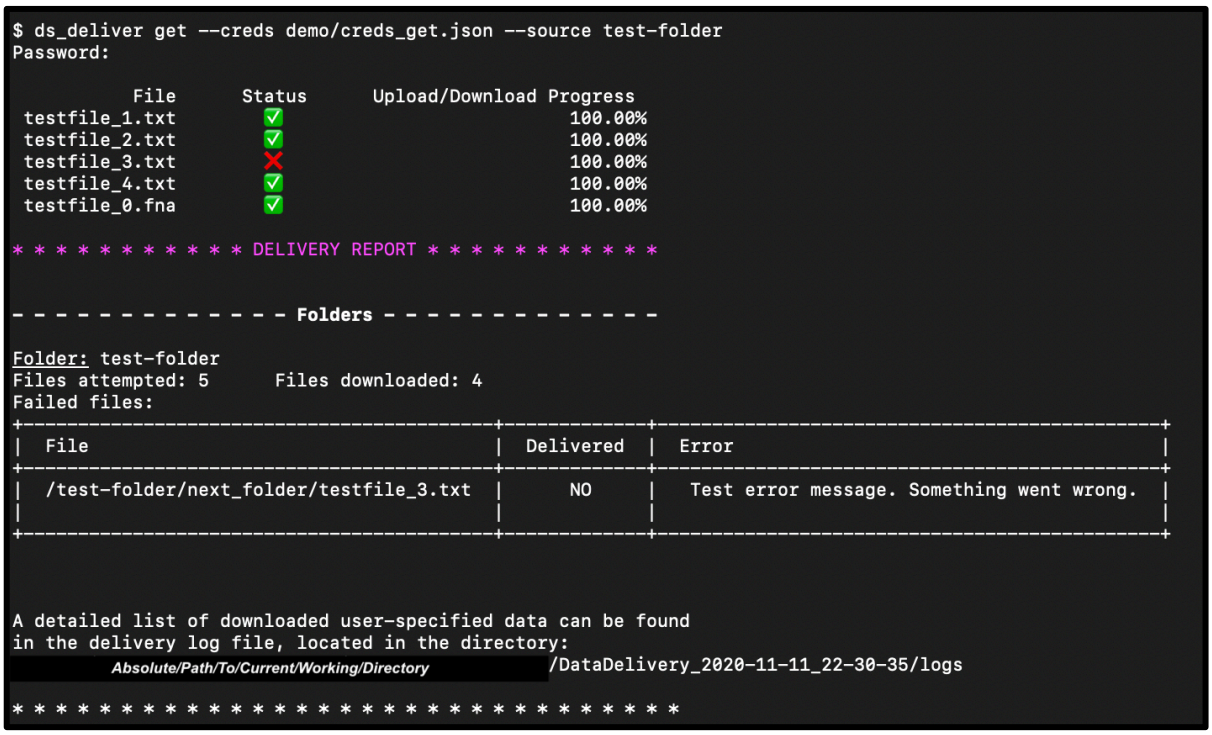
[16] This information will also be added for folders during future development.

Flags

`get` only allows for one flag: `--break-on-fail`. `--overwrite` may be added later when specifying the destination is added as an option (see *section 4.3*), however at this time all collected files are placed in a time stamped Data Delivery folder, created when the CLI is called.

--break-on-fail

In the event of an error during data collection (be it the initial checks, download or the file processing) the CLI only cancels the execution for the file(s) which produced the error. If the user has specified multiple files, the collection of the remaining files continues. This may be useful in cases where a previous data collection needs to be continued.



The example above shows:

- The file *demo/test-folder/next_folder/testfile_3.txt* fails at some point during the data collection[13].
- All other files continue to be collected.

The `--break-on-fail` flag, however, allows you to cancel the collection of all files if one fails. Files which have already been downloaded and started decryption will continue. The example below shows:

- The file */test-folder/next_folder/testfile_1.txt* fails at some point
- The collection of all other files are cancelled due to the error with *testfile_1.txt*

```
$ ds_deliver get --creds demo/creds_get.json --source test-folder --break-on-fail
Password:

          File      Status      Upload/Download Progress
 testfile_1.txt        ❌                        100.00%
 testfile_2.txt        ❌                        100.00%
 testfile_3.txt        ❌                        100.00%
 testfile_4.txt        ❌                        100.00%
 testfile_0.fna        ❌                        100.00%

* * * * * * * * * * DELIVERY REPORT * * * * * * * * * * *


- - - - - - - - - - - - - Folders - - - - - - - - - - - - -

Folder: test-folder
Files attempted: 5      Files downloaded: 0
Failed files:
+----------------------------------------+-------------+-----------------------------------------+
|  File                                  |  Delivered  |  Error                                  |
+----------------------------------------+-------------+-----------------------------------------+
|  /test-folder/next_folder/testfile_1.txt  |      NO     |   Test error message. Something went wrong.   |
|                                        |             |                                         |
|  /test-folder/next_folder/testfile_2.txt  |      NO     |   break-on-fail chosen --Test error message.  |
|                                        |             |   Something went wrong.                 |
|                                        |             |                                         |
|  /test-folder/next_folder/testfile_3.txt  |      NO     |   break-on-fail chosen --Test error message.  |
|                                        |             |   Something went wrong.                 |
|                                        |             |                                         |
|  /test-folder/next_folder/testfile_4.txt  |      NO     |   break-on-fail chosen --Test error message.  |
|                                        |             |   Something went wrong.                 |
|                                        |             |                                         |
|  /test-folder/testfile_0.fna           |      NO     |   break-on-fail chosen --Test error message.  |
|                                        |             |   Something went wrong.                 |
|                                        |             |                                         |
+----------------------------------------+-------------+-----------------------------------------+



A detailed list of downloaded user-specified data can be found
in the delivery log file, located in the directory:
       Absolute/Path/To/Current/Working/Directory        /DataDelivery_2020-11-12_08-19-19/logs

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

4. Future plans for the Data Delivery System

4.1. General

More accounts

As mentioned, the current version of the Data Delivery System allows for one account per facility and one account per user. This layout is far from ideal since there may be several individuals within a facility that should be able to deliver data to users in one project, and the users are often in the form of a research group or institution where more than one individual wants access to the uploaded data. Due to this, we will be adding the possibility of more accounts per group.

Sharing data

We are looking into the possibility of allowing all account types to upload and download data, thereby enabling sharing both within and between groups. This is not a first step, but instead a possible enhancement once the system is up and running.

Pipe to compute or storage

Since the Data Delivery System is not a storage solution, download by the users will need to be followed by the users moving their data to their own storage solution. In addition, some type of computation will often be performed on the data produced by the facilities. Therefore, choosing a destination for the data is a feature that we would like to add to the system. The idea in this scenario is that the facility uploads the data as usual, and the user is then able to decide whether the data should be downloaded or if it should be piped to cloud computation services or storage solutions.

4.2. Web Interface

The first goal for the Data Delivery System web interface is the possibility to upload and download data. Facilities will (as in the CLI) be able to upload, while users will be able to download. The web interface will only support the delivery of small data. For large files and collections, the CLI will need to be used. Deleting data is also a required operation and is high on the priority list. See the list below for a detailed specification on what each party will be able to do in the web interface.

User type	**Facilities** (data producers)	**Other users** (data recipients)
List (projects/data)	✓	✓
Upload	✓ (small data)[17]	✗ [18]
Download	✗	✓ (small data)[4]
Delete	✓	✓

[17] The maximum size possible to upload/download while using the web interface is not yet determined. More information on this will be provided at a later time.
[18] As mentioned previously, enabling upload and download by both facilities and users is a possible feature in the future.

At this point, all accounts will be manually issued by the SciLifeLab Data Centre. However, the web interface will support the registering of new accounts in the future[19].

4.3. Command Line Interface, CLI

4.3.1. Operations

The first operations to be added to the CLI are…

- **List** – All types of accounts will all be able to get a list of all projects that they are using the Data Delivery System for, and all files and collections connected to those projects.
- **Delete** – Facilities will be able to delete data which they have uploaded. Users will be able to delete data that they own.

User type	Facilities (data producers)	Other users (data recipients)
List (projects/data)	✓	✓
Upload	✓	✗
Download	✗	✓
Delete	✓	✓

4.3.2. General improvements and features

Delivery report format

The current CLI generates a delivery report after each upload and download. The delivery report is in a format that is easily read by humans, but is difficult to parse computationally. We are planning to alter the report format, to add a second output, or to add the possibility to choose the output format.

Source and destination

During upload of a folder, the path structure is maintained. Files which are uploaded separately however (not together with their parent directory) will end up in the root of the Safespring bucket. For example:

Local path/Source	Path at destination (Safespring)

`/Absolute/Path/To/Some/Folder/`
`- file1.txt`
`- file2.txt`
`- file3.txt`
	`./Folder/`
	`- file1.txt`
	`- file2.txt`
	`- file3.txt`

`/Absolute/Path/To/Some/Folder/WithA/file.txt`	`./file.txt`

[19] The exact process for how this will work is not yet determined. Some accounts will still require approval by the Data Centre. More information on this will be provided at a later time.

In the first example above, we want to upload the folder `Folder` and all its contents. After upload, `Folder` is located in the root of the Safespring bucket. Within `Folder` are all its subfolders and files.

During download however, the downloaded folders and files are saved in a directory with the format: `DataDelivery_<date>_<time>`. Folder structure is still maintained.

The ability to specify both the source and destination of the file, thereby having the ability of choosing whether separate files or folders should be placed in the root or in a specific folder at the destination, will likely be implemented in the future.

5. Guidelines and Information

This is an Alpha Testing phase. Due to this, we would like to inform the testers of the following:

- Before you start testing, you need to get an account from us.
- We will provide you with two accounts – one will be able to use for uploads, and one for downloads. It's up to you how you want to handle the downloading-account i.e. if you want to perform these tests as well or if you want to include another node or one of your users in the process.
- This is an initial version of the Data Delivery System. The system can and will change.
- Do NOT deliver sensitive information. We do not accept any responsibility or guarantee that the system works as intended at this point.
- You can test the system using any type of file.
- You can upload any number of files and folders.
- You cannot upload files or folders larger than 700 GB during this initial testing.

The Alpha Testing will go on for a approximately one month from the day you get access to the system. This period could be shorter, depending on how the testing goes. Our idea is to have regular contact with you during this month and answer any questions you have and discuss any thoughts you have regarding the Data Delivery System.

5.1. Testing protocol

Since this is the first testing phase where we want your opinion on *everything*, it's difficult to specify a detailed testing protocol. However, the general procedure should be the following:

Facilities (data producers)	**Users (data recipients)**
• Check your access to the account and test which commands the system allows you to perform • Test the **upload** of the data using the different options	• Check your access to the account and test which commands the system allows you to perform • Test the **download** of the data using the different options

Please make notes of aspects of the system that:

- Aspects of the system that **work/don't work**
  - If it doesn't work – in what way? What is the error message, what was the expected output or result?
- Features that you **like/don't like** (and why)
- Features that you feel are missing from the system

Feel free to give us your feedback during meetings or via email[5].

Also, things to think about and give feedback on could be:

- **Web Interface**
  - **Projects & files** – Does it list them properly?
- **CLI**
  - **Options –** Do they work? Are the defaults reasonable?
  - **Output** – Is it easy to understand?
  - **File** – Try different file formats and sizes.
- **General**
  - Is there any information missing that would make anything more understandable?
  - Is there any feature you are missing in the system, apart from those mentioned in section 4?

Thank you for helping us!
We hope that the Data Delivery System can be useful for you.

6. Questions & Answers

- Are there any caps on usage?

During testing: You can upload any type and number of files/folders, but a maximum of 700 GB per session. If you test the CLI three times, you could upload 3*700 GB.

After release: No.

- What is the charging model?

Once the Data Delivery System is released, the facilities using it will be receive invoices from the SciLifeLab Data Centre. The facilities are themselves responsible for charging their users.

- What will it cost?

During testing: There will of course be no costs for the facilities involved in the testing.

After release: You will pay for the amount of storage space (short-term) that you use. An exact figure is not yet determined[20], and this is something we want to include in the discussions during the testing phase and come up with a good cost model together with those involved. We are also discussing the possibility of having a "free quota" for every facility, so that you would only pay annually for volumes above a specific threshold.

[20] Currently the cost for Safespring storage is 70 kr / TB / month. What it will cost for you is not determined.

Appendices

Appendix A. How does the CLI work?

A.1. Access check/Login

Component	Steps
User	**1.** The user specifies the CLI command (link here to how to use it).
CLI	**2.** A Data Delivery System directory is created[21]. The directory serves as a collection for Data Delivery System logs, information, and file processing intermediates. The directory name has the format `DataDelivery_<date>_<time>` and the following subdirectories: • `files` – File processing intermediates. • `logs` – Log files. • `meta` – Other information. **3.** A `log` file is created in the logs subdirectory. This file records information about the Data Delivery process. **4.** The CLI asks for the user's Data Delivery System password. **5.** The CLI checks that enough user credentials have been entered and cancels the upload if any required information is missing. **6.** An HTTP Request[22] is made to the Data Delivery System REST API[23].
REST API	**7.** The user credentials are checked for access to the Data Delivery System. The user is granted access if: • There is a user with the specified username • The password is correct[24] **8.** If the user is granted access to the Data Delivery System, the user's access to the specified project is checked. Access to the project (and thereby the ability to upload/download data) is granted if: **put** • The facility is listed as the data producer for the specified project • The specified project owner is correct • The project has S3 as the data delivery option[25] **get** • The user is listed as the project owner • The project has S3 as the data delivery option[11] **9.** If the user is granted access to the specified project, an access token is generated. At this time, the "token" is a unique randomly generated string[26]. The string is saved in the database, together with two time stamps (time of token creation and time of token expiration) and the project ID. This token and the functionality around it will be improved in several aspects before a stable version of the Data Delivery System is released. **10.** The following information is returned to the CLI:

[21] The folder is created in the current working directory – the location where the CLI command is called.
[22] From now on only called *request*.
[23] From now on only called *API*.
[24] For information on the password authentication, see *Appendix B. Data Delivery System DecisionsB.3. Password verification: Scrypt*.
[25] The project user may have chosen another delivery route. This option is not implemented.
[26] `os.urandom(16)` is used to generate 16 bytes, which are then converted to a hex string.

	• User granted delivery access: Yes or No • Public key[27] • Token[28] • Error message[29]
CLI	**11.** The request response is checked for errors, and the upload is cancelled if any error occurred in the preceding steps.

[27] The **public key** is used when generating the encryption- and decryption key. See section *Appendix C. The components, C.4. The keys*.
[28] The token is saved throughout the CLI session and passed as an argument in all subsequent steps.
[29] The error message should be empty if access is granted.

A.2. Upload

The term ***delivery***[9] is in this section used as a collective word for the full chain of operations from CLI command to the files being located in the Safespring Storage bucket: initial checks, processing, upload, database update. The term ***processing*** is in turn used as a collective word for compression and encryption.

Once the access has been granted (see *Appendix A. How does the CLI work?, A.1. Access check/Login*), the CLI performs a series of checks and operations on the data specified by the user. The steps are described in the table below. The description assumes that the default (see *section 3.3.4*) options are used, and that no errors occur.

Component	Steps
CLI	1. A request is made to the API to get a list of all files previously delivered within the specified project.
REST API	2. The token is validated[30]. 3. If the token is valid, the names of all files previously delivered within the project are retrieved from the database and returned to the CLI.
CLI	4. For all paths specified by the user, the CLI first checks if the path points to a file or a directory on the local file system. If the path points to a directory, the CLI gets a list of all files within that directory, including those within subdirectories. 5. A check is performed on all files[31] to see if they are in a compressed format (e.g. GZIP) or not[32]. Information about which files have been previously compressed is saved. 6. All file names that were returned from the API in step 3 are checked against the files specified by the user. Those that match have been previously delivered and are thereby cancelled[33], unless the --overwrite option is added to the CLI call. 7. A progress table is created, showing the file name, what step[34] the CLI is on, and (when the upload has begun) the upload-percentage. 8. The CLI creates a pool of processes, enabling the files to be processed in parallel. Each file is processed in the following way: 8.1. A new ECDH[36] Key Pair is generated. This key pair consists of a public and a private component. 8.2. An encryption key (later used for the encryption of the file) is generated (see section *Appendix C. The components, C.4. The keys*). Essentially, the key generation combines: • The public key from step 10 *in A.1. Access check/Login* and… • The components from step 8.1. above to generate the key.

[30] **Token valid if***:* 1. There is a matching token. 2. It's registered to the correct project ID. 3. It has not expired.
[31] Both individually listed and those within a specified directory.
[32] Section *Appendix C. The components, C.5. Compression check* explains this process and lists the compression formats that the Data Delivery System can recognize at this time.
[33] **Cancelled:** Removed from the list of files which will be further processed and uploaded.
[34] **Steps:** "Waiting to start", "Encrypting", "Uploading" and ✔ ("Finished")
[36] **ECDH:** Elliptic-Curve Diffie-Hellman. See *section Appendix C. The components, C.1. Elliptic-Curve Diffie-Hellman (ECDH) Key Exchange* for more information.

	8.3. Files that were found to be previously compressed (checked in step 5) move on to step 8.4. Files which are not in a compressed format go through the compression algorithm Zstandard[37]. 8.4. The file is encrypted with the algorithm ChaCha20-Poly1305[38]. 8.5. The encrypted file is (temporarily) saved in the directory[39]: `DataDelivery_<date>_<time>/files/` 8.6. The following information is saved in a session variable: • The location and size (in bytes) of the encrypted file • Whether or not the file was compressed within the Data Delivery System • The public key component f the ECDH Key Pair in step 8.1. • Salt[41] used in the generation of the encryption key in step 8.2. • Error message – if there is one. 9. The CLI creates a pool of threads[42], enabling the files to be uploaded asynchronously. When encryption is finished for a specific file, the upload of that file begins: 9.1. A request is made to the API to get the information[43] required for the connection to Safespring S3.
REST API	9.2. The token is validated[30]. 9.3. The API retrieves the following information and returns it to the CLI[44]: • Endpoint URL – The location of Safespring storage. • Project[45] name – Currently the same for all facilities involved in testing. • Keys (access and secret).
CLI	9.4. The information returned from the API is used to connect to the Safespring project. 9.5. The encrypted file is uploaded to the bucket. The bucket has the same name as the project ID. See *section 4.3.2* for an example and explanation of the file destination. During the upload, the progress percentage is continuously updated. 10. When upload finishes for a specific file, a request is made to the API to add the file to the database.
REST API	11. The token is validated[30].

[37] Why Zstandard? See *Appendix B. Data Delivery System Decisions, B.4. Compression Algorithm: Zstandard*.
[38] Why ChaCha1305-Poly1305? See *Appendix B. Data Delivery System Decisions, B.6. Encryption Algorithm: ChaCha20*.
[39] As described in *section 4.3.2*, the folder structure is maintained.
[41] **Salt:** Random bytes used to increase the randomness within (in this case) the key and thereby make it more cryptographically secure.
[42] **Thread:** Part of a process, runs in a shared memory space. Best suited for I/O related operations. **Process:** Runs in a separate memory space. Best suited for CPU intensive operations.
[43] This information is currently the same for all facilities. Before the release of a stable version of the Data Delivery System, this will be changed so that each facility uploads and downloads data to its own private Safespring S3 project.
[44] At this time, the information is stored in a file on the server, and is returned (apart from using a secure transfer) without extra security measures (e.g. signing) in place. This is, of course, high on the priority list and will be fixed before any stable versions are released.
[45] NOT the same as the Data Delivery System project ID.

	12. The API adds the file information to the database. This information includes the original file name and it's size in bytes, the project ID of which the file belongs to, and the points listed in step 8.6 13. The API returns a message to the CLI that the file information has been saved.
CLI	14. The encrypted file, located within the `DataDelivery_<date>_<time>/files/` directory, is deleted[46]. 15. Upload of the file is marked as finished.

[46] Deletion of the encrypted files is performed once the upload and database update of that specific file is completed. If does not wait for all files to be completed. This statement is true for all steps listed above.

A.3. Download

The term ***data collection*** is in this section used as a collective word for the full chain of operations from CLI command to the files being located in the local folder created by the CLI: initial checks, download, processing, database update. The term ***processing*** is in turn used as a collective word for decryption and decompression.

Once the access has been granted (see *Appendix A. How does the CLI work?, A.1. Access check/Login*), the CLI performs a series of checks and operations on the data specified by the user. The steps are described in the table below. The description assumes that the default (see *section 3.3*) options are used, and that no errors occur.

<table>
<tr><th>Component</th><th>Steps</th></tr>
<tr><td>CLI</td><td>1. A request is made to the API to get a list of all files previously uploaded within the specified project.</td></tr>
<tr><td>REST API</td><td>2. The token is validated[30].
3. If the token is valid, the names of all files previously uploaded within the project are retrieved from the database and returned to the CLI.</td></tr>
<tr><td>CLI</td><td>4. The CLI checks if the string specified by the user match any of the paths returned by the API.
4.1. First it checks if the string fully matches any path. If so, the string specified by the user is a file.
4.2. If there is no match, the “root directory” for that path is checked: If this starts with the string specified by the user, it’s considered to be a folder and all it’s contents are chosen to be collected.
The table below shows an example of the user specified string (<code>--source</code>), database contents, and whether or not there is a match and the file (File name) will be collected.
<table>
<tr><th>--source</th><th>File name</th><th>Root directory</th><th>Match?</th></tr>
<tr><td rowspan="2">testfile.txt</td><td>testfile.txt</td><td>-[47]</td><td>✓</td></tr>
<tr><td>test-folder/testfile.txt</td><td>test-folder</td><td>✗</td></tr>
<tr><td rowspan="2">test-folder</td><td>test-folder/testfile.txt</td><td>test-folder</td><td>✓</td></tr>
<tr><td>test-folder.txt[48]</td><td>-</td><td>✗</td></tr>
</table>
5. Files which don’t match any file names in the database are cancelled – they have not been uploaded and thus cannot be downloaded. For the paths that do match and will be downloaded, the following information is saved in a session variable:
• Whether or not the the files were compressed by the Data Delivery System during upload (or before that)
• The public key (A.2. Upload step 8.1.)
• Salt used in the generation of the encryption key (A.2. Upload step 8.2.)
• Error message – if there is one.</td></tr>
</table>

[47] “-“ indicates that there is no root directory recorded for the file – it’s a file in the root directory, and in the root of the Safespring bucket.
[48] test-folder.txt is here a file called test-folder.

	6. A request is made to the API to get the project specific private key[49]
REST API	7. The token is validated[30]. 8. The API retrieves the following information[50] and returns it to the CLI: • Private key • Salt • Nonce • The project-specific passphrase.
CLI	9. The private key returned from the API is decrypted and validated in the following procedure: 9.1. A key is derived from the salt and passphrase returned in step 8. 9.2. The derived key is used to decrypt the private key. 9.3. The decrypted key is validated. It is valid if it consists of the following format: `Data Delivery System "signature" + Project ID + Private key` Thus, only the last part of the "private key" is the actual key used. 10. A progress table is created, showing the file name, what step[34] the CLI is on, and (when the download has begun) the download-percentage. 11. The CLI creates a pool of threads[42], enabling the files to be downloaded asynchronously. Each file is downloaded in the following way: 11.1. A request is made to the API to get information[43] required for the connection to Safespring S3.
REST API	11.2. The token is validated[30]. 11.3. The API retrieves the following information and returns it to the CLI[44]: • Endpoint URL – The location of Safespring storage. • Project[45] name – Currently the same for all facilities involved in testing. • Keys (access and secret).
CLI	11.4. The information returned from the API is used to connect to the Safespring project. 11.5. The file is downloaded from the bucket, to the directory `DataDelivery_<date>_<time>/files/`. During the download, the progress percentage is continuously updated. 12. The CLI creates a pool or processes, enabling he files to be processed in parallel. Once the download of a specific file finishes, the processing of that file begins: 12.1. The encryption/decryption key is generated (see *Appendix C. The components, C.4. The keys*). Essentially, the key generation combines: • The file-specific public key retrieved from the API in step 5 (and generated in step 8.1. in *A.2. Upload*) • The project-specific public key retrieved from the API in step 10 in *A.1. Access check/Login* and… • The project-specific private key[51] retrieved from the API in step 8 above. 12.2. The file is decrypted with the algorithm ChaCha20-Poly1305[38].

[49] The private key is generated when the order is created/project information is added to the Data Delivery System. See *footnote 50*.
[50] Sectio*n Appendix C. The components, C.4. The keys* explains the use of these components.
[51] Why were the project-specific public and private keys not retrieved at the same time? These keys will not be stored in the same location since they together (in addition to the other public key) enables a user to decrypt all data within that project.

	12.3. Files that were not compressed by the Data Delivery System during upload (checked in step 5) move on to step 12.4. Files that were compressed during upload are decompressed with the algorithm Zstandard[37]. 13. When the processing finishes for a specific file, a request is made to the API to update the delivery date.
REST API	14. The token is validated[30]. 15. The API creates a time stamp and adds it as a date and time of delivery to the file information in the database. 16. The API returns a message to the CLI that the time stamp has been saved.
CLI	17. The collected file, located within the `DataDelivery_<date>_<time>/files/` directory, is deleted. 18. The data collection is marked as finished and the decrypted file can be accessed in the directory mentioned in step 17.

Appendix B. Data Delivery System Decisions

This appendix provides a description of the different technical decisions made so far during the development of the Data Delivery System. The main section headers are named according to the topic discussed and what the final decision was. Thus their format is the following: `<Topic>: <Decision> (- <Sub-decision>).` The section contents vary slightly but have a general structure conforming to:

- The alternatives - If there were two or three specific options to choose from, these are listed.
- Comparisons – Lists or descriptions of the advantages and disadvantages of the different options.
- Conclusion – Final comment about which option was chosen and why.

B.1. Framework: Flask

The alternatives

- Tornado
- Flask

Comparisons

	FLASK	TORNADO
ADVANTAGES	• The top growing Python framework • Plenty of online resources available • Flexible • Simple • REST Support via extensions • Integrated testing system – improved stability and quality of web applications	• Asynchronous – enables developers to handle many (thousands) concurrent connections • Fast • Flexible • Simple • Built-in support for user authentication • Possible to implement third-party authentication and authorization schemes
DISADVANTAGES	• Many extensions no longer supported • No build-in support for user authentication, but can be included using extensions • Efficiency can be effected by extensions	• Online resources not as many as for Flask • No built in support for REST API and the user needs to implement it manually

Conclusion

Since Flask is flexible and simple, extensions provide a large variety of functionalities including REST API support, it has an integrated testing system and there is more online support than for Tornado, **Flask** was chosen as the better option for the Data Delivery System framework. The built-in asynchronicity in Tornado is not an important feature since the system will not be used by thousands of users at a given time.

B.2. Database: Relational – MariaDB

The initial database used in the beginning of the development was the non-relational CouchDB. As development progressed, it became of interest to investigate whether this was the best approach or whether a relational database was better for the systems purposes.

The alternatives

- Relational database
- Non-relational database

Comparisons

RELATIONAL	NON-RELATIONAL

• Structured – represent and store data in tables
• Use SQL – Structured Querying Language
• Allows you to link information from different tables
• Can handle a lot of complex queries and database transactions
• Are good…
  o If the data structure is not changing often and you have structured data
  o At keeping data transactions secure
• Highly scalable
	• Unstructured – represent data in collections of JSON documents
	• Easy to use and flexible
	• Efficient
	• Scalable
	• Ideal for organizations and applications with a lot of data

Conclusion

The main motivation behind choosing a **relational database** is that we are forced to keep it structured and that it, when used in the correct way, can be more efficient than a non-relational database.

MariaDB was chosen (rather than e.g. MySQL) as the relational database because of it's query performance and that it provides many useful features not available in other relational databases.

B.3. Password verification: Scrypt

Since the Data Delivery System is aimed at handing large amounts of sensitive information, it's important that the access to the system is controlled and validated in a secure manner. Simply hashing the passwords and saving the results is certainly not enough to guarantee this level of security, not is salting the passwords before hashing. To handle the user accounts passwords, the algorithms `scrypt`[52], `bcrypt`[53] and `Argon2`[54] were discussed. These also include steps such as salting etc. however they include additional steps, making them more resistant to multiple types of attacks, and are considered to be secure key derivation functions.

The alternatives

- Scrypt
- Bcrypt
- Argon2

Comparisons

The following information is stated on the websites linked at the bottom of this page.

SCRYPT	BCRYPT	ARGON2
• Strong cryptographic key-derivation function (KDF) • Memory-intensive • Designed to prevent GPU, ASIC and FPGA attacks	• Secure KDF • Older than Scrypt • Less resistant to ASIC and GPU attacks • Uses constant memory → easier to crack	• Highly secure KDF function • ASIC- and GPU-resistant • Better password cracking resistance than Bcrypt and Scrypt
"It is considered that Scrypt is more secure than Bcrypt, so modern applications should prefer Scrypt (or Argon2) instead of Bcrypt."		"In the general case Argon2 is recommended over Scrypt [and] Bcrypt […]"

Conclusion

In the current version of the Data Delivery System, **Scrypt** is used via the Python `cryptography` package. The plan is to change to **Argon2**, provided that the package mentioned in the link below[54] is deemed to be appropriate, or that an alternative Python package is found. This has not been investigated yet since Scrypt is considered to provide a high level of security and focus has been on developing other key parts of the system.

Scrypt is, as mentioned, memory-intensive, meaning that the cost is high to perform the key derivation. This is not an issue when performing the operation once (as in normal login procedures), however in the case of hackers

[52] Scrypt: **https://tools.ietf.org/html/rfc7914.html**, **https://wizardforcel.gitbooks.io/practical-cryptography-for-developers-book/content/mac-and-key-derivation/scrypt.html**
[53] Bcrypt: **https://wizardforcel.gitbooks.io/practical-cryptography-for-developers-book/content/mac-and-key-derivation/bcrypt.html**
[54] Argon2: **https://wizardforcel.gitbooks.io/practical-cryptography-for-developers-book/content/mac-and-key-derivation/argon2.html**

attempting to crack the password, the high amount of memory (and time depending on the *n*-parameter value) required results in the hacking becoming very costly. Thus the goal of the algorithm is to make hacking the passwords as inconvenient and impractical as possible.

The Scrypt parameters and the chosen values for them are listed in the table below. The default values are used.

PARAMETER	DESCRIPTION[52]	VALUE
n[55]	Number of iterations	16384
r	Block size	8
p	Threads to run in parallel	1
password	The password to protect	*No default value, the users password*
salt	Securely generated random bytes	*No default value, generated by:* `os.urandom(16)`[56]
derived-key-length	Output length wanted	32 (bytes)

The parameters affect the memory- and CPU usage. The memory required is calculated in the following way:

$$Memory\ requied = (128 \times n \times r \times p)\ bytes$$

All parameters apart from the *password* is saved to the database. This does not compromise security. During user authentication, the parameters are retrieved from the database and used to test the specified password. If the resulting password hash matches, the password is correct.

[55] Must be a factor of 2 (2^x)
[56] Value is in number of bytes. 16 is minimum. Less is not advised for security reasons. This value may be increased in later versions of the system.

B.4. Compression Algorithm: Zstandard

Gzip (GNU zip[57]) is one of the most popular compression algorithms and is suitable for compression of data streams, is supported by all browsers and comes as a standard in all major web servers. However, while gzip provides a good compression ratio[58], it is very slow compared to other algorithms. Since there is a variety of different compression algorithms it became unreasonable to test every alternative. Instead, prior knowledge about the algorithm Zstandard[59] lead to the choice of comparing gzip with Zstandard.

The alternatives

- Gzip
- Zstandard

Comparisons

To test the usefulness of the two options, they were tested in the processing chain together with the chosen encryption algorithm. The encryption speed using ChaCha20-Poly1305 (in mentioned case tested on a 109 MB file) is around 600 MB/s[60], but when adding compression as a preceding step, the speed was less than 3 MB/s and the compression ratio 3,25. Since the delivery system will be dealing with huge files, it's important that the processing is efficient, and therefore that the chosen algorithms are fast. Due to this, Zstandard was tested with the same chunk size, resulting in a speed of 119 MB/s and a compression ratio of 3,1.

Conclusion

Since Zstandard gave approximately the same compression ratio in a fraction of the time**, Zstandard** was chosen as the algorithm to be implemented within the Data Delivery System.

[57] Gzip website: **https://www.gnu.org/software/gzip/**
[58] Compression ratio = original/compressed size
[59] Zstandard information: **https://facebook.github.io/zstd/**
[60] **Computer specs:** Processor – 2,7 GHz Quad-Core Intel Core i7 (macOS Catalina 10.15.7). RAM – 8GB.

B.5. Encryption Process: Local

The alternatives

- Locally
- In-transit

Comparison and motivation

The optimal way of delivering the data to the owners would be to perform encryption- and decryption in-transit, thereby decreasing the amount of memory required locally and possibly the total delivery time. However, there have been some difficulties finding a working solution for this, including that the crypt4gh Python package used in the beginning of the development did not support it.

On further investigation and contact with Safespring, we learned:

- Server-side encryption (and server-side stored keys) is technically possible on Safespring S3 storage **BUT** Safespring has chosen to not activate that function.
- Most of the S3- and Boto clients that Safespring uses, e.g. the bash cli `s3cmd`, goes through GPG[61] which performs the encryption before uploading the files. GPG/PGP makes it possible to encrypt using one key and decrypt using one or more other keys. This enables a more automated process but does not simplify for us or contribute with anything useful to the delivery system.
- All users of the Safespring backup service perform encryption on their own and handle the keys themselves.

Conclusion

The encryption will be performed **locally** before upload to the S3 storage.

[61] **GPG:** Gnu Privacy Guard, based off of OpenPGP where PGP stands for Pretty Good Privacy

B.6. Encryption Algorithm: ChaCha20

The alternatives

- Crypt4GH (description of format here)
- ChaCha20
- AES-GCM

Comparisons

Characteristics

Crypt4GH	Crypt4GH is a file *format*, not an algorithm as the other two alternatives. The files are read in 64 KiB[62] blocks. Each block is encrypted with the algorithm ChaCha20 and a 16 byte message authentication code (MAC) is generated from the ciphertext using the authentication algorithm Poly1305. These are saved together with a 12 byte nonce, a randomly generated *number used only once*. Each data block thus looks like this: `Nonce - Encrypted Data - MAC` The data encryption is performed using a randomly generated, 32 byte, key. The data encryption key is itself encrypted with ChaCha20and saved in the files header. This encryption is performed using a 32 byte key generated from the Elliptic-Curve Diffie-Hellman key exchange algorithm, where the following combinations (R = recipient, S = sender) produce identical keys – a shared key: • R public key + R private key + S public key • S public key + S private key + R public key The senders public key is saved within the files header, which is later used by the recipient to generate the shared key, decrypt the data key, and finally decrypt the data blocks. More detailed information on the Crypt4GH file structure and algorithms can be found here (link?). The header can include multiple header packets, thereby enabling access to the file by multiple users.
AES-256-GCM	The Advanced Encryption Standard, AES, is a block cipher which combines the core algorithm with a mode of operation - techniques on how to process sequences of data blocks. In this case the mode is the Galois Counter Mode (GCM), an Authenticated Encryption with Associated Data (AEAD) mode. Block ciphers perform operations on blocks of bits, not individual bits. The block size for AES is 128 bits (16 bytes). The message size limit for AES-GCM - the amount of data that can be securely encrypted using the same key- and nonce pair - is ~64 GiB. AES is very fast on dedicated hardware.

[62] KiB = Kibibyte. Not to be confused with KB = Kilobyte. 1 Kilobyte (KB) = 1000 bytes, 1 Kibibyte (KiB) = 1024 bytes.

ChaCha20-Poly1305	ChaCha20-Poly1305 is an Authenticated Encryption with Associated Data (AEAD) algorithm, and combines the encryption algorithm ChaCha20 with the authentication algorithm Poly1305. Note: This is the algorithm used within the Crypt4GH file format. ChaCha20 is a stream cipher and therefore processes the files bit-wise, not block-wise as in block ciphers. Stream ciphers generate pseudorandom bits from the key and encrypts the plaintext by XORing it with the pseudorandom bits. The message size limit for ChaCha20-Poly1305 – the amount of data that can be securely encrypted using the same key- and nonce pair - is ~256 GiB.

Advantages

Crypt4GH	• Ready-to-use file encryption format developed by Global Alliance for Genomics and Health • Standard for genomics- and health related data • Provides *confidentiality* – The files can only be decrypted by holders of the correct secret key • Guarantees *integrity* – Each data block includes a message authentication code, MAC. A change in the data results in an invalid MAC The message size limit of the encryption algorithm (see ChaCha20-Poly1305) is not exceeded and there is a new key- nonce pair for each data block.
AES-256-GCM	• Most widely used authenticated cipher • Essentially an *encrypt-then-MAC* construction – strongest security • Protects against insertions, deletions, reordering – the MAC represent the entire file • Parallelizable – Possible to encrypt and decrypt different blocks independently of other blocks • Streamable – the MAC is computed from each block as it's encrypted. No need to store all blocks before computing, or process the file a second time. • Available in practically all crypto packages for all languagues. This would mean that client side encryption in the browser using JavaScript would be possible, and therefore a single algorithm could be used throughout the Data Delivery System. Part of the Internet Engineering Task Force (IETF) for secure network protocols IPSec, SSH, TLS
ChaCha20-Poly1305	• Is based on the ARX design – does not use substitution boxes as AES does and therefore does not produce cache footprints. It is therefore not vulnerable to cache-timing attacks. • Fast in software implementations. Multiple times faster than AES. • The security is currently considered to be at least as secure as AES-GCM • Intrinsically simpler than AES • Easier to implement and not as easy to get wrong • The message size limit is larger than for AES

	• Used in Crypt4GH file format – the encryption format standard for genomics and health related data • Part of the Internet Engineering Task Force (IETF) for secure network protocols IPSec, SSH, TLS

Disadvantages

Crypt4GH	• Does not protect against insertions, deletions or reordering. The MACs represent a block each, not the entire file. • Does not provide any way of authenticating the files – the sending and receiving parties identities cannot be proven. The reader cannot know who has encrypted and sent the file, and the writer cannot know who decrypts and reads the file. • Although the possibility of file access by multiple users and random access is useful in many cases, for delivery purposes these features are not important and do not help. • For encryption to begin, all previous operations such as compression, need to be finished. This means that the files need to be processed from start to end multiple times, leading to a large increase in delivery time. This will be specially evident for large or even huge files, which will be common in the delivery system. It may be possible to alter the crypt4gh package code to enable streaming (tested without success), however since the file format does not provide simplifying properties for our purposes, this was decided against.
AES-256-GCM	- Files larger than 64 GiB need to be partitioned. Alternatively multiple new key-nonce pairs would need to be saved and kept track of in another way. - Nonce repetition reduces the security drastically – tags can be forged - Not as fast in software implementations - Easy to get wrong - Vulnerable to cache-timing attacks
ChaCha20-Poly1305	• Not the general standard • Has not undergone as much cryptanalysis as AES and may therefore be vulnerable to attacks which are currently unknown • To our knowledge, it is not available within the WebCrypto API, used for client side encryption in the browser. Therefore, AES may still need to be implemented within the web interface.

Conclusion

For the reasons listed above Crypt4GH was NOT chosen as the encryption format within the Data Delivery System. Also, we found that the advantages of ChaCha20 overweighed the advantages of using AES-256-GCM and therefore AES-256-GCM was NOT chosen as the encryption algorithm within the Data Delivery System.

Since files delivered within the system can be huge, it is important to choose a format or algorithm which has a high message size limit, is as fast and secure as possible for all file sizes, and has the least possible risk of implementation error. ChaCha20 fulfills this, and is in addition supported by both the Global Alliance for Genomics and Health, and IETF secure network protocols. Although ChaCha20 is not the general encryption

standard, it is becoming increasingly more used, becoming favored over AES in certain aspects and so far cryptanalysis have not (to our knowledge) found any vulnerabilities that are deal-breakers. The issue regarding client-side encryption in the browser for the web interface may be solved in one of the following ways:

- HTTPS will be used, which is secured via TLS. If TLS is deemed to give enough protection to the uploaded data from the browser to the server, ChaCha20 encrypts the files when they have reached the server and uploads them to Safespring S3 in a similar way as the CLI.
- If an extra layer of security is needed, encryption can be performed in the browser using AES-GCM. In this case, files smaller than the web interface size cap will also be encrypted using AES-GCM in the CLI. Information on the algorithm used per specific file will be added to the database. Since only smaller files will be able to be uploaded and downloaded through the web interface, the files encrypted with ChaCha20 will not be a problem since they would have needed to be downloaded through the CLI regardless of used encryption algorithm.

Files larger than 256 GiB will not need to be partitioned[63], as oppose to if AES-GCM were to be used. Finally, the algorithm description does not include how the data encryption key will be distributed in a secure manner – this will be handled in a similar way to the Crypt4GH format, but the public keys will be signed and uploaded to the database and not saved within the files.

Due to this, **ChaCha20-Poly1305[64]** was chosen as the encryption algorithm within the Data Delivery System.

[63] The maximum size of 256 GB is per key-nonce pair. See *B.8. File Integrity Guarantee: Nonce Incrementation* for a description of how this works and why there is no practical size limit while encrypting with ChaCha20-Poly1305.
[64] ChaCha20-Poly1305-IETF to be more specific.

B.7. Size of file chunks: 64 KiB[62]

Compression and encryption is performed in a streamed manner to avoid reading large files completely in memory. This is done by reading the file in chunks, and the size of the chunks affect the speed of the algorithms, their memory usage, and the final size of the compressed and encrypted files.

Comparisons

To find the optimal chunk size, a 33 GB file was compressed and encrypted using the chosen algorithms (Zstandard and ChaCha20-Poly1305) after reading the file in different sized chunks ranging from 4 KiB to 500 KiB.

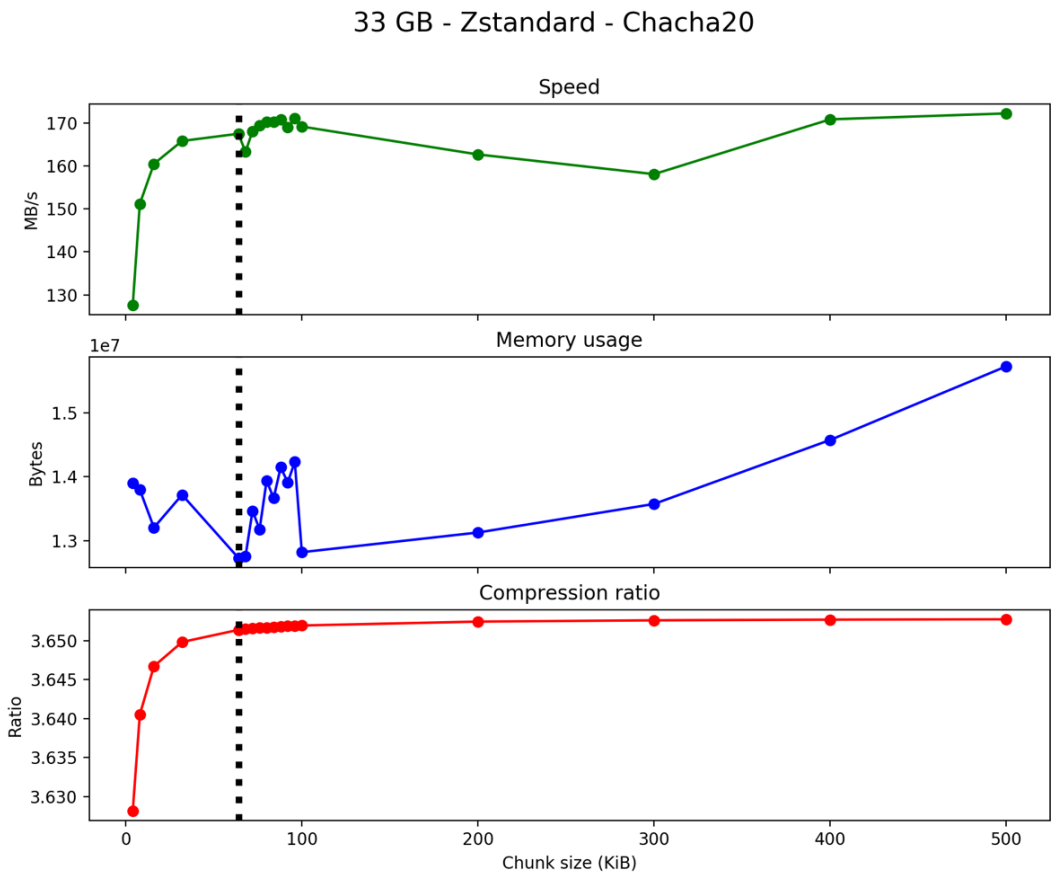


Figure 8. Measurements performed on a 33 GB file, while reading chunks of different sizes from the file. The x-axis shows the chunk sizes, ranging from 1 to 500 KiB. The dotted line shows the determined optimal chunk size of 64 KiB.

Conclusion

The image above shows why **64 KiB** was chosen - the memory required for the operations increases after 64 KiB, without any significant gain in speed or compression ratio[65].

[65] *Compression ratio* is defined as the uncompressed size divided by the compressed size, however, in this case the overhead of the encryption algorithm is also included. Thus here the compression ratio is calculated as the uncompressed size divided by the final size (compressed and encrypted).

B.8. File Integrity Guarantee: Nonce Incrementation

As described *in B.6. Encryption Algorithm: ChaCha20*, the Crypt4GH format encrypts the files in blocks of 64 KiB[62], after which each data blocks unique nonce, ciphertext and MAC are saved to the c4gh file. This guarantees the integrity of the data blocks, however it does not guarantee the integrity of the entire file, and it is therefore possible that some blocks are rearranged, duplicated or missing, without the recipient knowing. Although we have chosen to not use the Crypt4GH format within the delivery system, we do use the same encryption algorithm – ChaCha20-Poly1305 – and (since we cannot read huge files in memory) we have chosen to read the files in equally sized chunks. Therefore the integrity issue can potentially give problems for the delivery system.

The alternatives

- Generate a checksum for entire file
- Nonce incrementation

Motivation

File integrity check after upload and download

Initially, a file md5 checksum was generated during file processing. After upload, the checksum was compared to the ETag for the S3 object, indicating if the complete files had been uploaded. However, this method did not work for multipart uploads - files larger than 5 MB. On investigation of this, an GitHub issue was found[66], which addresses the issue:

> ”[…] the md5 is only equal to the ETag under certain circumstances (i.e. non multipart upload).
>
> Boto3 does not integrity check the md5 of the entire file for multipart uploads, but it does send an md5 header for each part that is uploaded when doing a multipart upload such that if there is an md5 mismatch in any of the parts, it will retry the request until it is correct.
>
> That is probably the best we can do while still doing the transfer efficiently because determining the MD5 and doing integrity checking would require streaming the entire file upfront into memory.”

Due to this, **no checksum** verification is used during the upload. However, the file integrity is checked by the Data Delivery System in other ways, as described below.

File integrity guarantee via compression/decompression

The Python Zstandard package used in the Data Delivery System allows the user to specify `write_checksum = True` which results in a checksum of the compressed data being saved in the compressed file. We have not yet determined the exact process of this (e.g. if it updates the checksum for each chunk resulting in one for the entire file or if it saves each chunks checksum), however this is under investigation. None the less, this (at the

[66] **https://github.com/boto/boto3/issues/845#issuecomment-253924586**

least) provides an integrity assurance of the 64 KiB chunks, since decompression will fail if the checksums do not match. This protects the data recipient from any corrupted (and possibly harmful) files.

However, files which are already in a compressed format will not be compressed by the Data Delivery System (see *C.5. Compression check*). At this time, there is no corresponding checksum generation or check for these files. This is being discussed and may be implemented in future versions of the system.

File integrity guarantee via encryption/decryption

One solution to the integrity guarantee issue is to generate an additional checksum, representing the entire file, but for security reasons it is recommended to let the checksum generation be handled by (and in connection to) the encryption algorithm. Another solution[67] is to generate a random nonce, and increment it for each block. In this case the decryption only needs to know the first blocks nonce, and then increments it in the same way as the encryption until the end of the file. If the decryption of a block fails - the file has been altered.

Since the nonces are regarded as public knowledge, the incrementation of the nonces do not decrease security - it may on the other hand *increase* the security: unique nonces for each data block is vital to the cryptographic security and generating a random nonce for each data block increases the risk of nonce reuse. Incrementing the nonces however mean that 2^{96} (for 96 bit nonces as used in ChaCha20-Poly1305) data blocks can be encrypted before nonce reuse – $2^{96} \times 64 \times 1024 \times 8 \approx 5.2 \times 10^{21}\ terrabytes\ (TB)$. This number may even be higher, since ChaCha20 itself adds and increments 4 bytes to the user-specified nonce. None the less, it's safe to say that files of this size will not exist. An extra advantage to the nonce incrementation option is that only the first nonce needs to be saved, reducing the encryption overhead by 128 bits per data block.

One problem with this solution is that, although we guarantee the order of the data blocks, we don't know if all the data blocks are present – blocks at the end of the file may be missing without us knowing. Due to this, we also save the last nonce to the file. If decryption reaches the last nonce, the files integrity is proven, if not – something has gone wrong.

Finally, as stated in the introduction of this section (B.8.), the ChaCha20-Poly1305 (more specifically the Poly1305 authenticator) algorithm produces a message authentication code (MAC) during encryption of each chunk. This MAC is automatically[68] saved to the file together with the ciphertext[69]. When decryption is performed, each MAC is automatically authenticated by the Poly1305 prior to the decryption process. Thus, if a MAC validation fails (the file chunk is corrupted or the key is not correct), decryption is not performed. This process thereby provides confidentiality of the data, confirms its integrity and provides a guarantee for the data recipient that it is not exposed to any corrupted data.

Conclusion

The file integrity checks performed within the Data Delivery System are:

- Automatically using **Boto3** as described above
- A **file checksum generated by Zstandard** and stored in the compressed file
- By using **nonce incrementation** during the encryption process
- The **MACs** generated during encryption and verified during decryption

[67] **https://libsodium.gitbook.io/doc/secret-key_cryptography/encrypted-messages**
[68] Not with all implementations of the algorithm. However, this is true for the Python PyNaCl package used in the Data Delivery System.
[69] Ciphertext: The encrypted plaintext

Appendix C. The components

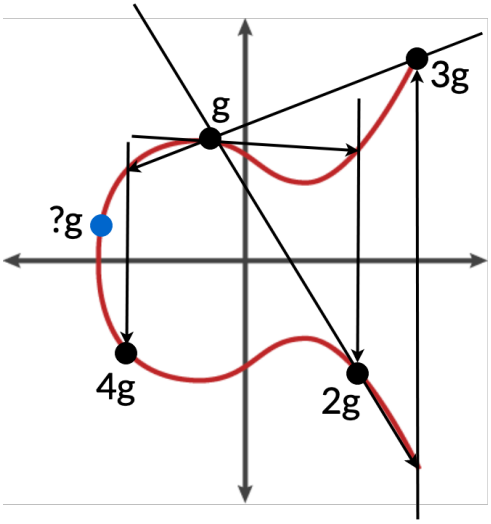
C.1. Elliptic-Curve Diffie-Hellman (ECDH) Key Exchange

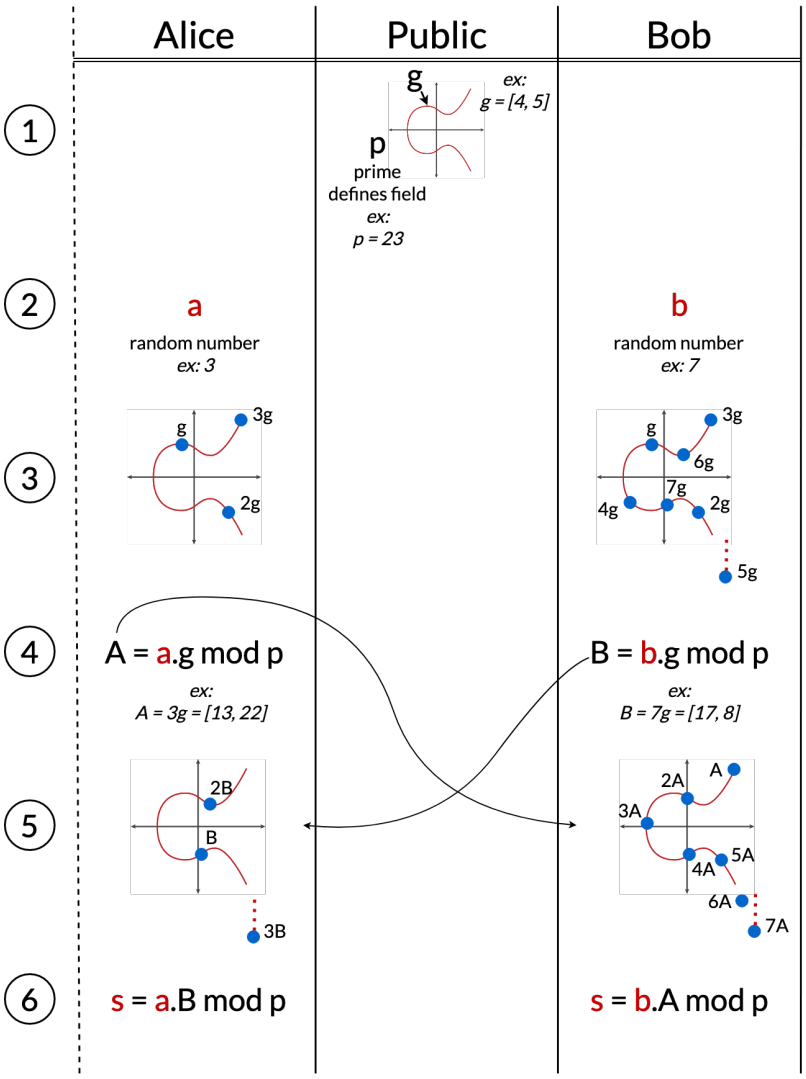
Elliptic-Curve Diffie-Hellman (ECDH) is a newer version of Diffie-Hellman (DH) and they are both used purely for exchanging keys (not encrypting or decrypting messages) or rather, enabling communicating parties to generate identical keys without sharing any secret parameters. ECDH replaces the exponential part in DH with a point multiplication - we add the starting point (the generator **g**) on the elliptic curve to itself **a** times. However, arithmetic operations with elliptic curves[70] does not have the conventional meanings:

OPERATION	FORMULA	EXPLANATION
POINT ADDITION	P + Q	A straight line is drawn through point **P** and point **Q**. Find the point **-R** where the line intersects the curve a 3rd time, and find the inverse (on the other side of the x-axis) of -R (**R**). R = P + Q
POINT DOUBLING **(MULTIPLICATION)**	4P	The tangent of point **P** is drawn. Find the point where the line intersects the curve a 2nd time, and find the inverse of that point. This is point **2P**. Draw a line through the points **P** and **2P**, find the point where the line intersects the curve a 3rd time and find the inverse of that point. This point is **3P**. Draw a line through the points **P** and **3P**, find the pointwhere the line intersects the curve a 3rd time and find the inverse of that point. This point is **4P**. Below is an example of this procedure, except **P** is called **g**.

[70] An elliptic curve has the form: $y^2 = x^3 + ax + b$. a and b are here only parameters defining the curve, **not the later used private random numbers.**

$$y^2 = x^3 + ax + b$$


g
3g
?g
4g
2g


Alice
Public
Bob
1
g
ex:
g = [4, 5]
p
prime
defines field
ex:
p = 23
2
a
random number
ex: 3
b
random number
ex: 7
3
g
3g
2g
g
3g
6g
7g
4g
2g
5g
4
A = a.g mod p
ex:
A = 3g = [13, 22]
B = b.g mod p
ex:
B = 7g = [17, 8]
5
2B
B
3B
2A
A
3A
4A
5A
6A
7A
6
s = a.B mod p
s = b.A mod p

Below is the procedure step by step.

1. Alice and Bob want to communicate. They first agree on a specific elliptic curve, such as Curve25519 used in crypt4gh, which is defined by a number of different parameters such as **p** - a large prime number which defines the field ("valid" points). They also agree upon a specific point **g** - the generator, just as in DH. The value of **p** is the x-coordinate. Calculate the y-coordinate with the curve function.
2. Alice and Bob generate their own random numbers **a** and **b**, respectively, which may be between 0 and **p**.
3. Alice performs point doubling[71] (see above) of **g** **a** number of times. Bob performs the same operation **b** number of times.
4. Both of them now calculate their public points **A** and **B**, respectively, by taking the final point found in step 3 (**ag** and **bg**) modulo **p**. Alice and Bob now share their public points with each other.
5. Point doubling is now done once more, Alice on point **B, a** number of times, and Bob on point **A**, **b** number of times.
6. Finally, they both calculate their shared key **s** by taking the final point (found in step 5) mod p, as done in step 4. They now share a symmetric key and can now begin communicating securely.

This results in…

- The **same level of security** with **smaller key sizes**
- A smaller key size results in the **computations** being **less extensive**…
- … and therefore also **faster**

[71] During point doubling, some points may end up outside of the field defined by p. Therefore, mod p is used, resulting in all points ending up in a single defined space, such as seen in the scatter-plot below. This is similar to how DH works except here we are working with a curve/scatter-plot instead of a circle.

C.2. User account creation

User accounts are as mentioned created manually. How this will be handled in later versions of the Data Delivery System is still under discussion.

When a user is created, the following information is stored in the database:

USER TYPE	INFORMATION (COLUMN IN TABLE)	DESCRIPTION
FACILITY (DATA PRODUCER)	***ID***	User ID
	Name	Name of facility
	Internal_ref	Internal reference ID
	Username	The Data Delivery System username
	Password	The derived hash of the password provided by the user.[72]
	Settings	The parameters[73] used for password hash derivation.
	Email	The users email address
	Phone	Phone number
OTHER USER (DATA RECIPIENT)	***ID***	User ID
	First name	The users first name
	Last name	The users last name
	Username	The Data Delivery System username
	Password	The derived hash of the password provided by the user[72].
	Settings	The parameters[73] used for password hash derivation.
	Email	The users email address
	Phone	Phone number
	Admin	If the user is an admin or not

Note that this setup does not include any invoicing information and that the database structure will change before the stable version of the system is released. The database also does not track the usage per user at this time.

[72] See *Appendix B. Data Delivery System Decisions, B.3. Password verification: Scrypt*
[73] These are regarded as public information and storing them in this manner does not reduce the security level of the system. See *footnote 70*.

C.3. Project creation

As with the user accounts, project information is currently manually added to the database. This will also be changed in later versions of the system.

On project creation[74], the following information is specified:

	INFORMATION (COLUMN IN TABLE)	DESCRIPTION	WHERE DO I FIND MORE INFORMATION?
CHOSEN BY USER	***ID***	Project ID	Required option for CLI. See *section 3.3. Command Line Interface, CLI*Command Line Interface, CLI
	Title	Project title	Information displayed by web interface. See *section 0. Web* Interface.
	Category	Type of project	
	Sensitive	If the project is of sensitive nature (and there by protected by privacy regulations etc) or not.	
	Description	Short description of the project.	
	PI	Principal investigator. Primary individual responsible for the project.	
	Owner	Project owner. Data recipient.	Required option for CLI. See *section 3.3. Command Line Interface, CLI.*
	Facility	The ID of the facility involved in the project. The data producer.	
	Delivery option	Chosen data delivery medium. Currently only S3.	Checked before upload and download. See *Appendix A. How does the CLI work?*

[74] With ”project creation” we do not mean the initial start of a specific project, but the (manual) creation of the project information and saving it to the database.

GENERATED BY SYSTEM (AUTOMATICALLY)	***Order date***	Timestamp for when order was created/added to the database[75].	Code for time stamp generation seen on GitHub[76].
	Delivery date	Initially zero. Updated to be a timestamp when an upload is performed[77].	
	Status	Whether the data delivery has begun or not, or if is finished.	See section *0. Web* Interface.
	Size	Sum of the original size of all files uploaded within the project[78].	
	Encrypted size	Sum of the size of all files, after processing, uploaded within the project.	
	Public key	Project-specific public key. Part of the ECDH key pair.	See the next page.
	Private key[79]	Project-specific private key. Part of the ECDH key pair.	
	Salt	The string of random bytes used for generating the key which later is used to encrypt the private key.	See the next page.
	Nonce	The string of random bytes used for encrypting the private key.	See the next page.
	Passphrase[80]	The passphrase which (together with the salt mentioned above) creates the key later used to encrypt the private key.	See the next page.

[75] The formatting of the timestamp is under development.
[76] **https://github.com/ScilifelabDataCentre/Data-Delivery-System/blob/f52dfd62df6749aff565ea708400770382a2d019/code_dds/__init__.py#L65**
[77] In future versions of the system, there will likely be multiple timestamps relating to one project, enabling the system to keep track of both the final upload and the previous ones.
[78] This information is currently not used in any meaningful way. It may included in some kind of integrity check in future versions.
[79] This key will not be stored together with the other project information in later versions. This is a initial version. The proper key management is under construction.
[80] The passphrase will not be stored together with the other project information in later versions. This is a initial version. The proper key management is under construction.

Creating the project information which will later be automatically generated by the system is currently handled by a Python script. The script works in the following way[81]:

1. The following variables are defined:
   - Data Delivery System signature (`DS_MAGIC`)
   - Project ID (`PROJECT_ID`, `P_ID_BYTES`)
   - Passphrase for private key encryption (`PRIVKEY_ENC_PASSPHRASE`)

   The project ID is first saved in a JSON file. The script gets the string from the file and transforms it to a byte format.

```
-   """Manually create an account for the Data Delivery System."""
-
-   ##############################################################################
-   # IMPORTS ################################################### IMPORTS #
-   ##############################################################################
-
-   # Standard library
-   import logging
-   import os
-   import json
-
-   # Installed
-   from cryptography.hazmat import backends
-   from cryptography.hazmat.primitives import serialization
-   from cryptography.hazmat.primitives.asymmetric import x25519
-   from cryptography.hazmat.primitives.kdf import scrypt
-   from nacl import bindings
-
-   ##############################################################################
-   # LOGGING ################################################### LOGGING #
-   ##############################################################################
-
-   logging.basicConfig(filename='create_account.log',
-                       level=logging.DEBUG)
-
-   ##############################################################################
-   # VARIABLES ################################################# VARIABLES #
-   ##############################################################################
-
-   # Delivery System signature
-   DS_MAGIC = b'DelSys'
-
-   # Get the project ID and passphrase from json file
-   with open("project_info.json") as f:
-       info = json.load(f)
-
-   # Project ID
-   PROJECT_ID = info["PROJECT_ID"]             # Project ID
-   P_ID_BYTES = bytes(PROJECT_ID, "utf-8")     # Project ID in bytes
-
```

[81] This code is not included in the GitHub repository.

```
-   # Passphrase to encrypt private key
-   PRIVKEY_ENC_PASSPHRASE = os.urandom(16)
```

2. Generate the keys
   - **2.1.** The *private key* is generated.
   - **2.2.** The private key is transformed to bytes.
   - **2.3.** The corresponding *public key* is derived from the private key.
   - **2.4.** A 16-byte *salt* is generated.
   - **2.5.** Scrypt[82] is used to derive a key from the passphrase (step 1) and the salt (step 2.4.). This key is from now on called *derived key*.

```
-   ################################################################################
-   # KEYS - GENERATE ############################################ KEYS - GENERATE #
-   ################################################################################
-
-   # Generate keys
-   private = x25519.X25519PrivateKey.generate()
-   # logging.debug("private key: %s", private)
-
-   # Convert private key to bytes
-   private_bytes = private.private_bytes(
-       encoding=serialization.Encoding.Raw,
-       format=serialization.PrivateFormat.Raw,
-       encryption_algorithm=serialization.NoEncryption()
-   )
-   # logging.debug("private bytes: %s", private_bytes)
-
-   # Get public key
-   public = private.public_key()
-   # logging.debug("public key: %s", public)
-
-   # Convert public key to bytes
-   public_key = public.public_bytes(encoding=serialization.Encoding.Raw,
-                                    format=serialization.PublicFormat.Raw)
-
-   # Generate salt
-   salt = os.urandom(16)
-   # logging.debug("salt: %s", salt)
-
-   # Derive private-key-encryption-key
-   kdf = scrypt.Scrypt(salt=salt, length=32, n=2**14, r=8, p=1,
-                       backend=backends.default_backend())
-   derived_key = kdf.derive(PRIVKEY_ENC_PASSPHRASE)
-   # logging.debug("derived key-encryption-key: %s", derived_key)
```

The logging is included for detecting bugs during development.

[82] See section *Appendix B. Data Delivery System Decisions, B.3. Password verification: Scrypt*

3. The private key (step 2) is prepended with extra information:
   - The length of the Data Delivery System signature and the signature itself.
   - The length of the project ID and the project ID itself (step 1).
   - The length of the private key and the private key itself.

```
-   # Format of key:
-   # -------------------------Format in database:--------------------------------
-   #  len(magic) + magic + len(projID) + projID + len(privateKey) + privateKey
-   # ----------------------------------------------------------------------------
-   priv_key_formatted = (
-       len(DS_MAGIC).to_bytes(2, byteorder='big') + DS_MAGIC +
-       len(P_ID_BYTES).to_bytes(2, byteorder='big') + P_ID_BYTES +
-       len(private_bytes).to_bytes(2, byteorder='big') + private_bytes
-   )
-   # logging.debug("key for db: %s", priv_key_formatted)
```

This format will from now on be called the *formatted key*.

4. The formatted key is verified to check that the previously defined format is correct and nothing has gone wrong in the generation.
   4.1. The first two bytes are read. This gives information on how long the Data Delivery Signature is. This length is now referred to as ***X***.
   4.2. The next ***X*** bytes are read. These are checked to confirm that these bytes match the Data Delivery Signature.
   4.3. The next two bytes are read. This gives information on how long the project ID is. This length is now referred to as ***Y***.
   4.4. The next ***Y*** bytes are read. These are checked to confirm that these bytes match the project ID from the json file (step 1).
   4.5. The next two bytes are read. This gives information on how long the private key is. This length is now referred to as ***Z***.
   4.6. The next ***Z*** bytes are read. These are checked to confirm that these bytes match the private key.
   4.7. The script controls that there are no bytes left in the formatted key to read. If there is, the key generation failed. If there is not, it continues.

```
-   ##############################################################################
-   # KEYS - VERIFY ############################################## KEYS - VERIFY #
-   ##############################################################################
-
-   # Read 2 bytes and get the length of the delivery system signature
-   START = 0       # Start reading at position 0
-   TO_READ = 2     # Number of bytes to read
-   magic_id_len = int.from_bytes(priv_key_formatted[START:START+TO_READ], 'big')
-   # logging.debug("magic_id_len: %s", magic_id_len)
-   assert magic_id_len == len(DS_MAGIC), ("The DDS signature lengths do not "
-                                          f"match! Should be {len(DS_MAGIC)} "
-                                          f"but found {magic_id_len}.")
-
-   # Read magic_id_len bytes -> magic id - should be b'DelSys'
-   START += TO_READ        # Increase the start - have read bytes prev
-   TO_READ = magic_id_len
-   magic_id = priv_key_formatted[START:START+TO_READ]
```

```
-   # logging.debug("magic id: %s", magic_id)
-   assert magic_id == DS_MAGIC, ("Error in private key! Signature should be "
-                                 f"{DS_MAGIC} but found {magic_id}.")
-
-   # Read 2 bytes and get length of project id
-   START += TO_READ
-   TO_READ = 2
-   proj_len = int.from_bytes(priv_key_formatted[START:START+TO_READ], 'big')
-   # logging.debug("project id len: %s", proj_len)
-   assert proj_len == len(P_ID_BYTES), ("The project id lengths do not match! "
-                                        f"Should be {len(P_ID_BYTES)} "
-                                        f"but found {proj_len}.")
-
-   # Read proj_len bytes -> project id - should be equal to current proj
-   START += TO_READ
-   TO_READ = proj_len
-   project_id = priv_key_formatted[START:START+TO_READ]
-   # logging.debug("project id: %s", project_id)
-   assert project_id == P_ID_BYTES, ("Error in private key! Project ID incorrect!"
-                                     f"Should be {P_ID_BYTES} "
-                                     f"but found {project_id}")
-
-   # Read 2 bytes and get the length of the private key
-   START += TO_READ
-   TO_READ = 2
-   key_len = int.from_bytes(priv_key_formatted[START:START+TO_READ], 'big')
-   # logging.debug("key length: %s", key_len)
-   assert key_len == len(private_bytes), ("The private key lengths do not match! "
-                                          f"Should be {len(private_bytes)} "
-                                          f"but found {key_len}.")
-
-   # Read key_len bytes -> key
-   START += TO_READ
-   TO_READ = key_len
-   key = priv_key_formatted[START:START+TO_READ]
-   # logging.debug("key: %s", key)
-   assert key == private_bytes, ("Error in private key bytes!"
-                                 f"Should be {private_bytes} but found {key}")
-
-   # Error if there are bytes left after read key
-   assert priv_key_formatted[START+TO_READ::] == b'', ("Error in private key! "
-                                                       "Extra bytes after key -- "
-                                                       "parsing failed or key "
-                                                       "corrupted!")
-   # logging.debug("Private key to encrypt: %s", priv_key_formatted)
```

5. The formatted key is encrypted.
   5.1. A 12-byte nonce[83] is generated.
   5.2. The formatted key is encrypted with the derived key (step 2). This key is now referred to as the *encrypted key*.

```
-   ###############################################################################
-   # KEYS - ENCRYPT ############################################## KEYS - ENCRYPT #
-   ###############################################################################
-
-   # Generate a nonce for encryption
-   nonce = os.urandom(12)
-   # logging.debug("nonce: %s", nonce)
-
-   # Encrypt private key (in full format)
-   encrypted_key = bindings.crypto_aead_chacha20poly1305_ietf_encrypt(
-       message=priv_key_formatted, aad=None, nonce=nonce, key=derived_key
-   )
-   # logging.debug("encrypted key: %s", encrypted_key)
```

6. The encrypted key is transformed to a hex-string.
7. The formatted key ("PRIVATE KEY"), public key, salt and nonce are printed out[84].

```
-   ###############################################################################
-   # PRINT AND SAVE ############################################## PRINT AND SAVE #
-   ###############################################################################
-
-   # Transform keys to hex string
-   hex_private_encrypted = encrypted_key.hex().upper()
-
-   print(
-       "\n\n*****************************************************************\n\n"
-       f"PRIVATE KEY: \t{hex_private_encrypted}\n"
-       f"PUBLIC KEY: \t{public_key.hex().upper()}\n"
-       f"SALT: \t\t{salt.hex().upper()}\n"
-       f"NONCE: \t\t{nonce.hex().upper()}"
-       "\n\n*****************************************************************\n\n"
-   )
-
```

[83] Nonce: "Number used only once". A string of random bytes.
[84] The printout of this information will be replaced by a call to the database.

C.4. The keys

As seen in this document, there are several different keys, passwords and related objects used within the Data Delivery System. This section aims to clarify which all the keys are and when and how they are used. To avoid confusion, the tables below show the terminology used in this section and a summary of their different usages. The information stored in the database is highlighted in light green. Information which is stored in the database, but which storage solution will be altered for security purposes, also has a dark green border.

Table 1

Time of creation:	**When account is created (user is registered)**	
Key/Object	**From now on called**	**Used for**
Password	*Password*	Getting access to the system
Salt[82]	*PwSalt*	Deriving the password hash, *PwHash*
n	*n*	
p	*p*	
r	*r*	
Password hash	*PwHash*	Protecting the password

Table 2

Time of creation:	**When project is created (project added to system database)**	
Key/Object	**From now on called**	**Used for**
Passphrase	*Passphrase*[85]	Deriving DKey
Salt[86]	*PfSalt*	Deriving DKey
Derived key	*DKey*	Encrypting PPrivF/Decrypting PPrivEF
Nonce[86]	*PrivNonce*	Encrypting PPrivF/Decrypting PPrivEF
Project-specific Private Key	*PPriv*	Deriving the shared key (decryption), see *Table 3*
Formatted project-specific Private Key	*PPrivF*	Providing extra layer of security
Formatted and encrypted project-specific Private Key	*PPrivEF*	Protecting the private key
Project-specific Public Key	*PPub*	Deriving the shared key (encryption/decryption), see *Table 3*
Access key	*Access key*	Access to the Safespring Storage project
Secret key	*Secret key*	Access to the Safespring Storage project

[85] Note that the passphrase will not be stored in the clear, and will be protected by and extra layer of security. Also note that this setup is not final.
[86] See *C.3. Project creation*

Table 3

Time of creation: When a file is being delivered (uploaded)		
Key/Object	**From now on called**	**Used for**
File-specific Private Key	*FPriv*	Deriving the shared key (encryption)
File-specific Public Key	*FPub*	Deriving the shared key (encryption/decryption)
Shared key	*Shared key*	Deriving the encryption key
Salt[87]	*FSalt*	Deriving the encryption key
Derived key[88]	*Encryption/decryption key*	Encrypting/decrypting a file
Nonce[89]	*FNonce*	Encrypting/decrypting a file

The following descriptions show the step-by-step process of generating the shared keys before encryption and decryption. These steps does not include the calls to the API or information on what and when keys and objects are retrieved from the database. For information on those steps, see *Appendix A. How does the CLI work?* how it works.

ENCRYPTION	**DECRYPTION**
1. The key-pair *FPriv* + *FPub* is generated – now referred to as *FPair*. **2.** A *shared key* is generated by combining[90]… • *FPair* • *PPub* **3.** *FSalt* is generated. ***4.*** The *encryption key* is derived by using the… • *Shared key* • *FSalt* **5.** *FNonce* is generated **6.** The file is encrypted by using the… • *Encryption key* • *FNonce*	The key-pair *PPriv* & *PPub* is now referred to as *PPair.* **1.** The *shared key* is generated by combining[90]… • *PPair* • *FPub* **2.** The *decryption key* is generated by using the… • *Shared key* • *FSalt* **3.** The file is decrypted with… • *Decryption key* • *FNonce*

[87] See *Appendix A. How does the CLI work?, A.2. Upload*
[88] Note! This does not refer to the "derived key" in Table 2. This is a different key, as can be seen from the "from now on called" column.
[89] See *Appendix A. How does the CLI work?, A.2. Upload*
[90] See *C.1. Elliptic-Curve Diffie-Hellman (ECDH) Key Exchange*

C.5. Compression check

All files go through a “compression check” before further processing and upload. This is done to see whether the Data Delivery System should compress the files using Zstandard[37] or if it should move on and simply encrypt the file.

The compression check does check the file extension, however, this is not the important step. The check reads the beginning of each file and looks for the signature belonging to common compression formats. The signatures (and therefore formats) it currently looks for a seen in the table below. Note that this is an initial version and that compression formats will be changed, removed and added to the check.

FORMAT	SIGNATURE (BYTES)	TYPE
ZIP	`PK\x03\x04`	Archive file format
GZIP	`\x1F\x8B`	File compression
BZIP2	`BZh`	File compression
7-ZIP	`7z¼¯'`	File archive
ZSTANDARD	`(\xb5/\xfd`	File compression
HAP	`\x913HF`	Video compression
ARJ	`\`\xea`	Compressed file archive
JAR	`_\'\xa8\x89`	Java Archive
ZOO	`ZOO`	General purpose file compression
STUFFIT	`StuffIt`	File compression
RAR V4	`Rar!\x1a\x07\x00`	Archive file format
RAR V5	`Rar!\x1a\x07\x01\x00`	Archive file format
LHA	`-lh`	Archive compression
HQX	`(This fi`	Compressed archive file

If there are formats missing from the check, please let us know. The same applies to if you find an error in a signature (or type description) or if we have included a format which you find irrelevant.

The compression check is performed in the following way:

1. The CLI calculates the length of the longest signature mentioned above. This length is now referred to as ***X***.
2. The first ***X*** bytes of the file is read
3. The CLI checks if the bytes match any of the signatures
4. If there is a match, the file is considered to be compressed and the Data Delivery System does therefore not perform Zstandard compression on the file
5. Information on if the file is compressed by the Data Delivery System is saved to the database
6. On data collection (download), this information is retrieved
7. If the file was compressed by the Data Delivery System during delivery (upload), the file is decompressed using Zstandard. If the file contained a matching signature, no decompression is performed by the system. The data recipient is in this case responsible for decompression of the file.