

# ShinyDataSHIELD User's Guide

Escribà Montagut, Xavier; González, Juan R.

2022-04-25



# Contents

<b>1</b>	<b>Overview</b>	<b>5</b>
<b>2</b>	<b>Setup</b>	<b>7</b>
2.1	Install with Docker . . . . .	7
<b>3</b>	<b>Functionalities</b>	<b>9</b>
3.1	Data entry . . . . .	9
3.2	Table column types . . . . .	17
3.3	Descriptive statistics . . . . .	29
3.4	Statistical models . . . . .	44
3.5	Genomics . . . . .	55
3.6	Omics . . . . .	60
<b>4</b>	<b>Developers Guide</b>	<b>63</b>
4.1	File structure of ShinyDataSHIELD . . . . .	63



# Chapter 1

## Overview



ShinyDataSHIELD is a non-disclosive data analysis toolbox powered by DataSHIELD with the following features:

- Descriptive statistics: Summary, scatter plots, histograms and heatmaps of table variables.
- Statistic models: GLM and GLMer model fittings
- Omic analysis: GWAS, LIMMA, ... using different types of resources (VCF files, PLINK, RSE, eSets)

The features available on ShinyDataSHIELD are powered by different packages of the DataSHIELD project (dsBaseClient and dsOmicsClient), it uses them in a seamless way so the final user of ShinyDataSHIELD can perform all the included studies without writing a single line of code and get all the resulting figures and tables by the click of a button.



## Chapter 2

# Setup

### 2.1 Install with Docker

Docker can be installed on a Linux / Mac OS X machine without any complications as any other application, on Windows systems however it can be a little bit more troubling, there are many online resources to help. Please refer to the following links to install Docker on Windows Home, to setup the Linux Windows Subsystem and terminal and to execute Docker on Windows.

Once Docker is up and running, execute the following command on a bash terminal (make sure Docker is running, if not search for the **Docker Desktop** app and launch it) to download and launch ShinyDataSHIELD. Be aware that the Docker images weights ~ 1.5 GB, so if your internet connection is slow it may take a while.

```
docker run --rm -p 80:80 brgelab/shiny-data-shield
```

The container will be exposed on the local port 80 and it will render on that port the application itself, so to start using ShinyDataSHIELD open your web browser of choice and go to the site

```
localhost:80
```

At the beginning it may take some time for the application to render, this is because all the needed R libraries are being loaded, to be sure the container is actually working, take a look at the terminal where you inputted the Docker command, there you will see all the R verbose stating the libraries are being loaded.

Once the user has finished using ShinyDataSHIELD, the container needs to be stopped to avoid wasting CPU resources, to do so, input the following command on a bash terminal (the command needs to be inputted on a new bash window):

```
docker container ls
```

This will prompt all the running containers, find the one with the NAMES `brgelab/shiny-data-shield` and copy its CONTAINER ID, then input the following bash command:

```
docker stop xxxxxxxxxxxx
```

Where xxxxxxxxxxxx is the CONTAINER ID.

To run the application again, just enter the first bash command (`docker run --rm -p 80:80 brgelab/shiny-data-shield`), since it has already been downloaded, the application is cached on the computer and it will launch straight away. If the user wants to remove the Docker image from the computer, input the following bash command:

```
docker image rm brgelab/exposome-shiny
```

If the user wants to download the actual source code of the Shiny, install all the required packages and launch it locally on its machine, feel free to download it from Github. There's a script called `installer.R` at the root of the repository with a short installer of all the required packages. Please note that the installer script may fail depending on the R version and others, for that reason is advised to always run the Docker version of ShinyDataSHIELD, as it only requires a single terminal command and will work no matter what.



## Chapter 3

# Functionalities

Along this section, an overview of the functionalities implemented on Shiny-DataSHIELD is given. There's information about how to use the functionalities as well as some limitations or constraints to take into account when using Shiny-DataSHIELD.

### 3.1 Data entry

---

#### DISCLAIMER

---

Along this section the terms **table** and **resource** are widely used, it is important noting that when the autor talks about a table, it refers to what is shown as a table on the Opal server. A resource that holds a table is called (and treated) as a resource.

---

The first step to any analysis is to load the required tables or resources to the study server(s). To do so, the user has to provide the server URL and the login credentials. This will allow the application to connect to the OPAL server and retrieve all the projects and resources / tables from them. Afterwards, the user can select the desired resources / tables and load them to the study servers.

It is very important to understand the difference between a server (OPAL server) and a study server (R instances running inside an OPAL server), a study server can be visualized as an R session that has some tables and resources loaded. For that reason we can have multiple study servers on a single OPAL server, which is indeed needed when performing pooled studies.

To properly explain how to load the desired study servers for multiple types of analysis, let's do a guided tour of the interface.

Since Opal 4.2 there is a new concept introduced called **profiles** which correspond to different R servers (rockr). The user can choose which profile to use for each study server (more info here).

On Figure 3.1 the initial state of the *Connect to server* tab is shown, here we can see some credentials are already inputted, which correspond to a demo server. There is the option of performing a login to the server via traditional user/pass or using personal access token [PAT].

Once connected to the server, the interface is actualized showing the available projects that contain tables inside. To show the projects that contain resources, just click on the *Table* toggle. An example is illustrated on Figure 3.2.

If we want to get further information from a table(s) or resource(s), select them and click on the *Further information of selection*, this will open the OPAL UI on a browser and will display the associated page for the selected items. If more than one item is selected, a page for each one will be opened. An example of this functionality is shown on the Figure 3.3 where a table was selected to retrieve further information.

Knowing which items are of our interest, we can add them to be loaded on the study servers, to do that select the items and click *Add selected item(s)*. A new table will appear underneath showing everything selected, illustrated on Figure 3.4. On this table all the elements from all the servers will be added to have the general picture of what will be available on the study servers.

Reached this point, let's see different use cases and how we have to configure the application for them.

### 3.1.1 Single resource / table approach

Not much to say about this, just select it and click the *Connect* button.

### 3.1.2 Multiple resource study

Some studies require to put more than one resource on a single study server, this is the case of using VCF files to perform a GWAS; they require two resources, the VCF resource and the covariates resource (which is a resource that holds a plain table). For this use case, the user has to select the multiple resources from the dropdown inputs, add them to a single study and connect to it. Two important things must be said for this use case, 1) By default the tables/resources from one server are added to the same study server; 2) to have tables/resources on the same study server, they have to be from the same server, this translates to: we can't put a table from server X and a resource from server Y on the same study server.

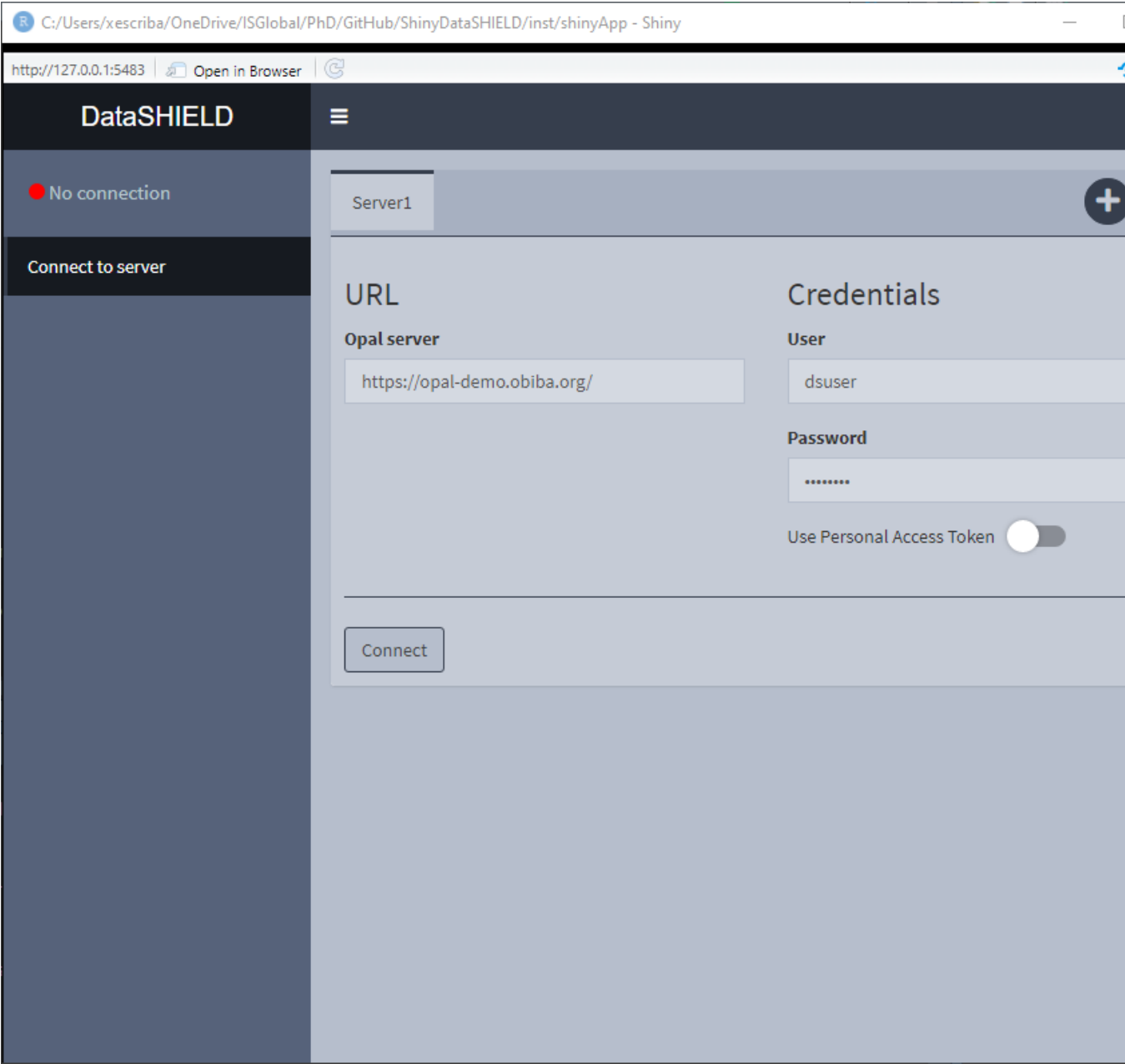


Figure 3.1: Connect to server, initial state

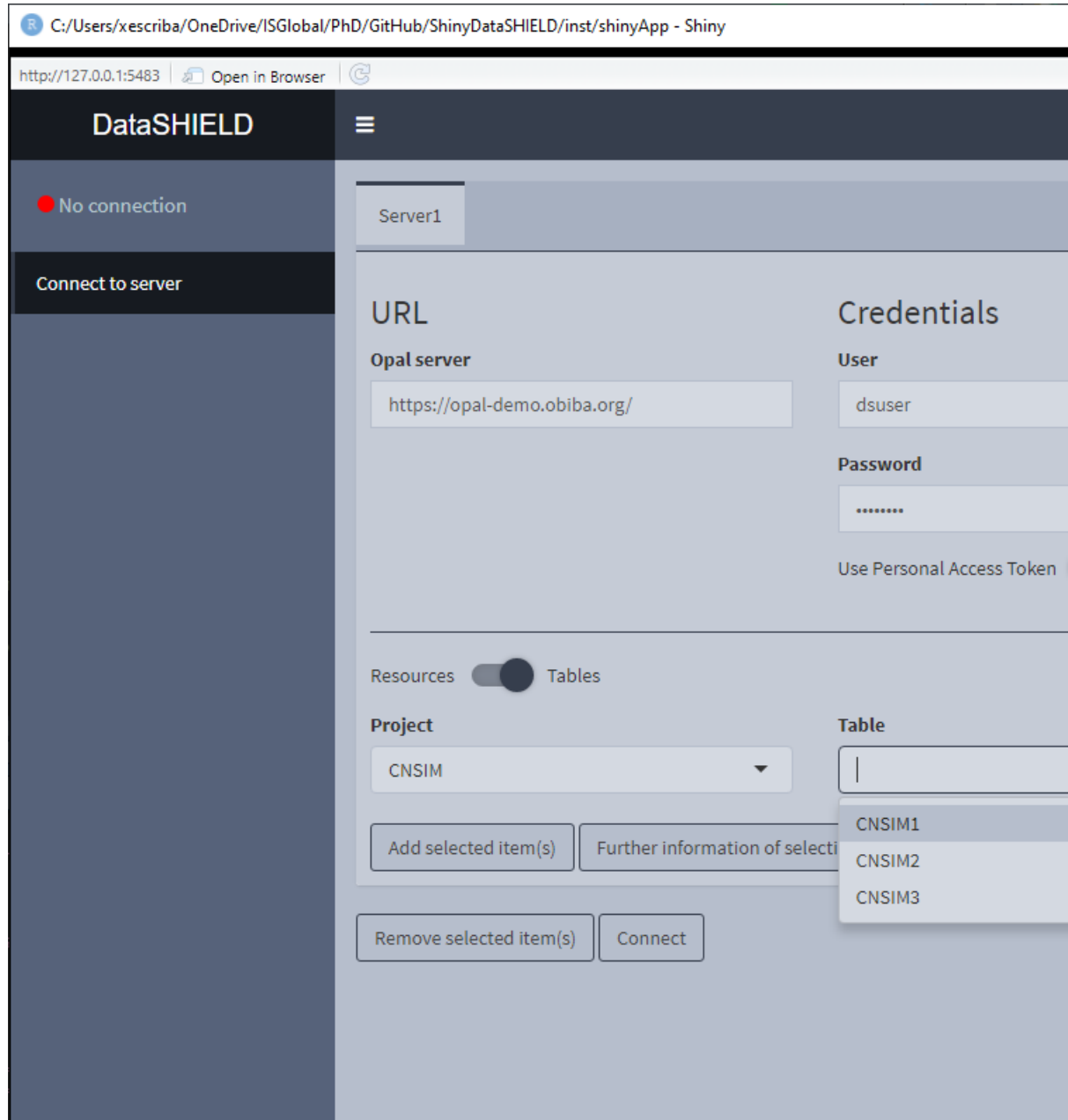



Figure 3.2: Connect to server, selecting tables and resources

Opal


Dashboard

Projects


Search

0


S




Tables




T CNSIM / CNSIM1 ☆




Dictionary




Summary




Values




Analyses



Permissions



Properties 

Name	CNSIM1
Entity Type	Participant

Variables

+ Add Variable

Select variables to add to view or cart, manage attributes or remove.

<input type="checkbox"/>	Name	Label
<input type="checkbox"/>	LAB_TSC	<div>en</div> Total Serum Cholesterol
<input type="checkbox"/>	LAB_TRIG	<div>en</div> Triglycerides
<input type="checkbox"/>	LAB_HDL	<div>en</div> HDL Cholesterol
<input type="checkbox"/>	LAB_GLUC_ADJUSTED	<div>en</div> Non-Fasting Glucose

Figure 3.3: OPAL UI with information from a table

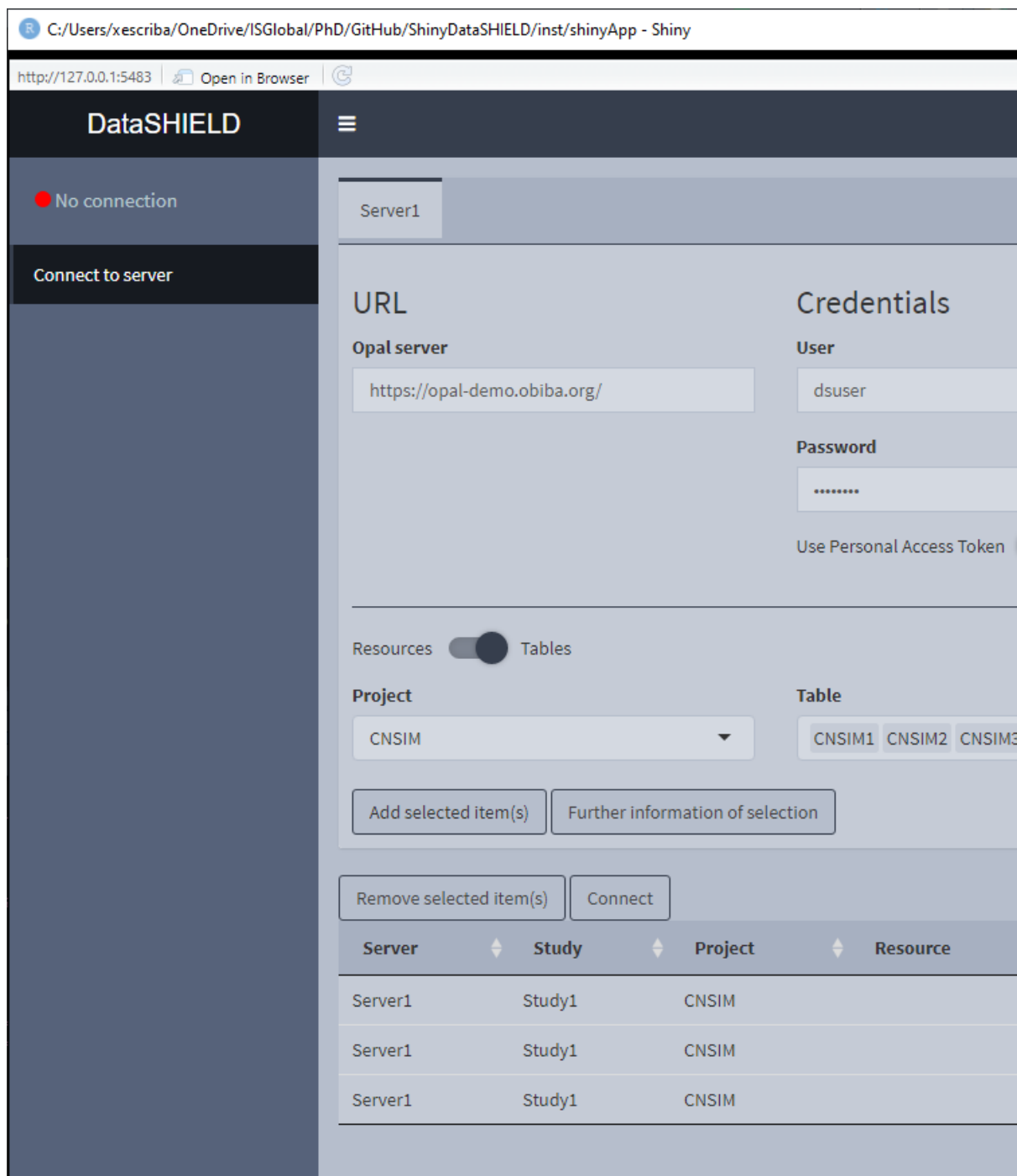


Figure 3.4: Connect to server, selected items

**DataSHIELD**

No connection

Connect to server

Server1

**URL**

Opal server

**Credentials**

User

Password

Use Personal Access Token ☐

Resources ☒ Tables

**Project**

**Table**

Server	Study	Project	Resource	Table
Server1	Study1	CNSIM		CNSIM1

Figure 3.5: Connect to server. Single resource / table approach

C:/Users/xescriba/OneDrive/ISGlobal/PhD/GitHub/ShinyDataSHIELD/inst/shinyApp - Shiny

http://127.0.0.1:5483 Open in Browser

## DataSHIELD

● No connection

Connect to server

### Server1

#### URL

Opal server

#### Credentials

User

Password

☐ Use Personal Access Token

Resources ☐ Tables

Project

Resource

Server	Study	Project	Resource
Server1	Study1	RSRC	1000G_covars
Server1	Study1	RSRC	1000G_vcf

Figure 3.6: Connect to server. Single resource / table approach



### 3.1.3 Pooled data from the same server

It is not uncommon that the same OPAL server has multiple tables that we wish to analyze using a pooled approach. To perform this kind of analysis we have to select all the tables (they have to consistent column-wise to be pooled). The important thing to take into account is that the tables need to be on different study servers in order to be analyzed using a pooled approach, in order to achieve that the user has to double click on the cells in order to edit them (Figure 3.7). There are a couple of rules regarding the naming of the study servers, 1) User inputted names can't be of the format **StudyX** (where X is a number), and 2) Similarly to what has been stated before, items from different servers can't be on the same study server.

### 3.1.4 Pooled data from different servers

When pooling data from different servers, we have to separately login to all of them. On the upper part of the interface there is a  $+$  symbol used to add a new server (Figure 3.9), when clicking the  $-$  symbol the last server added will be removed from the interface. The procedure is exactly the same as what we've already seen, the only difference is that on the table of selected items, we will now see items from different servers (Figure 3.10). Since by default items from different servers are added to different study servers, there is no need to manually configure that.

### 3.1.5 Study server profiles

Extracted from the official documentation: "A DataSHIELD profile is a R server profile combined with a DataSHIELD configuration (allowed functions, options and permissions). DataSHIELD users can then decide in which environment their analysis are to be performed, for a better reproducible science.". On the opal-demo site, there is a different array of profiles, illustrated on the Figure 3.11.

In ShinyDataSHIELD, there is the option of assigning the required profile to each study server, there is only one requirement to be taken into account: A study server can only have one distinct profile. So the configuration table of Figure 3.12 is valid, and the configuration table of Figure 3.13 will yield a connection error (Figure 3.14).

## 3.2 Table column types

TABLES USED TO DEMO THIS SECTION

From <https://opal-demo.obiba.org/> :

STUDY

TABLE

C:/Users/xescriba/OneDrive/ISGlobal/PhD/GitHub/ShinyDataSHIELD/inst/shinyApp - Shiny

http://127.0.0.1:3218 | Open in Browser

### DataSHIELD

● No connection

Connect to server

Server1

URL

Opal server

Credentials

User

Password

Use Personal Access Token ☐

Resources ☒ Tables

Project

Table

Add selected item(s)

Further information of selection

Remove selected item(s)

Connect

Server	Study	Project	Resource
Server1	<input type="text" value="Study1"/>	CNSIM	
Server1	Study1	CNSIM	
Server1	Study1	CNSIM	

Figure 3.7: Connect to server. Study server edit

The screenshot shows the DataSHIELD Shiny application interface. The browser address bar indicates the URL is `http://127.0.0.1:5483`. The sidebar on the left shows a red status indicator for 'No connection' and a 'Connect to server' button. The main panel is titled 'Server1' and contains the following sections:

- URL:** Labeled 'Opal server', with the text `https://opal-demo.obiba.org/` entered in the input field.
- Credentials:** Includes a 'User' field with 'dsuser', a 'Password' field with masked characters, and a toggle for 'Use Personal Access Token' which is currently off.
- Resources/Tables:** A toggle switch is set to 'Tables'. Below it, a 'Project' dropdown menu is set to 'CNSIM'. To the right, a 'Table' selection area shows three options: 'CNSIM1', 'CNSIM2', and 'CNSIM3'.
- Action Buttons:** 'Add selected item(s)', 'Further information of selection', 'Remove selected item(s)', and 'Connect'.
- Table:** A table at the bottom displays the selected configuration:
 

Server	Study	Project	Resource	Table
Server1	c1	CNSIM		CNSIM1
Server1	c2	CNSIM		CNSIM2
Server1	c3	CNSIM		CNSIM3

Figure 3.8: Connect to server. Pooled data from the same server approach

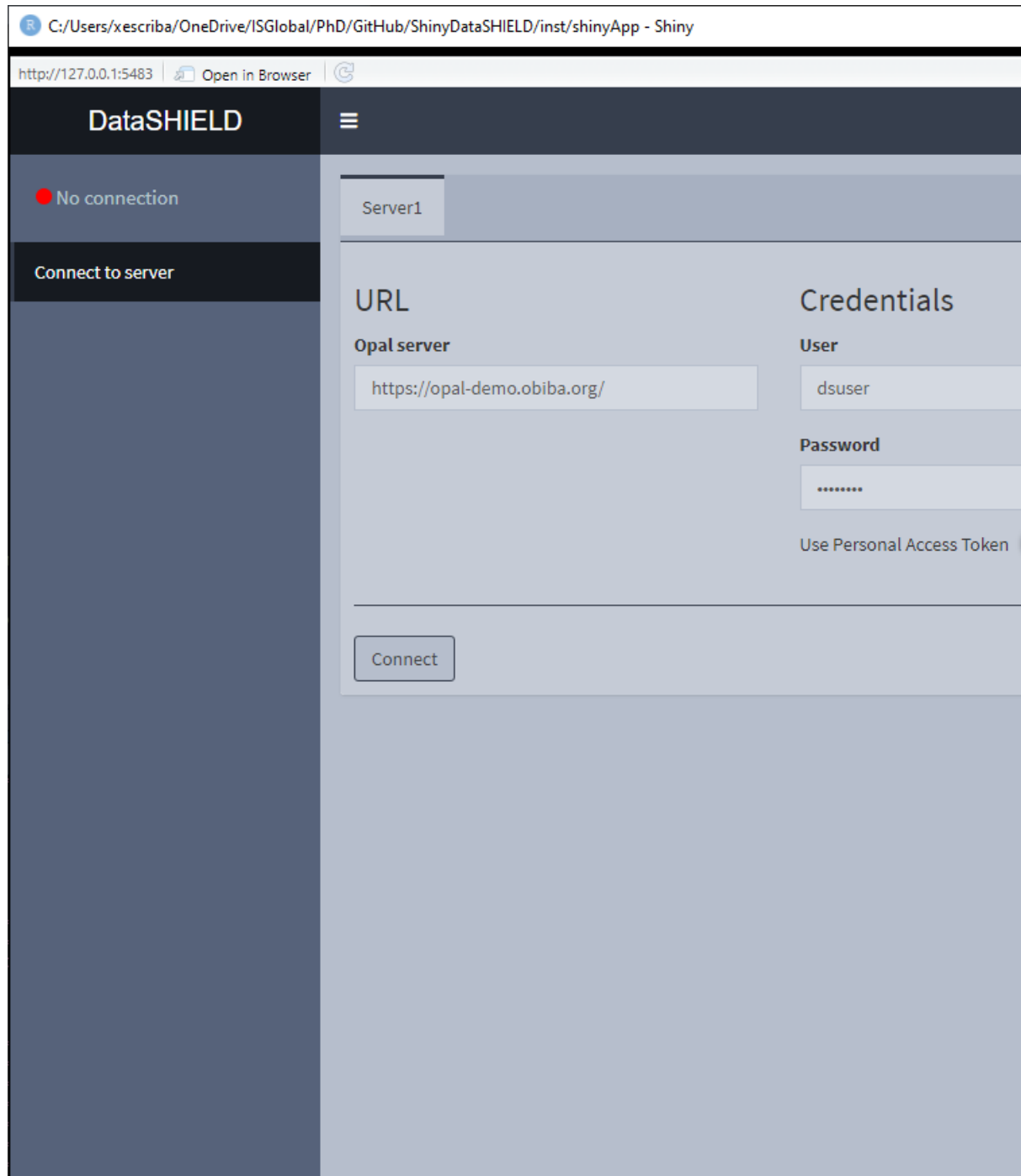


Figure 3.9: Connect to server. New server

C:/Users/xescriba/OneDrive/ISGlobal/PhD/GitHub/ShinyDataSHIELD/inst/shinyApp - Shiny

http://127.0.0.1:5483 Open in Browser

## DataSHIELD

● No connection

**Connect to server**

Server1

URL

Opal server

https://opal-demo.obiba.org/

Credentials

User

dsuser

Password

.....

Use Personal Access Token ☐

Resources ☒ Tables

Project

CNSIM

Table

CNSIM1 CNSIM2 CNSIM3

Add selected item(s)

Further information of selection

Remove selected item(s)

Connect

Server	Study	Project	Resource	Table
Server1	Study1a	CNSIM		CNSIM1
Server1	Study1b	CNSIM		CNSIM2
Server1	Study1c	CNSIM		CNSIM3

Figure 3.10: Connect to server. Pooled data from different servers approach

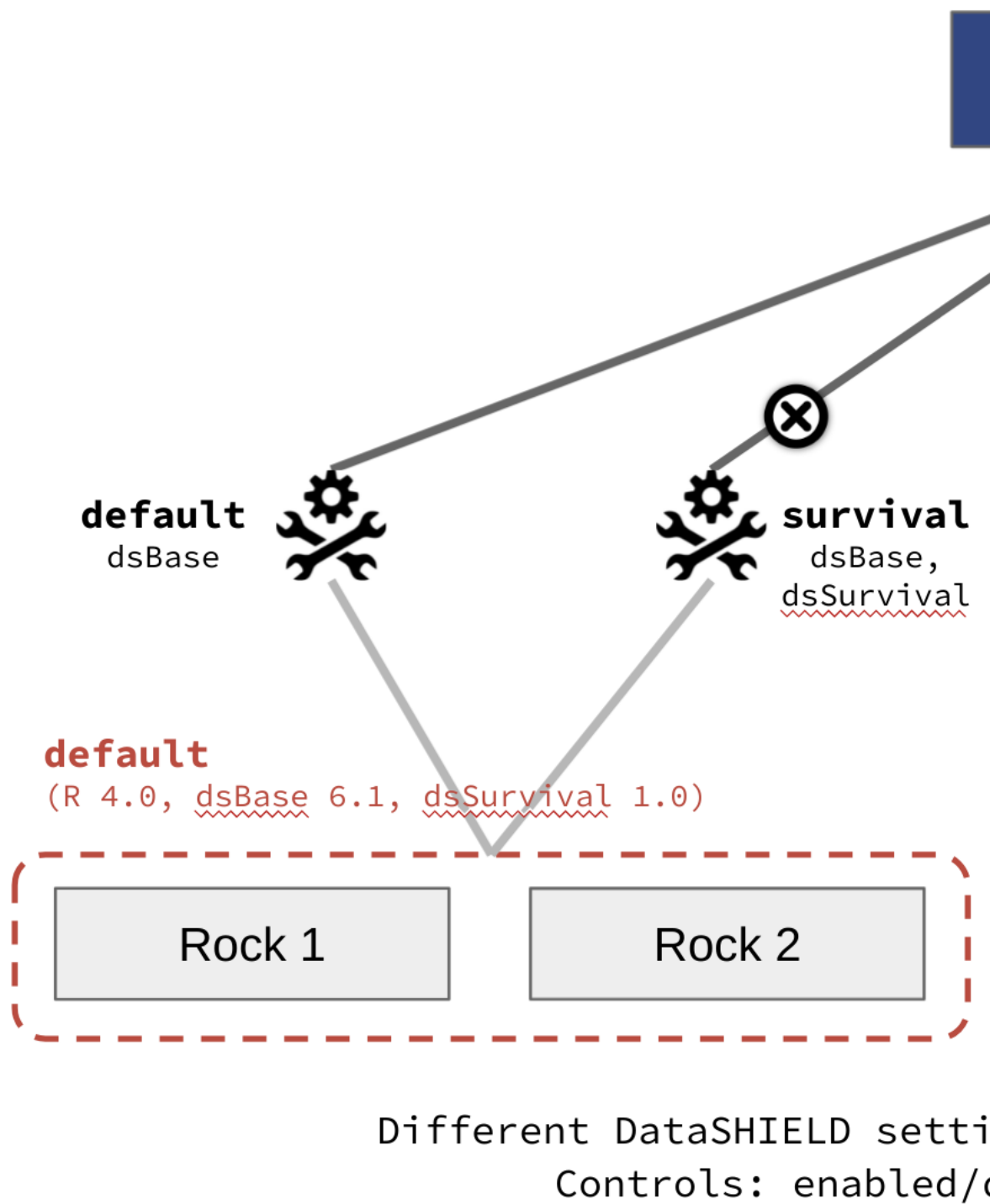


Figure 3.11: Opal demo profiles. Extracted from <https://opaldoc.obiba.org/>

C:/Users/xescriba/OneDrive/ISGlobal/PhD/GitHub/ShinyDataSHIELD/inst/shinyApp - Shiny

http://127.0.0.1:7917 Open in Browser

## DataSHIELD

● No connection

**Connect to server**

Server1

### URL

**Opal server**

### Credentials

**User**

**Password**

Use Personal Access Token ☐

Resources ☐ Tables

**Project**

Add selected item(s) Further information of selection

**Profile**

**Resource**

Remove selected item(s) Connect

Server	Study	Project	Resource	Table	P
Server1	Study1	CNSIM		CNSIM1	de
Server1	Study1	CNSIM		CNSIM2	de
Server1	Study1b	RSRC	1000G_covars		on
Server1	Study1b	RSRC	1000G_vcf		on

Figure 3.12: Correct profiles configuration

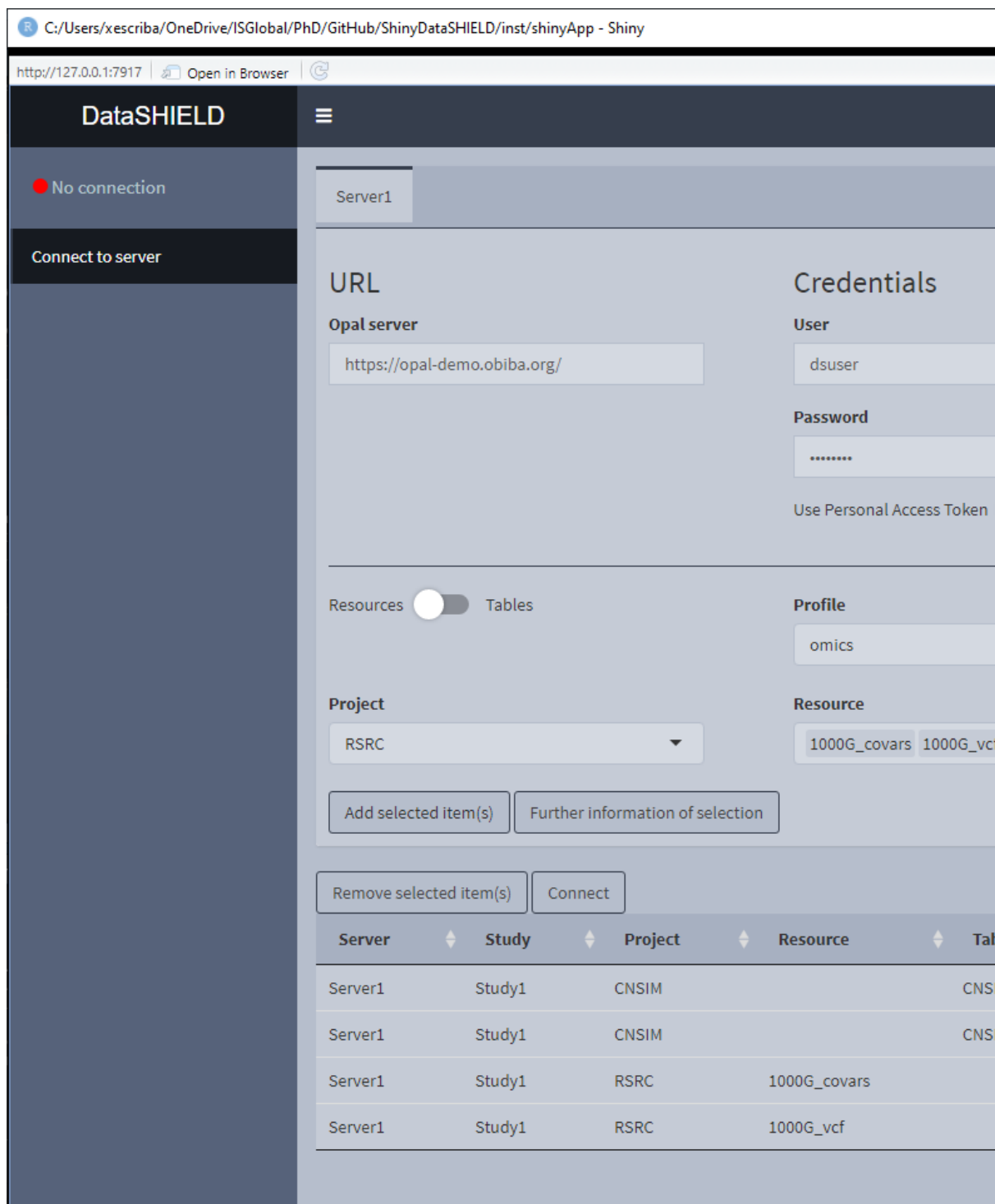


Figure 3.13: Incorrect profiles configuration



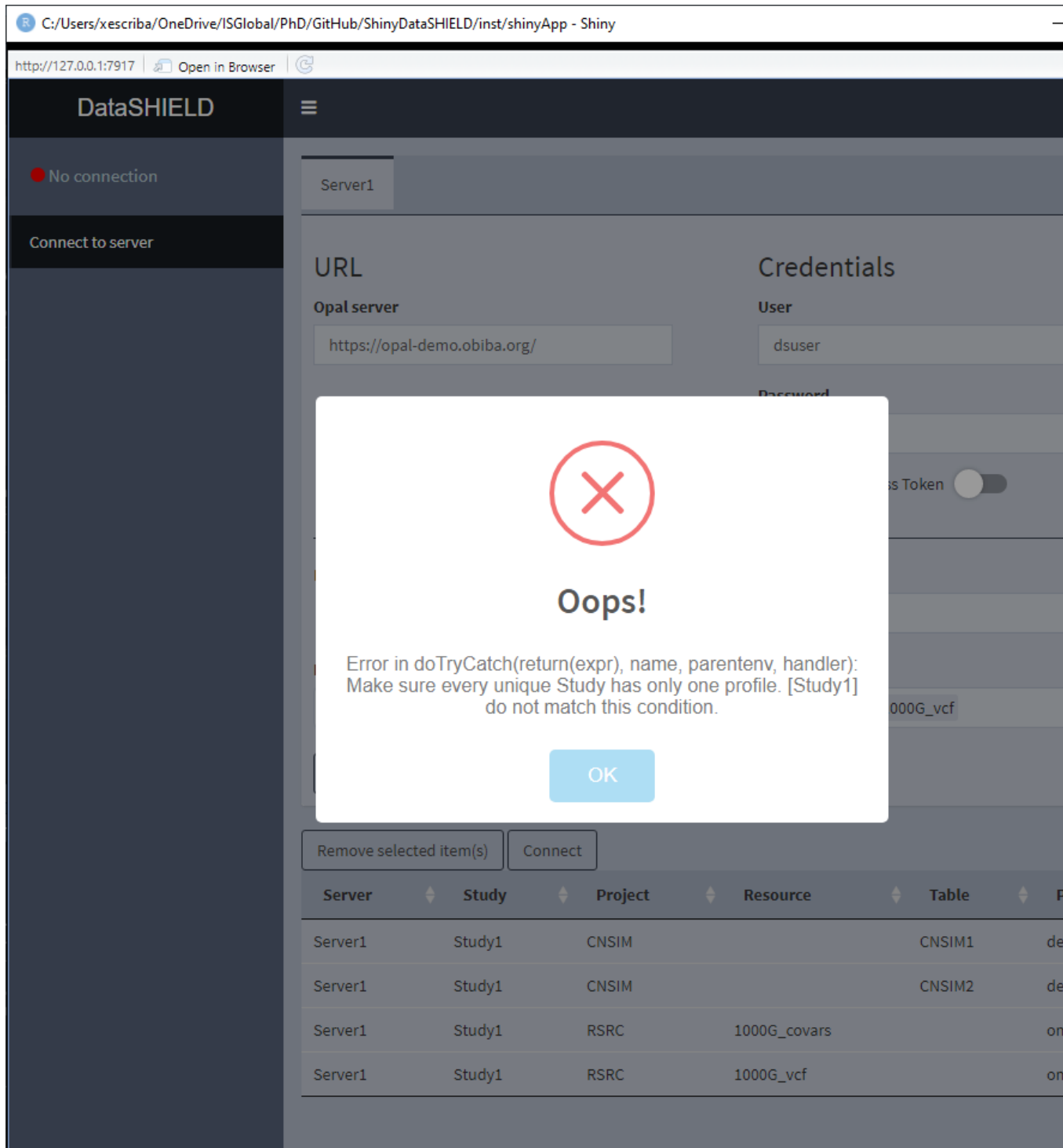


Figure 3.14: Incorrect profiles configuration error message

## PROFILE

Study1a

CNSIM.CNSIM1

default

Study1b

CNSIM.CNSIM2

default

Study1c

CNSIM.CNSIM3

default

The table column types functionality is available for tables as well as the following resource types:

- SQL tables
- Tidy data files (tables): \*.csv, \*.tsv, etc

All of the above options will be shown on the table that shows the available tables to use on this module. As stated on the use cases, to perform a pooled analysis the tables have to be on different study servers. Column integrity is checked before allowing the user the access to the other tabs.

The function of this module is to get information about the class of each column of a table.

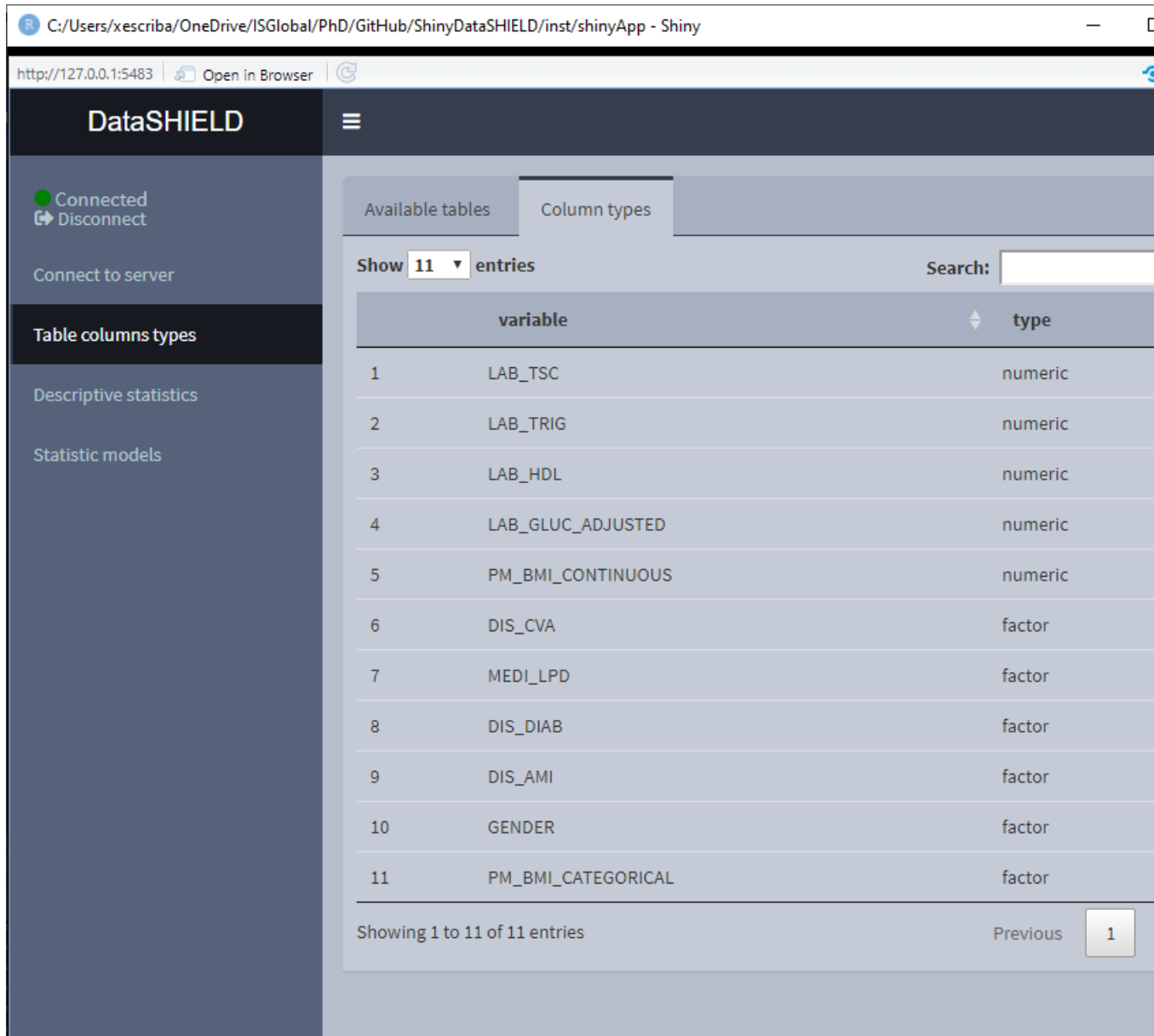
Aside from that, there's the option of changing the class of a column. This transformation is done using the proper DataSHIELD functions that perform disclosive checks before allowing the transformation. This is specially important for transformation from numeric to factor for example.

The allowed classes to perform transformations are:

- Character
- Numeric
- Factor

To perform a class change, double click on the desired row and a drop-down menu will appear, choose the new class and click 'Confirm', after the checks, the table will be updated to display the new class.

Integer could be added if it was of interest. Please file an issue on GitHub if that is the case.



The screenshot displays the DataSHIELD web application. The browser address bar shows the URL `http://127.0.0.1:5483`. The application header includes the DataSHIELD logo and a hamburger menu icon. The sidebar on the left contains the following elements:

- Connected (with a green status indicator) and Disconnect button.
- Connect to server button.
- Table columns types** (selected).
- Descriptive statistics.
- Statistic models.

The main content area has two tabs: 'Available tables' and 'Column types'. The 'Column types' tab is active, showing a table with 11 entries. The table has columns for 'variable' and 'type'. A 'Show 11 entries' dropdown and a 'Search:' input field are located above the table. The table lists the following variables and their types:

	variable	type
1	LAB_TSC	numeric
2	LAB_TRIG	numeric
3	LAB_HDL	numeric
4	LAB_GLUC_ADJUSTED	numeric
5	PM_BMI_CONTINUOUS	numeric
6	DIS_CVA	factor
7	MEDI_LPD	factor
8	DIS_DIAB	factor
9	DIS_AMI	factor
10	GENDER	factor
11	PM_BMI_CATEGORICAL	factor

At the bottom of the table, it says 'Showing 1 to 11 of 11 entries'. There are 'Previous' and '1' (current page) navigation buttons.

Figure 3.15: Classes of the CNSIM table

The screenshot shows the DataSHIELD web application interface. The browser address bar indicates the URL is `http://127.0.0.1:5483`. The application title is "DataSHIELD". The left sidebar contains navigation options: "Connected" (with a green dot and "Disconnect" button), "Connect to server", "Table columns types" (highlighted), "Descriptive statistics", and "Statistic models". The main content area has two tabs: "Available tables" and "Column types". Under "Available tables", there is a "Show 11 entries" dropdown and a "Search:" input field. Below this is a table listing 11 variables:

	variable	ty
1	LAB_TSC	num
2	LAB_TRIG	num
3	LAB_HDL	num
4	LAB_GLUC_ADJUSTED	num
5	PM_BMI_CONTINUOUS	num
6	DIS_CVA	factor
7	MEDI_LPD	factor
8	DIS_DIAB	factor
9	DIS_AMI	factor
10	GENDER	factor
11	PM_BMI_CATEGORICAL	factor

At the bottom of the table, it says "Showing 1 to 11 of 11 entries" and "Previous" (partially visible).

Figure 3.16: Class change

### 3.3 Descriptive statistics

TABLES USED TO DEMO THIS SECTION

From <https://opal-demo.obiba.org/> :

STUDY

TABLE

PROFILE

Study1a

CNSIM.CNSIM1

default

Study1b

CNSIM.CNSIM2

default

Study1c

CNSIM.CNSIM3

default

The descriptive statistics functionality is available for tables as well as the following resource types:

- SQL tables
- Tidy data files (tables): \*.csv, \*.tsv, etc

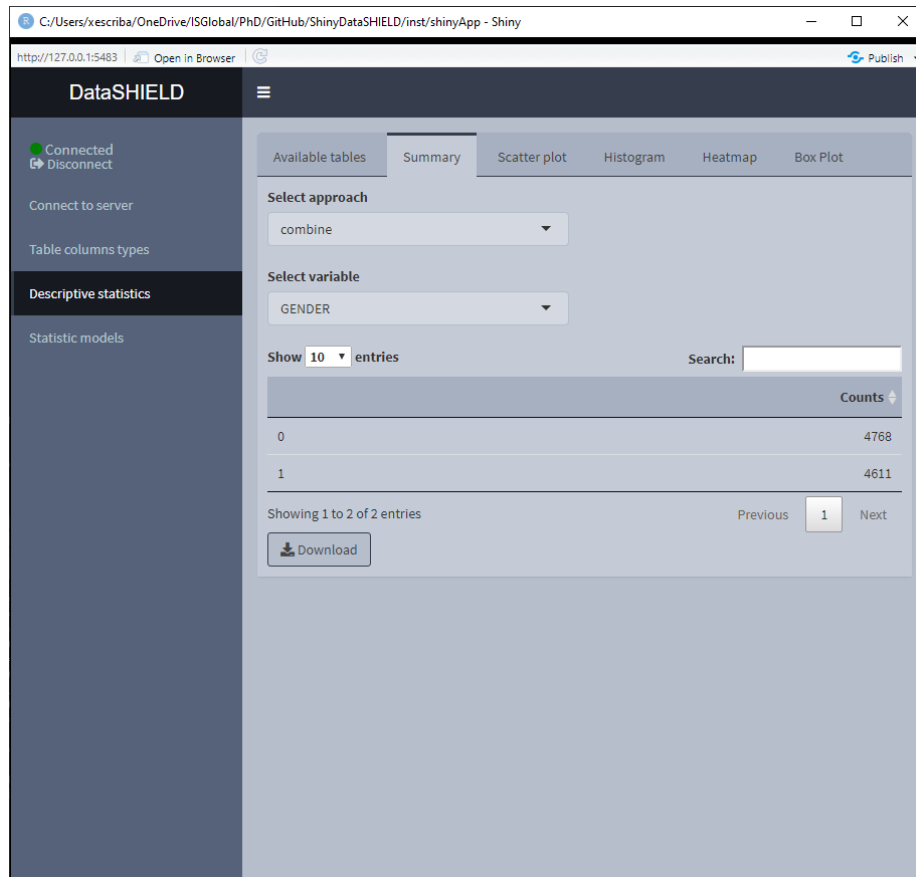
All of the above options will be shown on the table that shows the available tables to use on this module. As stated on the use cases, to perform a pooled analysis the tables have to be on different study servers. Column integrity is checked before allowing the user the access to the other tabs.

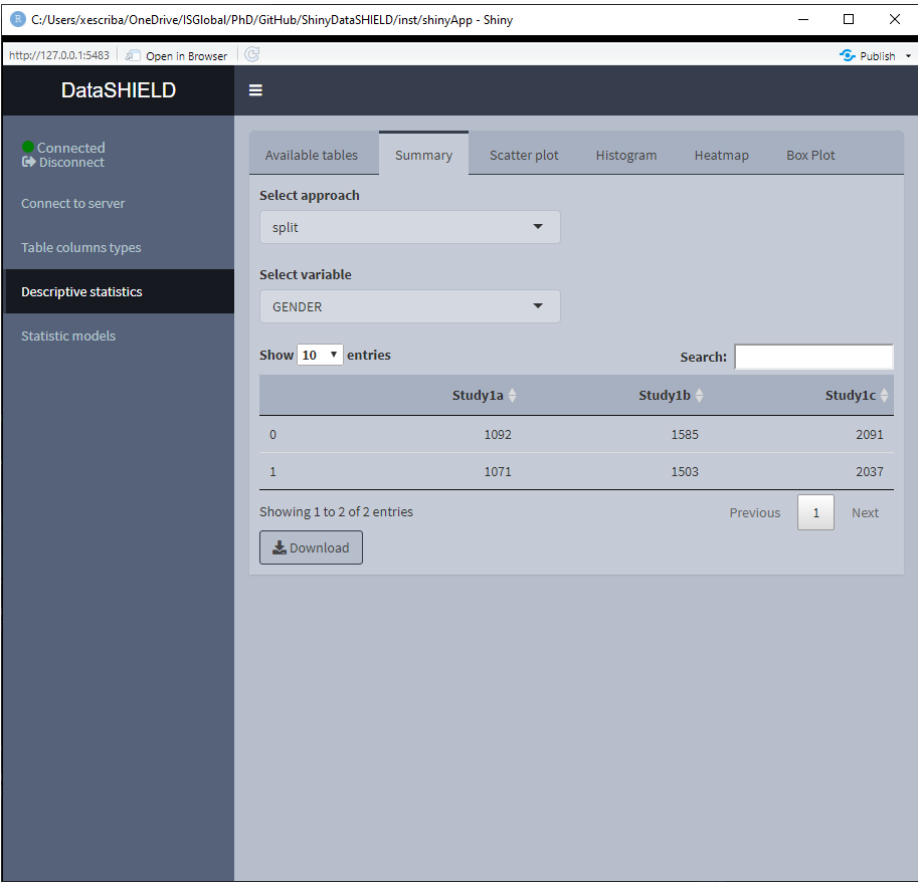
When using pooled data the descriptive statistics is by default of the pooled data, however, the graphical visualizations included on descriptive statistics provide the option of showing separated plots for the different studies.

#### 3.3.1 Summary

The summary provides non-disclosive insights on the different variables of the loaded data. This functionality is only available for factors and numeric variables, only variables that meet this criteria will be on the selector. When the desired summary is disclosive no table will be shown (as the function call returns an Error stating that the the return is disclosive).

When the selected variable is a factor, the output shown is a count of all the different factors. It can be visualized with the pooled data or divided by study servers.





When the selected variable is numerical, the output shown is a quantiles and mean table. It can be visualized with the pooled data or divided by study servers.

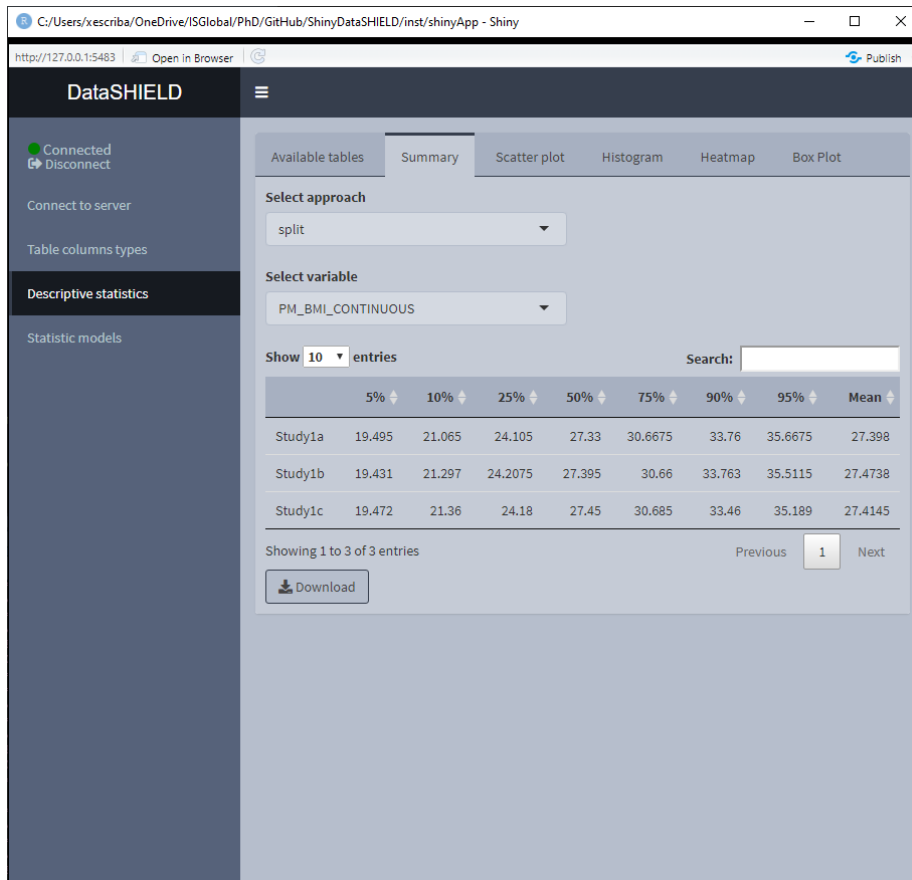
The screenshot shows the DataSHIELD Shiny application interface. The browser address bar indicates the URL is `http://127.0.0.1:5483`. The application title is "DataSHIELD". The left sidebar contains the following options: "Connected" (with a green dot and "Disconnect" button), "Connect to server", "Table columns types", "Descriptive statistics" (highlighted), and "Statistic models". The main panel has tabs for "Available tables", "Summary" (selected), "Scatter plot", "Histogram", "Heatmap", and "Box Plot".

Under the "Summary" tab, the "Select approach" dropdown is set to "combine". The "Select variable" dropdown is set to "PM\_BMI\_CONTINUOUS". The "Show" dropdown is set to "10" entries. A search bar is present with the text "Search:". Below this is a table with a header row labeled "data" and a downward arrow. The table contains 8 rows of data:

	data
5%	19,4638
10%	21,271
25%	24,1717
50%	27,4041
75%	30,6727
90%	33,6292
95%	35,4059
Mean	27,4302

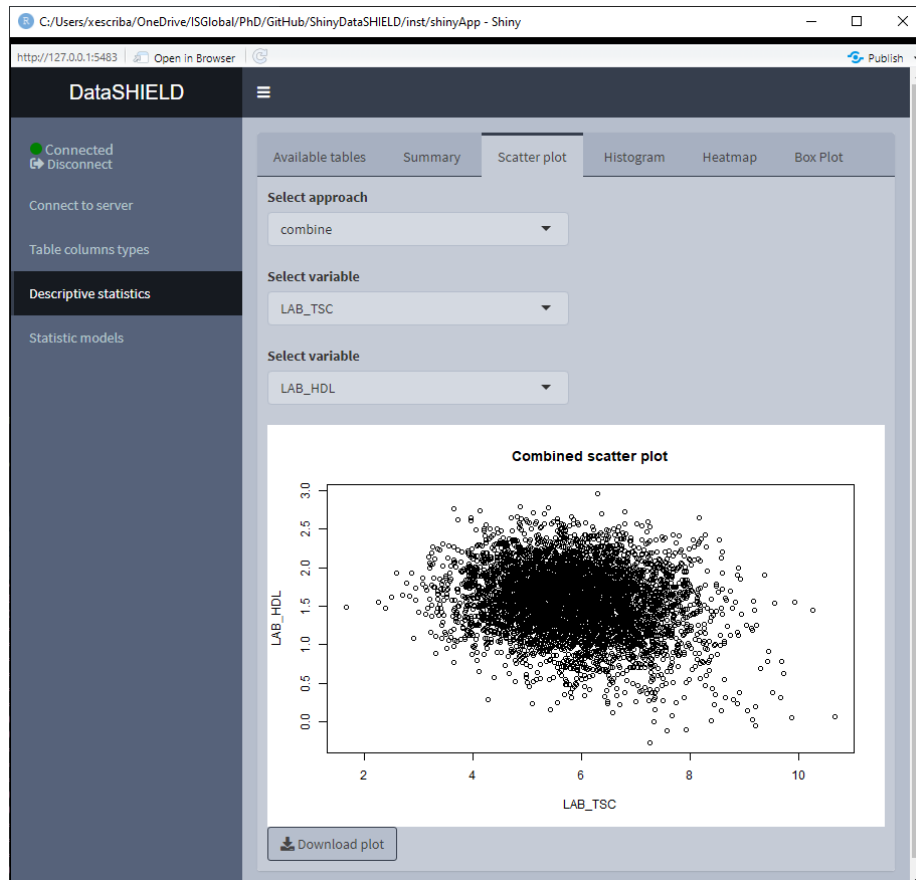
At the bottom of the table, it says "Showing 1 to 8 of 8 entries". There are "Previous", "1" (highlighted), and "Next" buttons. A "Download" button is also present.

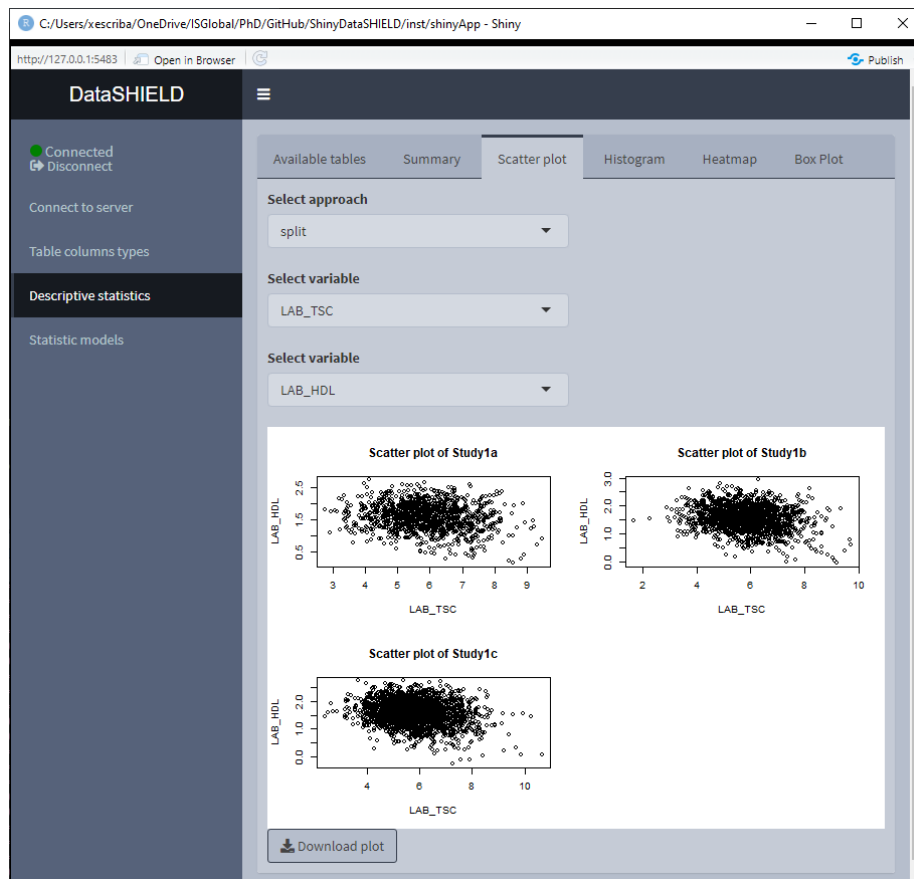




### 3.3.2 Scatter plot

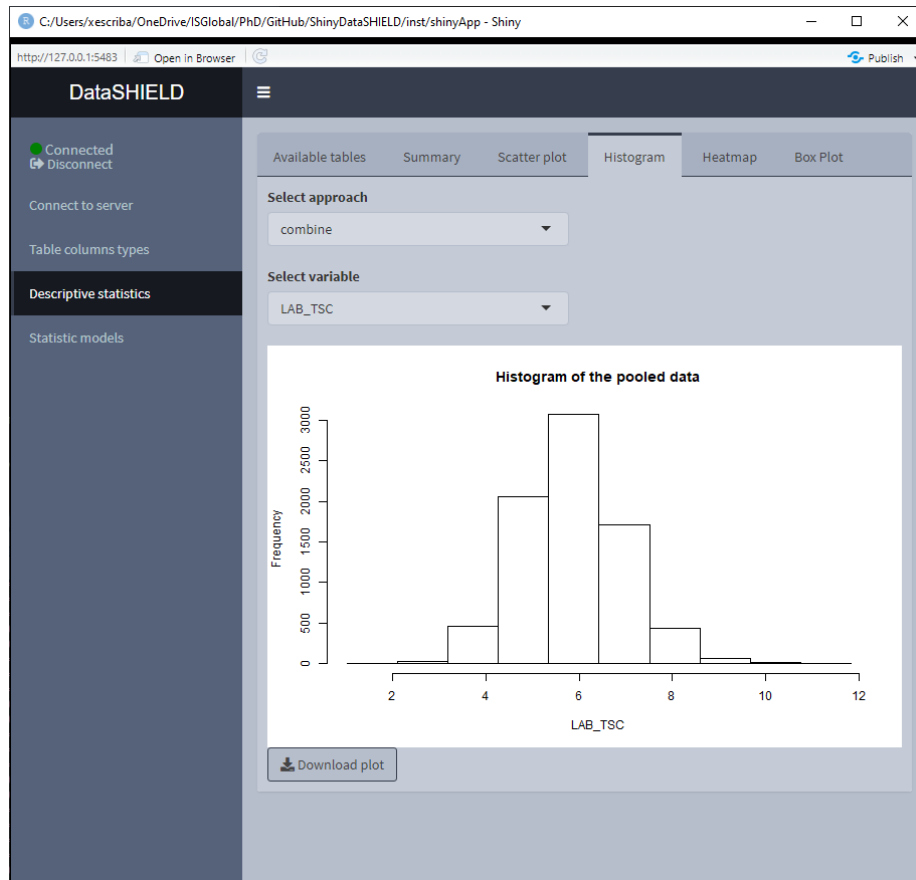
Create a non-disclosive scatter plot by selecting two numerical variables (one for each axis). This type of plot can only be generated using numerical variables, for that reason variables that do not meet this criteria are not shown on the selector. It can be visualized with the pooled data or divided by study servers.

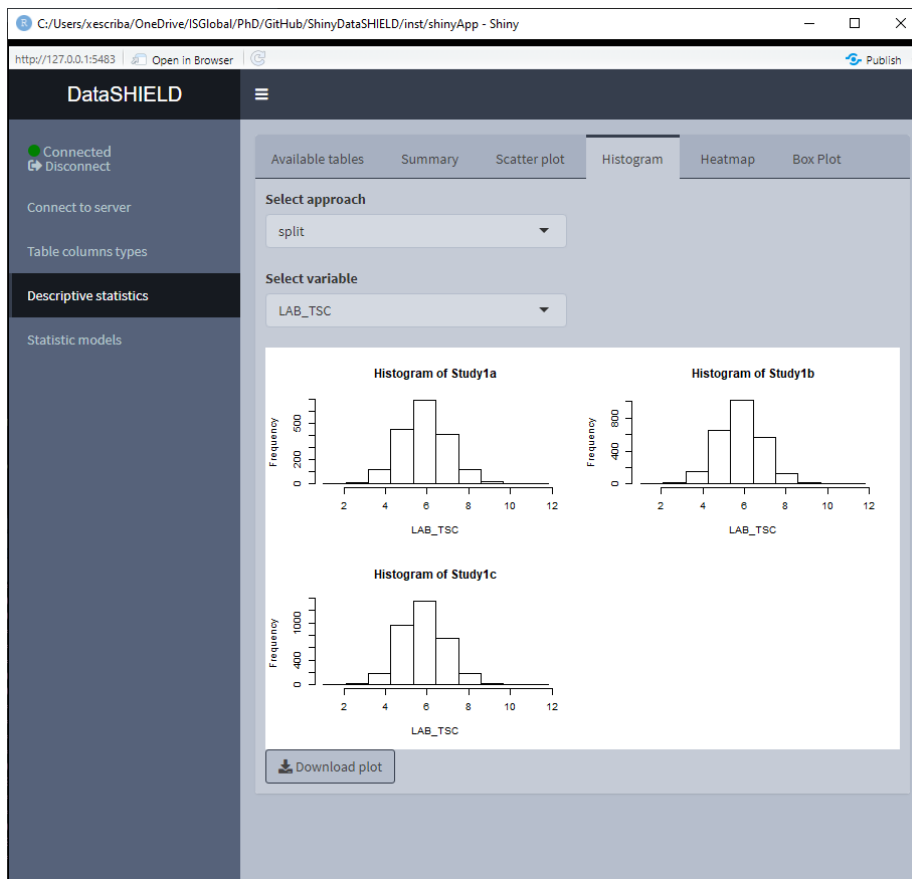




### 3.3.3 Histogram

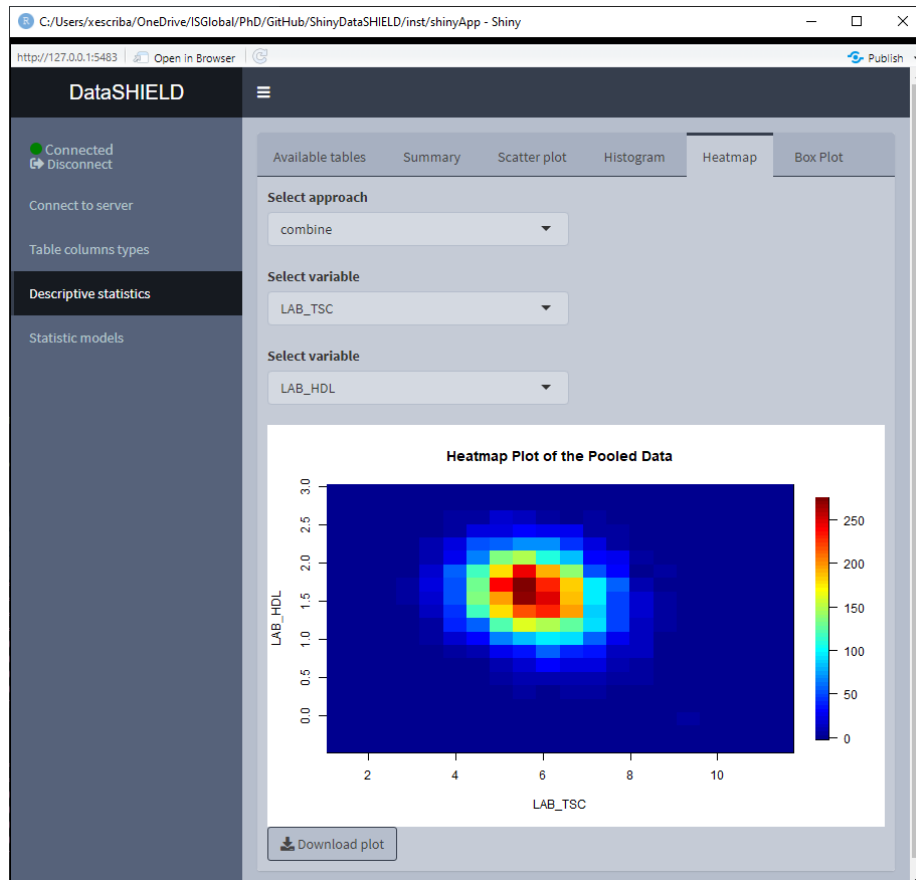
Create a non-disclosive histogram of a selected variable. This type of plot can only be generated using numerical variables, for that reason variables that do not meet this criteria are not shown on the selector. It can be visualized with the pooled data or divided by study servers.

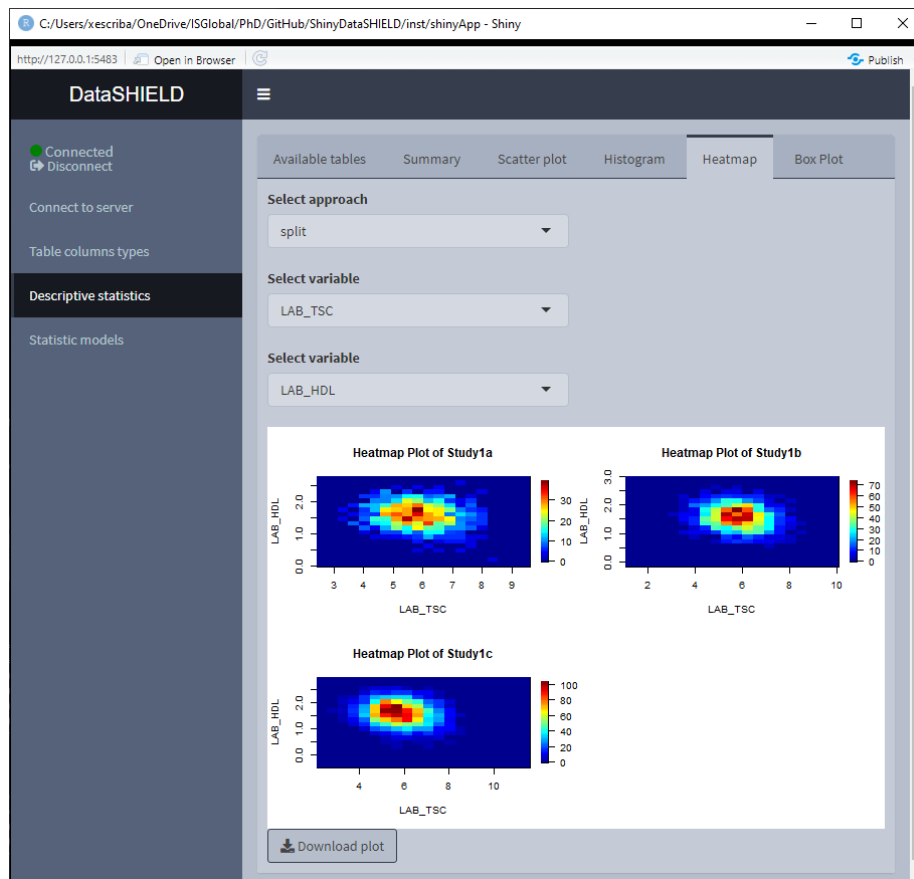




### 3.3.4 Heatmap

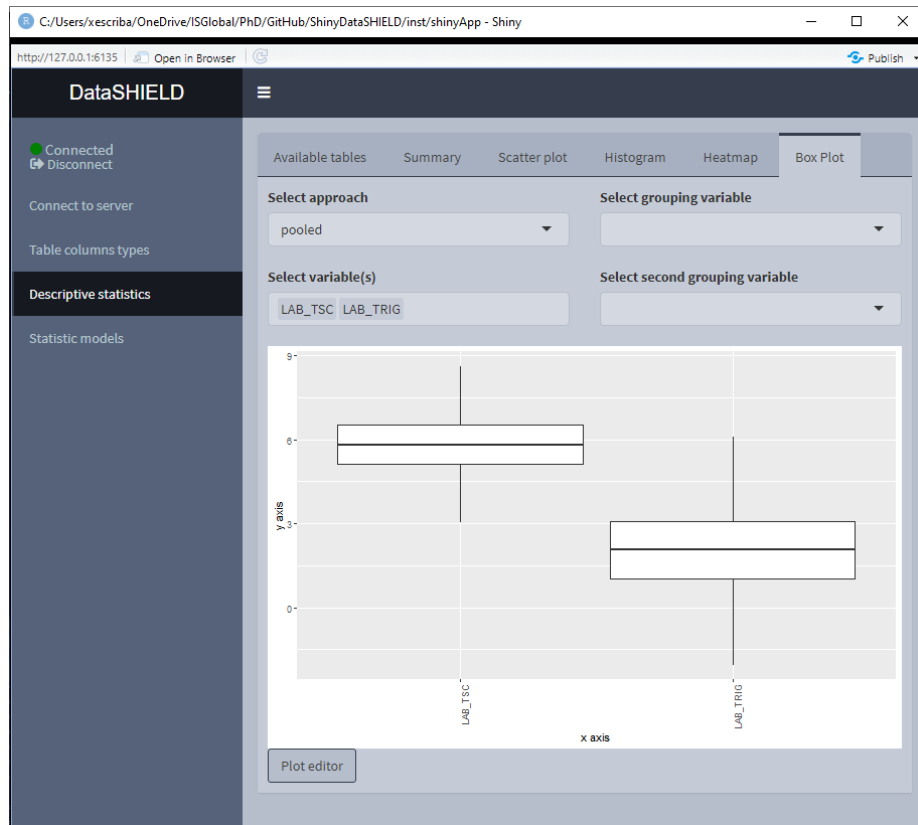
Create a non-disclosive heatmap plot by selecting two numerical variables (one for each axis). This type of plot can only be generated using numerical variables, for that reason variables that do not meet this criteria are not shown on the selector. It can be visualized with the pooled data or divided by study servers.



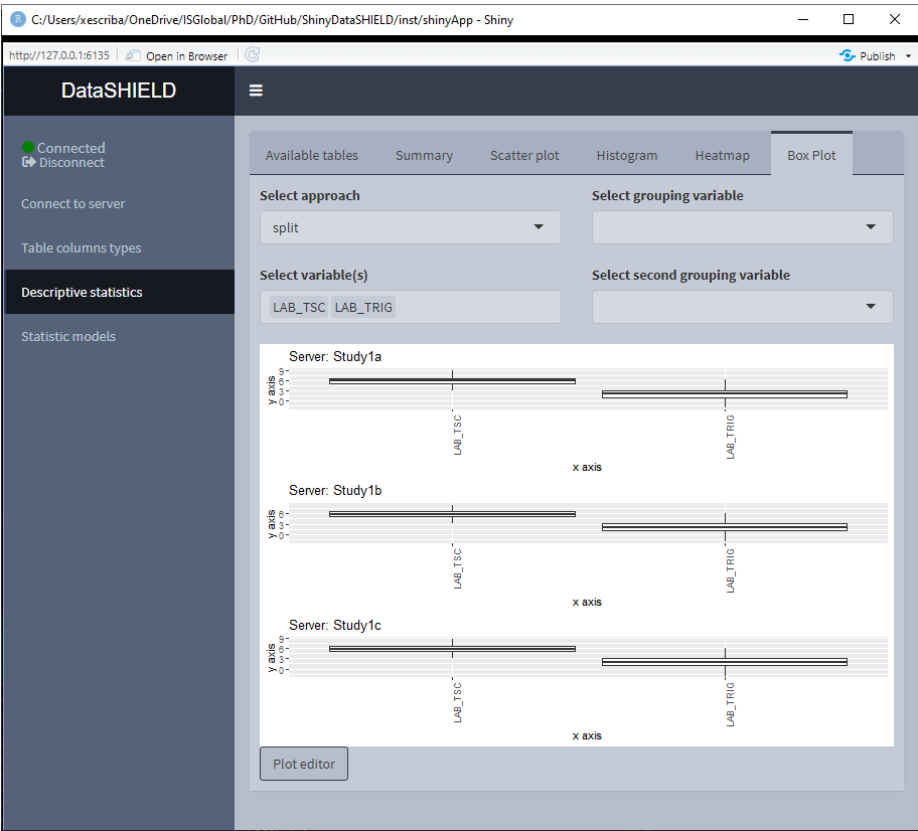


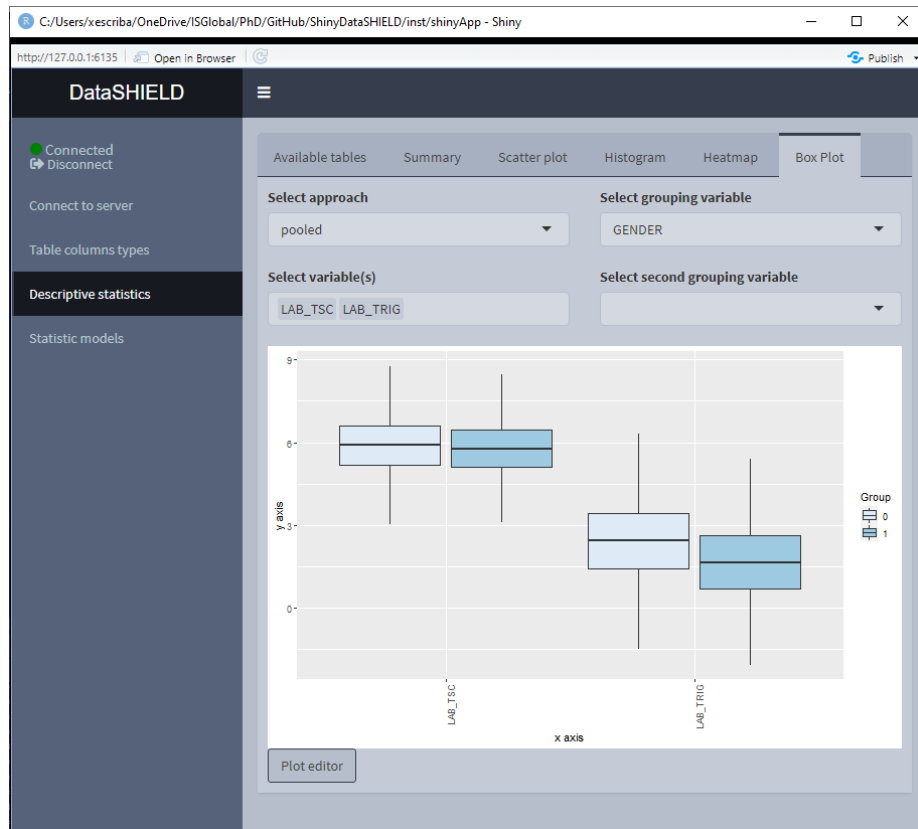
### 3.3.5 Boxplot

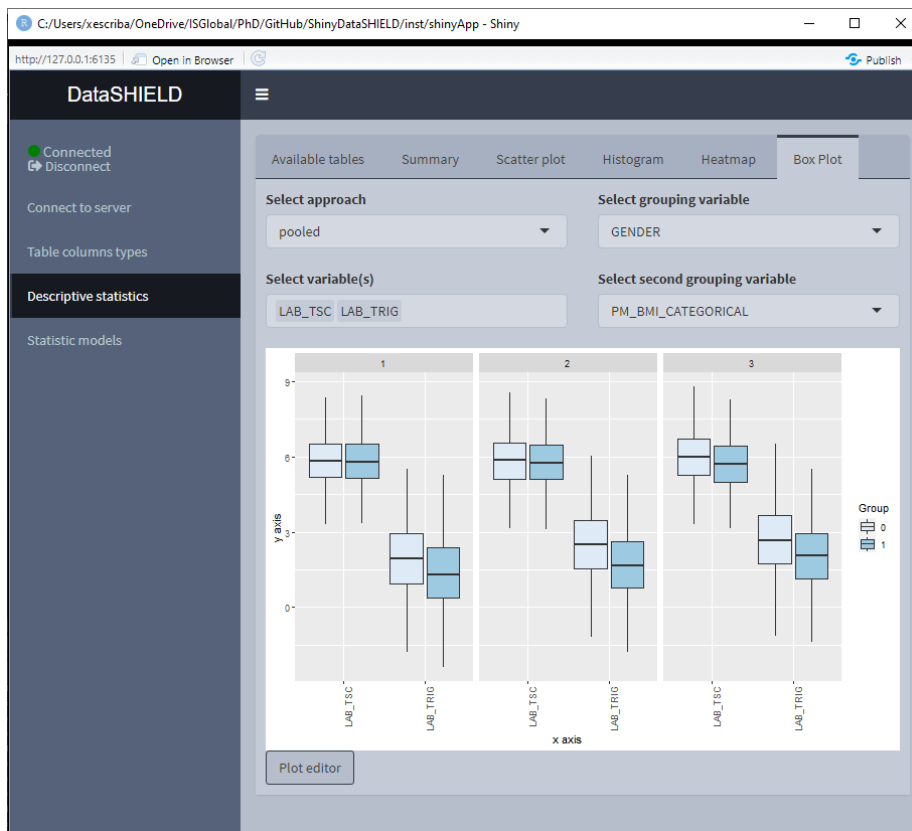
Create a non-disclosive Boxplot by selecting as many numerical variables as desired. This plot has the option of performing groupings using one or two factor variables from the same table. It can be visualized with the pooled data or divided by study servers.



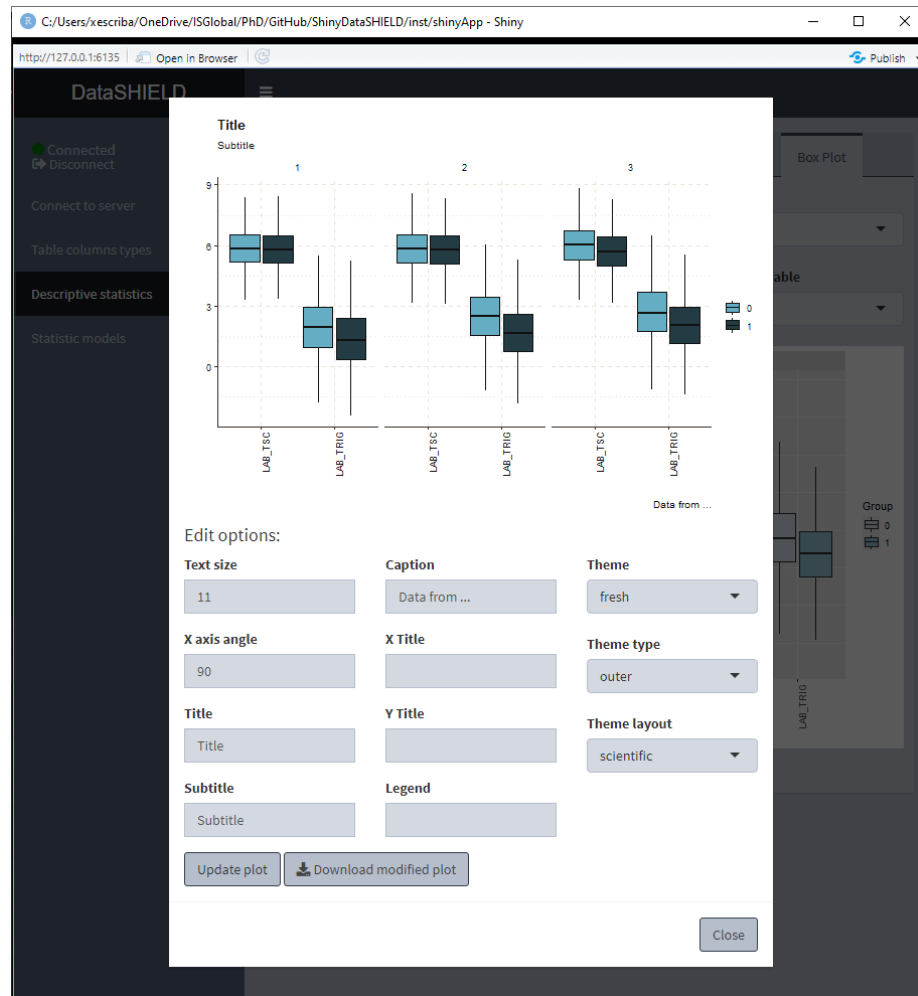








The Boxplot functionality uses `ggplot2`; a novel in-app editor named `ggEditLite` has been introduced inside the Shiny application to allow simple graphic modifications of the plot to adapt it to the style of choice, add title, subtitle, etc. This feature will be rolled out to all the plots as they get upgraded to make use of the `ggplot2` package.



### 3.4 Statistical models

TABLES USED TO DEMO THIS SECTION

From <https://opal-demo.obiba.org/> :

GLM and mixed models

STUDY

TABLE

PROFILE

Study1a

CNSIM.CNSIM1

default

Study1b

CNSIM.CNSIM2

default

Study1c

CNSIM.CNSIM3

default

Survival analysis

STUDY

TABLE

PROFILE

Study1a

SURVIVAL.EXPAND\_WITH\_MISSING1

survival

Study1b

SURVIVAL.EXPAND\_WITH\_MISSING2

survival

Study1c

SURVIVAL.EXPAND\_WITH\_MISSING3

survival

Statistic models are available for tables as well as the following resource types:

- SQL tables
- Tidy data files (tables): \*.csv, \*.tsv, etc

There are three different statistical models available to fit, GLM models (Statistics models tab), GLMer models (Mixed statistical models tab) and Survival cox models (Survival analysis tab).

### 3.4.1 GLM models

The tab to fit a non-disclosive generalized linear models (GLM) contains a box to manually input the formula, a selector for the output family and a table displaying the variables of the data and the type of each variable. There is

finally the option to perform a pooled analysis or a meta-study. The possible output families are:

- Gaussian
- Poisson
- Binomial

There's some help built into ShinyDataSHIELD regarding how to write the GLM formula, which is prompted to the user when clicking on the “Formula input help” button. The display of the variables can be toggled on and off for the convenience of use.

Once the GLM model is fitted a table below the variables display will be rendered with the model results. The download button will prompt a system window to select where to store the shown table, it will save it as a **\*.csv**.

When using pooled data, the results of the GLM model will be of the combined data.

(To do: Display more information of why a model fitment fails)

The screenshot shows the ShinyDataSHIELD web application interface. The browser address bar indicates the URL is `http://127.0.0.1:5483`. The application has a dark blue sidebar on the left with the following menu items: "Connected" (with a green dot and "Disconnect" button), "Connect to server", "Table columns types", "Descriptive statistics", and "Statistic models" (highlighted in black). The main content area has a top navigation bar with tabs: "Available tables", "Generalized linear model" (active), "Mixed effects model", and "Survival analysis". Below the tabs, the "Generalized linear model" section contains:
 

- Input GLM formula:** A text box containing `PM_BMI_CATEGORICAL~LAB_TSC+LAB_TRIG`.
- Output family:** A dropdown menu set to "binomial".
- Select the approach:** A dropdown menu set to "Pooled".
- A "Toggle variables table" button.
- Buttons for "Formula input help" and "Perform GLM".

 Below these controls is a table of model results:
 

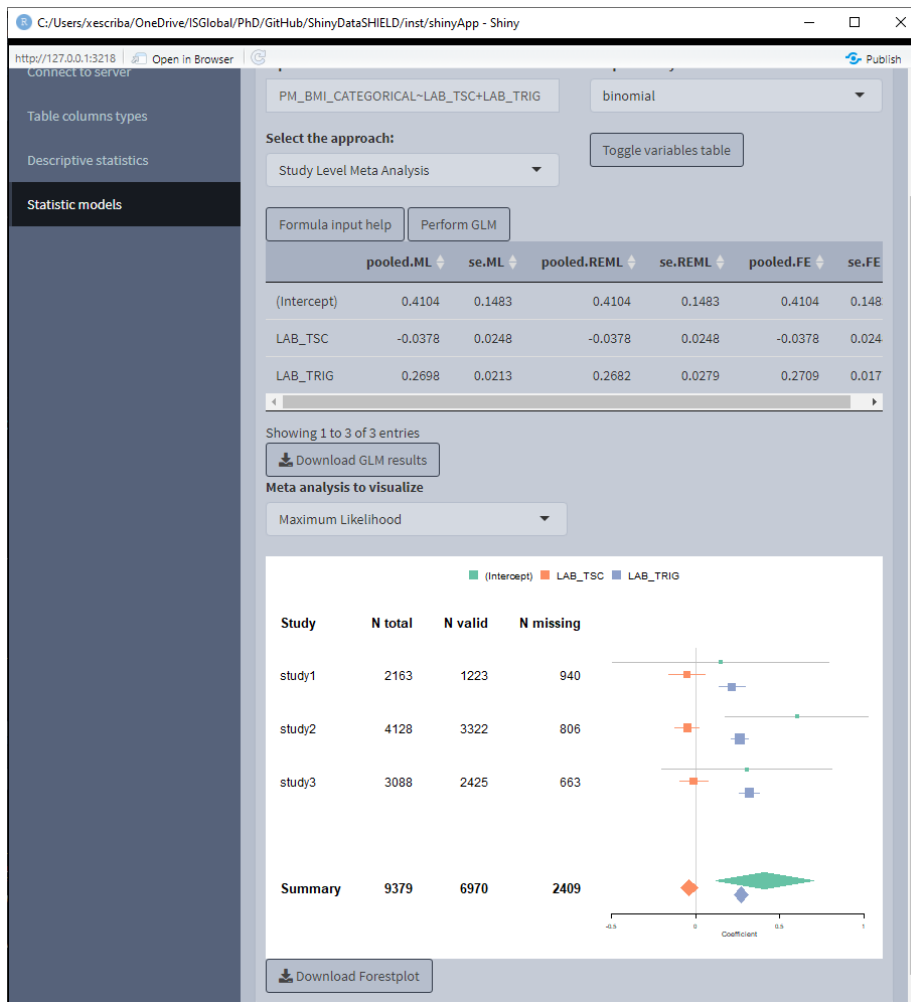
	Estimate	Std. Error	z-value	P-value	low0.95CI.LP	high0.95CI.LP
(Intercept)	0.4295	0.1476	2.9106	0.0036	0.1403	0.7187
LAB_TSC	-0.0398	0.0247	-1.616	0.1061	-0.0882	0.0085
LAB_TRIG	0.2707	0.0176	15.3976	0	0.2362	0.3052

 At the bottom of the results section, it says "Showing 1 to 3 of 3 entries" and includes a "Download GLM results" button with a download icon.

When using a meta-study approach the results correspond to the betas and standard errors for three different meta study methodologies:

- Maximum Likelihood
- REstricted Maximum Likelihood
- Fixed-Effects meta-analysis

These three methodologies can be visualized on a forest plot by selecting the desired one.



### 3.4.2 Mixed models

The tab to fit non-disclosive generalized mixed effects models (GLMer) contains a box to manually input the formula, a selector for the output family and a table displaying the variables of the data and the type of each variable. The

possible output families are:

- Poisson
- Binomial

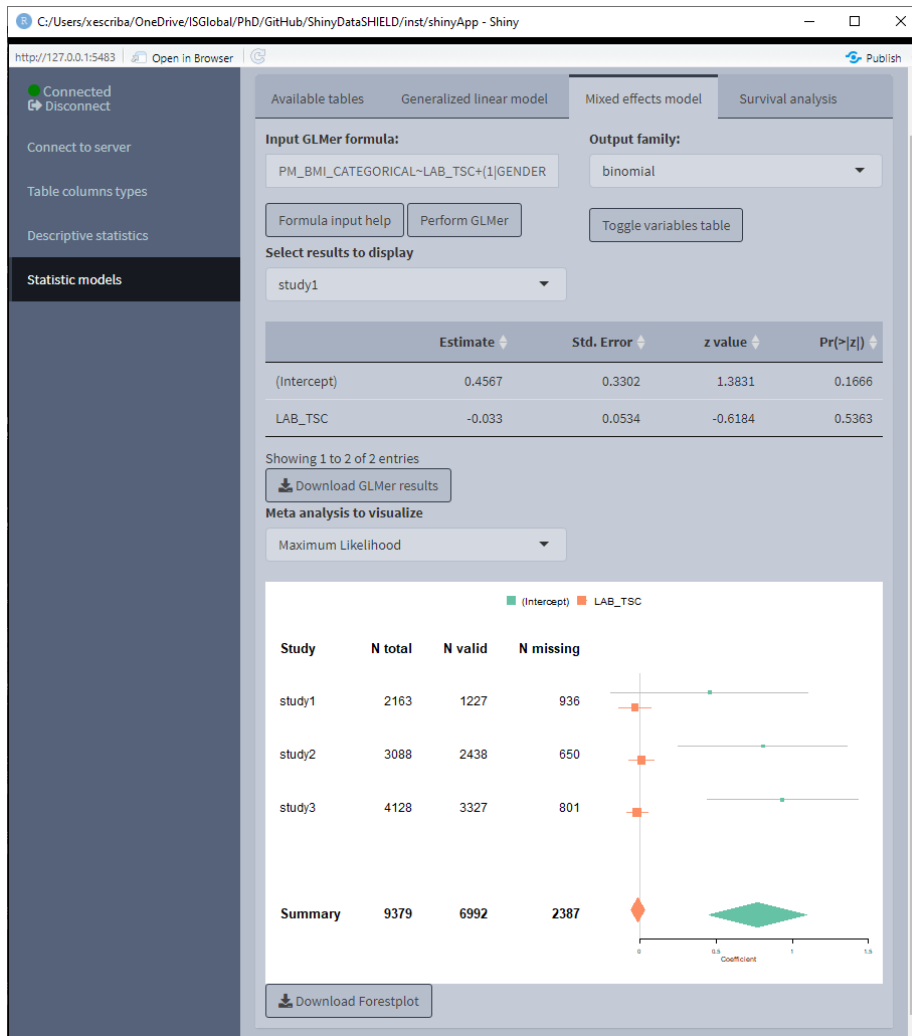
There's some help built into ShinyDataSHIELD regarding how to write the GLMer formula, which is prompted to the user when clicking on the "Formula input help" button. The display of the variables can be toggled on and off for the convenience of use.

Once the GLMer model is fitted a table below the variables display will be rendered displaying the results. The download button will prompt a system window to select where to store the shown table, it will save them as a `*.csv`.

The mixed model results are independent for each study server. There's a selector to toggle between the results of the different study servers.

(To do: Display more information of why a model fitment fails)





### 3.4.3 Survival Analysis

The tab to fit non-disclosive survival analysis is divided into four different sub-tabs. The first subtab, is used to create a survival object (See the information about `survival::Surv` function for more information <https://www.rdocumentation.org/packages/survival/versions/2.11-4/topics/Surv>). To create this object, three columns from the selected tables are needed and a parameter:

(Information about each column copied from the `survival::Surv` function documentation)

- Start time variable: for right censored data, this is the follow up time. For interval data, the first argument is the starting time for the interval.

- End time variable: ending time of the interval for interval censored or counting process data only. Intervals are assumed to be open on the left and closed on the right, `(start, end]`. For counting process data, event indicates whether an event occurred at the end of the interval.
- Event variable: The status indicator, normally 0=alive, 1=dead. Other choices are `TRUE/FALSE` (`TRUE` = death) or 1/2 (2=death). For interval censored data, the status indicator is 0=right censored, 1=event at `time` (Start time variable), 2=left censored, 3=interval censored. For multiple endpoint data the event variable will be a factor, whose first level is treated as censoring. Although unusual, the event indicator can be omitted, in which case all subjects are assumed to have an event.
- Type of censoring (parameter): character string specifying the type of censoring. Possible values are “`right`”, “`left`”, “`counting`”, “`interval`”, “`interval2`” or “`mstate`”.

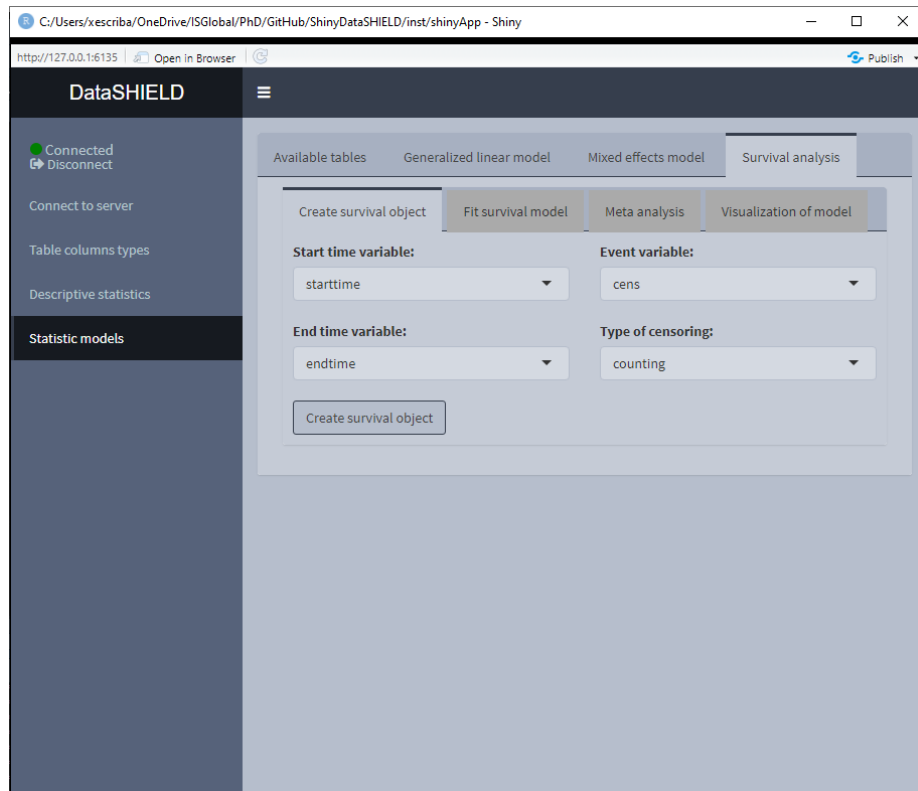
For the data used on this demo, the columns are the following:

- Start time variable: `starttime`
- End time variable: `endtime`
- Event variable: `cens`
- Type of censoring (parameter): `counting`

We have to make sure all three columns are numeric on this demo:

The screenshot shows the DataSHIELD web application interface. The browser address bar displays the URL `http://127.0.0.1:6135`. The application has a dark sidebar on the left with the following menu items: "Connected" (with a green status icon and a "Disconnect" link), "Connect to server", "Table columns types" (highlighted), "Descriptive statistics", and "Statistic models". The main content area has two tabs: "Available tables" and "Column types". Under the "Column types" tab, there is a "Show 12 entries" dropdown and a "Search:" input field. Below this is a table with two columns: "variable" and "type". The table contains 12 rows of data. The 7th row, for the variable "cens", has a dropdown menu open showing options: "factor", "numeric" (highlighted), "factor", and "character". To the right of this dropdown are "Confirm" and "Cancel" buttons. At the bottom of the table, it says "Showing 1 to 12 of 12 entries" and has "Previous", "1", and "Next" navigation controls.

	variable	type
1	id	integer
2	study.id	integer
3	time.id	integer
4	starttime	numeric
5	endtime	numeric
6	survtime	numeric
7	cens	factor
8	age.60	numeric
9	female	factor
10	noise.56	numeric
11	pm10.16	numeric
12	bmi.26	numeric



Once the survival object is created, the tab ‘Fit survival model’ is unlocked. On this tab, as with the other statistic models, there is a help button and a ‘Toggle variables tables’ to visualize the variables available to construct the model. The formula has the following structure:

```
survival_object~tables_sm$variable+tables_sm$variable2
```

Where `survival_object` and `tables_sm` can’t be modified. An example model for this demo data would be:

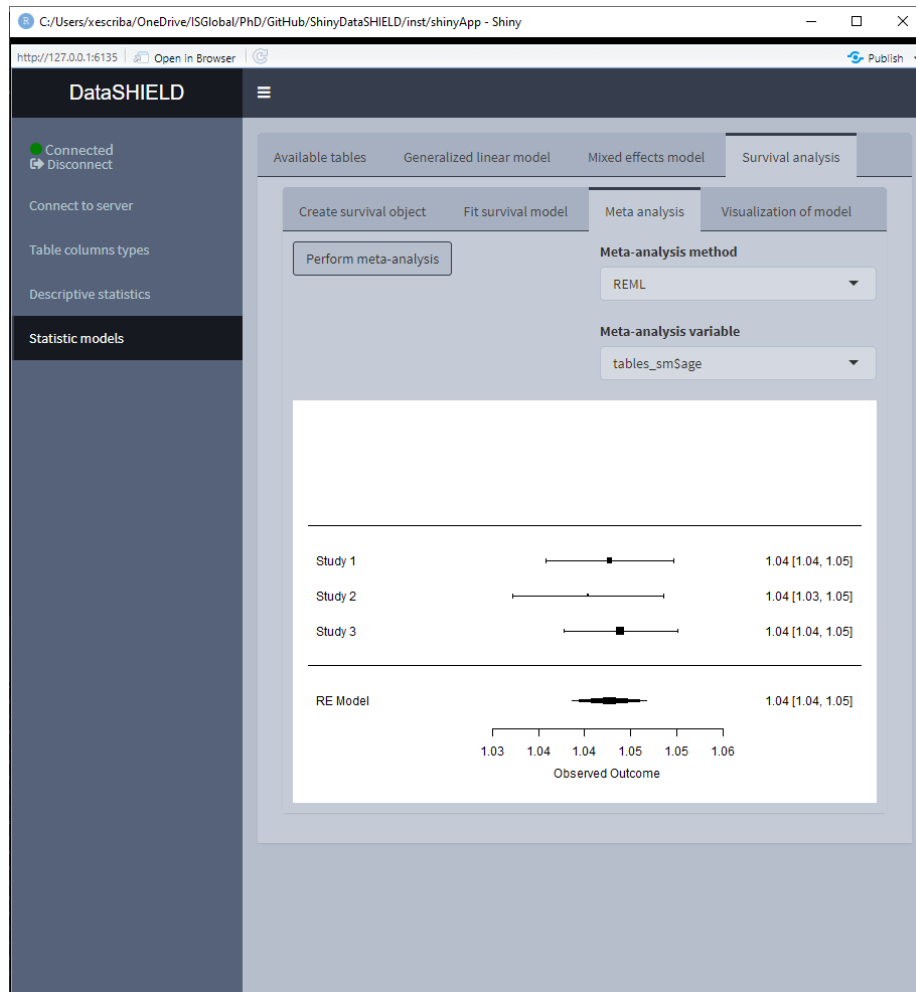
```
survival_object~tables_sm$age+tables_sm$female
```

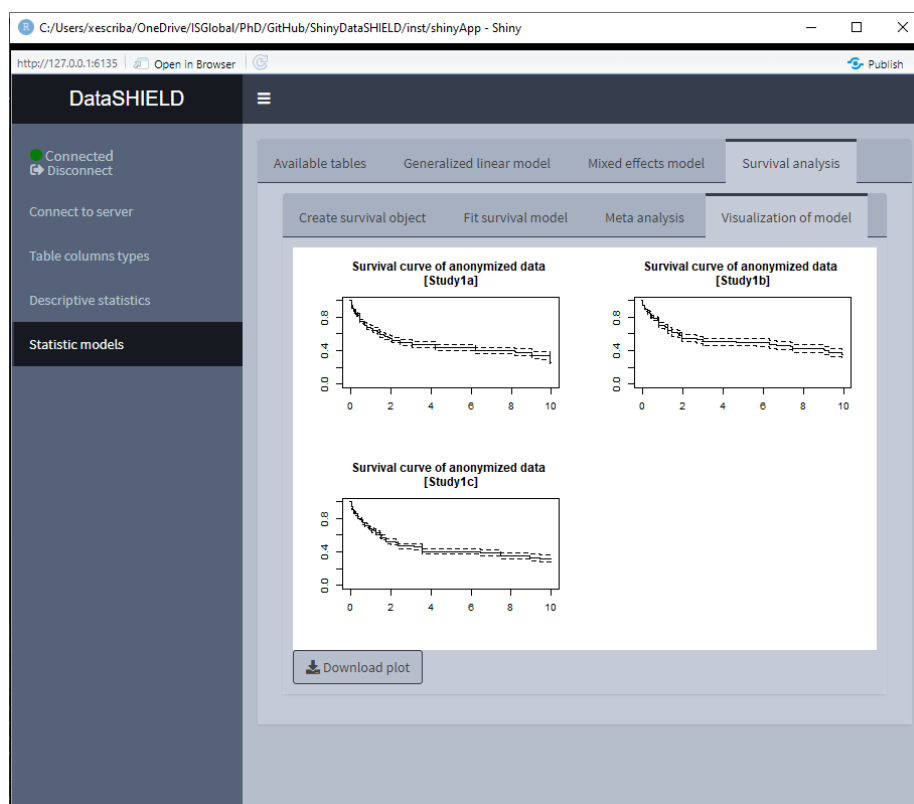
The screenshot shows the DataSHIELD web application interface. The browser address bar indicates the URL is `http://127.0.0.1:6135`. The application has a dark sidebar on the left with the following menu items: **DataSHIELD**, **Connected** (with a Disconnect button), **Connect to server**, **Table columns types**, **Descriptive statistics**, and **Statistic models** (which is highlighted). The main content area has a top navigation bar with tabs: **Available tables**, **Generalized linear model**, **Mixed effects model**, and **Survival analysis** (which is selected). Below this, there are sub-tabs: **Create survival object**, **Fit survival model** (which is selected), **Meta analysis**, and **Visualization of model**. The **Fit survival model** sub-tab contains a **Formula:** input field with the text `survival_object~tables_sm$age+tables_sm$female`, a **Run cox survival model** button, a **Toggle variables table** button, and a **Formula input help** button. Below the formula field is a **Select study server:** dropdown menu showing **Study1a**. There is also a **Show 10 entries** dropdown and a **Search:** input field. A table displays the results of the fit survival model:

	coef	exp(coef)	se(coef)	z	Pr(> z )
tables_sm\$age	0.0418	1.0427	0.0035	11.8415	0
tables_sm\$female1	-0.6452	0.5245	0.1005	-6.4181	0

Below the table, it says "Showing 1 to 2 of 2 entries". There are **Previous** and **Next** buttons, with the number **1** in a box between them. At the bottom left of the main content area is a **Download survival results** button.

If a model is fitted, the left two tabs can be accessed, this two tabs correspond to two possible visualization options of the survival analysis. The ‘Meta analysis’ plots a forestplot of a selected model variable using different meta-analysis methods. The ‘Visualization of model’ plots the survival curves for each study server.





### 3.5 Genomics

RESOURCES USED TO DEMO THIS SECTION

From <https://opal-demo.obiba.org/> :

Analysis with BioConductor

STUDY

RESOURCE

PROFILE

Study1

RSRC.brge

omics

Study1

RSRC.brge\_vcf

omics

Analysis with PLINK

STUDY

RESOURCE

PROFILE

Study1

RSRC.brge\_plink

omics

Inside the genomics tab of dsOmicshiny there are two subtabs, one to perform analysis using BioConductor methods and another to perform analysis using PLINK methods.

### 3.5.1 Analysis with BioConductor

To perform non-disclosive genomic analysis using BioConductor methodologies, the user has to input a VCF resource with a covariates resource (table) on the same study.

When performing this kind of analysis, as explained on the Data Entry section, only one study server can be used.

The Analysis with BioConductor has two sub-tabs, the first one corresponds to the GWAS, and as the name implies is used to perform a GWAS (Genome wide association study) non-disclosive analysis on the loaded data. There is a selector for the condition and the covariates to adjusted for. The fitted model is:  $\text{snp} \sim \text{condition} + \text{covar1} + \dots + \text{covarN}$ . The results of the model appear on a table below the selectors. The download button will prompt a system window to select where to store the shown table, it will save it as a \*.csv. The second subtab is to display a Manhattan plot of the GWAS results. The download plot button saves the shown figure as a \*.png.



Shiny application interface for DataSHIELD, showing the GWAS results table.

Available resources: GWAS, Manhattan Plot

Variable: asthma

Covariable: gender, smoke

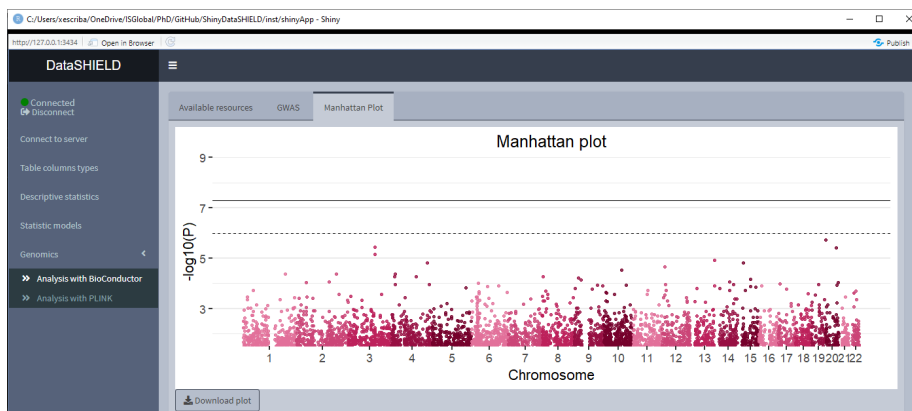
Perform GWAS

Show 10 entries

rs	chr	pos	n.obs	freq	p.value	Est	Est.SE
rs12008773	X	64955287	2222	0.0126	0	1.3784	0.2635
rs2267914	20	1858988	2306	0.1041	0	0.4966	0.1041
rs7153	3	122865987	2303	0.2555	0	0.3408	0.0736
rs6097326	20	51324797	2306	0.1255	0	0.4519	0.0978
rs3732410	3	122898410	2303	0.2538	0	0.3322	0.0739
rs7995146	13	111804035	2299	0.0528	0	0.6316	0.1445
rs6495788	15	27663291	2302	0.2669	0	0.3095	0.0716
rs1602679	4	167576714	2294	0.1005	0	0.4591	0.1062
rs11055608	12	13804693	2301	0.4463	0	-0.2722	0.0641
rs7098143	10	83329037	2290	0.2092	0	-0.328	0.0785

Showing 1 to 10 of 99,288 entries

Download GWAS results



### 3.5.2 Analysis with PLINK

To perform non-disclosive analysis using PLINK commands, the user has to load a SSH resource. The tab contains a field to input the PLINK command and a brief memo stating that when inputting the PLINK command to run there is no need of inputting it as `plink ...` as would be done on a terminal interface, the user has to input just the `...`; also, there is no need to put `-out` to indicate the output file.

Once the command is run, a table with the results is displayed under the command input, the download button will prompt a system window to select where to store the shown table, it will save them as a `*.csv`. A button to display the raw terminal output appears to display the user on a popup the plain text.

The screenshot shows the DataSHIELD Shiny application interface. The left sidebar contains navigation options: 'Connected', 'Disconnect', 'Connect to server', 'Genomics', 'Analysis with BioConductor', and 'Analysis with PLINK'. The main panel has three tabs: 'Available SSH resources', 'PLINK', and 'Manhattan Plot'. The 'PLINK' tab is active, displaying a 'PLINK Shell command' input field with the command: `--bfile brge --logistic --covar brge.phe --covar-name gender,age`. Below the command field, there are two notes: 'NOTE: we avoid -out to indicate the output file' and 'NOTE: No need to input plink as in a shell command'. A 'plink < >' button is present, followed by a 'can be inputted as' section with a '< >' button. There are two buttons: 'Run Shell command' and 'Show PLINK terminal output'. Below these, a 'Show 10 entries' dropdown and a 'Search:' input field are visible. A table of results is displayed with columns: CHR, SNP, BP, A1, TEST, NMISS, OR, STAT, and P. The table shows 10 rows of data, including SNPs like MitoC3993T and MitoG4821A. At the bottom, there is a 'Showing 1 to 10 of 302,587 entries' message, a pagination bar with 'Previous', '1', '2', '3', '4', '5', '...', '30259', and 'Next', and a 'Download PLINK results' button.

CHR	SNP	BP	A1	TEST	NMISS	OR	STAT	P
0	MitoC3993T	3993	T	ADD	2286	0.7521	-1.333	0.1824
0	MitoC3993T	3993	T	gender	2286	0.7425	-3.271	0.0011
0	MitoC3993T	3993	T	age	2286	1.004	0.5648	0.5722
0	MitoG4821A	4821	A	ADD	2282	2.685	1.707	0.0879
0	MitoG4821A	4821	A	gender	2282	0.7398	-3.308	0.0009
0	MitoG4821A	4821	A	age	2282	1.003	0.4648	0.6421
0	MitoG6027A	6027	A	ADD	2307	1.864	1.919	0.055
0	MitoG6027A	6027	A	gender	2307	0.7312	-3.457	0.0005
0	MitoG6027A	6027	A	age	2307	1.003	0.5237	0.6005
0	MitoT6153C	6153	C	ADD	2308	1.016	0.071	0.9434

The screenshot shows the DataSHIELD web interface. On the left, there is a sidebar with a 'Connected' status and a 'Connect to server' button. The main area displays the 'PLINK Terminal output' in a terminal window. The output shows the PLINK version (v1.07) and date (10/Aug/2009), followed by a copyright notice for Shaun Purcell. It then lists the options in effect: --bfile brge, --logistic, --covar brge.phe, --covar-name gender,age, --noweb, and --out /tmp/ssh-7170/out. The output continues with a detailed summary of the analysis, including the number of markers (100000), individuals (2312), and SNPs (100000). It also shows the results of the missingness test (0 SNPs failed). On the right side of the terminal output, there is a table with two columns: 'STAT' and 'P'. The table contains several rows of data, including values like -1.333, 0.1824, -3.271, 0.0011, 0.5648, 0.5722, 1.707, 0.0879, -3.308, 0.0009, 0.4648, 0.6421, 1.919, 0.055, -3.457, 0.0005, 0.5237, 0.6005, 0.071, 0.9434, and 30259. A 'Next' button is visible at the bottom right of the table.

```

@-----@
|   PLINK!   |   v1.07   |  10/Aug/2009  |
|-----|
| (C) 2009 Shaun Purcell, GNU General Public License, v2 |
|-----|
| For documentation, citation & bug-report instructions: |
|   http://pngu.mgh.harvard.edu/purcell/plink/             |
|-----|
@-----@

Skipping web check... [ --noweb ]
Writing this text to log file [ /tmp/ssh-7170/out.log ]
Analysis started: Wed Jun  9 14:01:54 2021

Options in effect:
--bfile brge
--logistic
--covar brge.phe
--covar-name gender,age
--noweb
--out /tmp/ssh-7170/out

Reading map (extended format) from [ brge.bim ]
100000 markers to be included from [ brge.bim ]
Reading pedigree information from [ brge.fam ]
2312 individuals read from [ brge.fam ]
2312 individuals with nonmissing phenotypes
Assuming a disease phenotype (1=unaff, 2=aff, 0=miss)
Missing phenotype value is also -9
725 cases, 1587 controls and 0 missing
1097 males, 1215 females, and 0 of unspecified sex
Reading genotype bitfile from [ brge.bed ]
Detected that binary PED file is v1.00 SNP-major mode
Reading 6 covariates from [ brge.phe ] with nonmissing values for 2199 indivi
Selected subset of 2 from 6 covariates
For these, nonmissing covariate values for 2312 individuals
Before frequency and genotyping pruning, there are 100000 SNPs
2312 founders and 0 non-founders found
6009 heterozygous haploid genotypes; set to missing
Writing list of heterozygous haploid genotypes to [ /tmp/ssh-7170/out.hh ]
7 SNPs with no founder genotypes observed
Warning, MAF set to 0 for these SNPs (see --nonfounders)
Writing list of these SNPs to [ /tmp/ssh-7170/out.nof ]
Total genotyping rate in remaining individuals is 0.994408
0 SNPs failed missingness test ( GENO > 1 )

```

STAT	P
-1.333	0.1824
-3.271	0.0011
0.5648	0.5722
1.707	0.0879
-3.308	0.0009
0.4648	0.6421
1.919	0.055
-3.457	0.0005
0.5237	0.6005
0.071	0.9434
30259	Next

There's also a sub-tab to show a Manhattan plot with the results obtained. The download plot button saves the shown figure as a \*.png.



### 3.6 Omics

#### RESOURCES USED TO DEMO THIS SECTION

From <https://opal-demo.obiba.org/> :

LIMMA

STUDY

RESOURCE

PROFILE

Study1

RSRC.GSE80970

omics

DESeq and edge R: To be determined

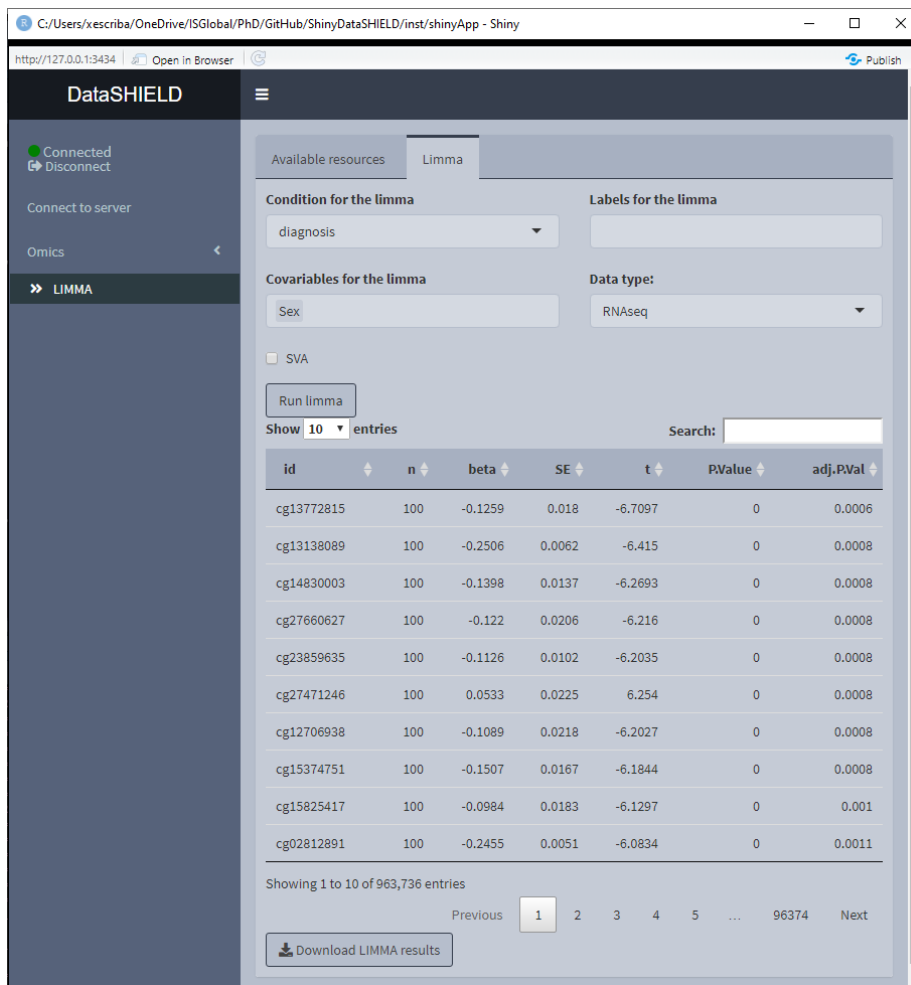
On the Omics tab there are three different subtabs for different methodologies to perform non-disclosive analysis: limma, DESeq and edgeR. The resources that can be used are ExpressionSets and RangeSummarizedExperiments. If the resources are pooled the user has to input each one in a different study on the data entry.

### 3.6.1 LIMMA

The limma non-disclosive analysis tab contains two selectors to select the condition and covariables of the analysis (resulting formula is:  $\text{feature} \sim \text{condition} + \text{covar1} + \dots + \text{covarN}$ ), there's also a selector to input the annotations columns desired on the output of the analysis. Finally, there's a selector to indicate the type of data that is being studied, whether is microarray or RNAseq. There's a selector to choose to do a surrogate variable analysis.

Once the analysis is performed a table with the results is displayed below the parameter selectors. The download button will prompt a system window to select where to store the shown table, it will save them as a \*.csv.

If the analysis is being performed using a pooled dataset, the shown table corresponds to all the pooled data.



The screenshot shows the DataSHIELD web interface with the LIMMA analysis tab selected. The interface includes a sidebar with navigation options (Omics, LIMMA) and a main panel with analysis parameters and results.

**Analysis Parameters:**

- Condition for the limma:** diagnosis
- Covariables for the limma:** Sex
- Data type:** RNAseq
- Run limma:** [button]
- Search:** [input field]

**Results Table:**

id	n	beta	SE	t	PValue	adj.PVal
cg13772815	100	-0.1259	0.018	-6.7097	0	0.0006
cg13138089	100	-0.2506	0.0062	-6.415	0	0.0008
cg14830003	100	-0.1398	0.0137	-6.2693	0	0.0008
cg27660627	100	-0.122	0.0206	-6.216	0	0.0008
cg23859635	100	-0.1126	0.0102	-6.2035	0	0.0008
cg27471246	100	0.0533	0.0225	6.254	0	0.0008
cg12706938	100	-0.1089	0.0218	-6.2027	0	0.0008
cg15374751	100	-0.1507	0.0167	-6.1844	0	0.0008
cg15825417	100	-0.0984	0.0183	-6.1297	0	0.001
cg02812891	100	-0.2455	0.0051	-6.0834	0	0.0011

Showing 1 to 10 of 963,736 entries

Navigation: Previous 1 2 3 4 5 ... 96374 Next

[Download LIMMA results]

### **3.6.2 DESeq**

To be implemented.

### **3.6.3 edgeR**

To be implemented.

## Chapter 4

# Developers Guide

Along this section, documentation for future developers and maintainers of ShinyDataSHIELD is provided. It contains information about how the whole Shiny application is structured, all the different scripts that contains, flowcharts of the different files and information on how to extend the capabilities of ShinyDataSHIELD to new types of resources as well as new methodologies.

---

### Observation

---

Please read this documentation with the actual source code on the side for easier understanding.

---

## 4.1 File structure of ShinyDataSHIELD

Typically Shiny applications are contained in a single file or two files, since the typical structure of a Shiny application is to have a **server** function and a **ui** function that can be on the same file or split for larger applications. On ShinyDataSHIELD the **server** function has been split into different scripts where all of them contains the code of a certain block of the application. It has been done this way to not have a really long **server** file that is difficult to navigate and debug. There is no need to split the **ui** file into different scripts since it only contains the graphical declarations of the applications and is really easy to update and navigate.

The different scripts that compose the whole ShinyDataSHIELD are the following:

- **ui.R**
- **server.R**, composed of the following scripts:
  - **connection.R**
  - **descriptive\_stats.R**

```

- download_handlers.R
- genomics.R
- omics.R
- plot_renders.R
- statistic_models.R
- table_renders.R
- table_columns.R

```

The file `server.R` exists to source the different files and it also includes some small functionalities.

Now a file per file explanation will be given with flowcharts (when needed), remarkable bits of code explanations and general remarks. Also, details on how to implement new functionalities will be given when needed.

#### 4.1.1 ui.R

Inside this file there are all the declarations of how the graphical user interface (GUI) will look like.

First, it contains a declaration of all the libraries that have to be loaded for the application to run. The libraries are the following: `DSI`, `DSOpal`, `dsBaseClient`, `dsOmicsClient`, `shinydashboard`, `shiny`, `shinyalert`, `DT`, `data.table`, `shinyjs`, `shinyBS`, `shinycssloaders`, `shinyWidgets`, `stringr`).

The next piece of code found

```

jscode <- '
$(document).keyup(function(event) {
  if ($("#password1").is(":focus") && (event.keyCode == 13)) {
    $("#connect_server1").click();
  };
  if ($("#pat1").is(":focus") && (event.keyCode == 13)) {
    $("#connect_server1").click();
  }
});
'
```

Is a JavaScript declaration that reads as: When the `#password1` item (corresponds to the text input of the password on the data entry tab) is active (the user is writing in it) and the “Intro” key is pressed, trigger the `#connect_server1` item (corresponds to the “Connect” button on the GUI). That provides the user the typical experience of inputting the login credentials and pressing “Intro” to log in.

It’s important noting that this is only the declaration of a string with the code inside, to actually make use of it, there is the line 58 of this same file that actually implements it.



```
tags$head(tags$script(HTML(jscode)))
```

Two more pieces of JavaScript and CSS are found

```
jscode_tab <- "
shinyjs.disableTab = function(name) {
  var tab = $('<div>.nav li a[data-value=' + name + ']'</div>');
  tab.bind('click.tab', function(e) {
    e.preventDefault();
    return false;
  });
  tab.addClass('disabled');
}

shinyjs.enableTab = function(name) {
  var tab = $('<div>.nav li a[data-value=' + name + ']'</div>');
  tab.unbind('click.tab');
  tab.removeClass('disabled');
}
"

css_tab <- "
.nav li a.disabled {
  background-color: #aaa !important;
  color: #333 !important;
  cursor: not-allowed !important;
  border-color: #aaa !important;
}"
```

The CSS is just for aesthetics, the JS however is to introduce the functionality of enabling and disabling panels of the web application, this is used as `js$disableTab()` and `js$enableTab()` along the application. Those two scripts are integrated to the application when building the `dashboardBody` by using this

```
useShinyjs(),
extendShinyjs(text = jscode_tab, functions = c("enableTab", "disableTab")),
inlineCSS(css_tab)
```

There are a some functions used in this file that are worth mentioning:

- `hidden()`: From the `shinyjs` library. The elements wrapped inside of this function will not be rendered by default, they have to be toggled from the server side. Example: A GUI element that needs to be displayed only when a certain condition is met.
- `withSpinner()`: From the `shinycssloaders` library. The elements wrapped inside of this function will be displayed as a “loading spinner”

when they are being processed. This is used to wrap figure displays. Example: A plot that is being rendered, it's better for the user experience to see a "loading spinner" so that it knows something is being processed rather than just staring at a blank screen waiting for something to happen.

- `bsModal()`: From the `shinyBS` library. It's used to prompt pop-ups to the user. Example: By the click of a button you want to render a pop-up to the application with a figure of an histogram of a selected column of a table.
- `conditionalPanel()`: From the `shiny` library. It is useful to display certain elements on the GUI regarding a condition is met or not, here is used to display the user / password fields or the personal access token (PAT) fields by checking the state of the selector. Note that the condition has to be written using JavaScript, that's why it looks like `"input.pat_switch1 == true"` rather than the typical R Shiny `input$pat_switch1 == TRUE`.

In order to declare the elements when the user wants to add another server some R tricks are used, they are described and coded on the `connection.R` file.

The rest of this file is your average Shiny functions and declarations, read the official documentation for any doubts. Please note that `ShinyDataSHIELD` uses `shinydashboard` to improve the looks and user experience, for any doubts regarding that please read it's documentation.

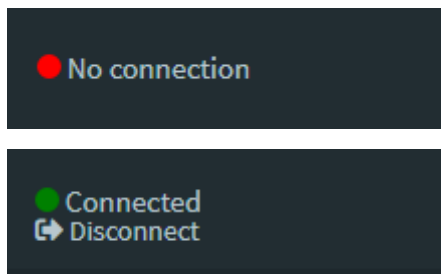
#### 4.1.2 server.R

The server file is divided into the following blocks.

- Declaration of reactiveValues: As a code practice measure, all the variables that have to be used in different parts of the code (Example: Table that contains the information about the loaded resources, has to be written when loading the data and afterwards to check whether a resource has been loaded or not) are reactive values. The only occasions where there are "regular" variables are inside functions that use variables as placeholders to be used only inside of that function (Example: Storing the results of a middle ground operation to be later used inside the same function to perform the final analysis, whose results will be saved on a reactive value variable). Developers used to lower level languages can see this as `public` and `private` variables.
- Sourcing of scripts: Sourcing all the different scripts that actually make up `server.R`. As said before this is done this way to have a more structured application where each script takes care of a certain block of the application.
- Disabling of all the tabs except the server connector: By default all the tabs are visible on Shiny, in order to provide a good user experience all are disabled at the launch of the application using `js$disableTab()`, once tables or resources are loaded into the study servers tabs are enabled (only

the ones that makes sense, if the user loads a Table, only the tabs to interacts with tables will be enabled).

- Function declaration: Declaration of a function that given a column of a data table will truncate the decimal places to 4, it's used when rendering tables to not have tables with 9 decimals that look hideous.
- Functions to manage the “Connected” / “No connection” display. It's a bunch of logic and CSS to just control a small element of the GUI. Basically if the variable `connection$active` is `TRUE` the GUI will show “Connected” next to a green dot with a “Disconnect” button, otherwise it will display “No connection” next to a red dot. When the button “Disconnect” is pressed, the function to log out of the server is triggered and the `connection$active` variable is set to false.
- Stop function: This declaration is left for future developers. When having trouble on a certain spot, add a stop button using `actionButton("stop", "stop")`, press it on the GUI to stop the execution and perform the required debugging.



The scripts sourced for by the `server.R` are the following:

#### 4.1.2.1 `connection.R`

This is probably the most important script of the whole application, as it's the one that is responsible for loading the data in order to ensure that the application capabilities can be extended in the future painlessly (modular).

Inside this script there are five different sections that are triggered by different actions:

1. Creation of GUI for the new server tabs
2. Creation of `observeEvents` for all the different server tab elements.
  1. URL builder to display selected items on a browser
  2. Connection to the server to obtain the projects and resources. Triggered by the button with label “connect\_server”.
  3. Get tables / resources from the selected project. Triggered everytime the selector with label “project\_selected” is changed.
  4. Add a study. Triggered by the button with label “add\_server”.
3. Remove a server tab. Triggered by the button with label “remove”.
4. Remove a study item Triggered by the button with label “remove\_item”.

5. Load the selected studies to the study servers. Triggered by the button with label “connect\_selected”.

The first element to explain is the creation of the new server tabs (Point 1, `observeEvent(input$add, {})`). It's just a matter of noting two things to understand it easily, 1) The use of a reactive value `tabIndex()` which returns a integer (initialized at 1), this integer corresponds to the tab being created (hence it's updated at the top of the call); 2) The rest of the call is an `appendTab()` that adds a new tab on the element id “`tabset1`” with the exact structure of `ui.R` but changing the element IDs using the reactive value so that all the buttons/input fields are numbered according to the tab they are located on. When removing a server tab (`observeEvent(input$remove, {})`) the tab itself is removed using `removeTab()` and the reactive value is actualized.

The creation of `observeEvents` for all the different server tab elements (Subitems of point 2) is done using a small trick. `max_servers` number of `observeEvents` are created (using `lapply`) for the different functionalities so that all the server tabs are functional, this integer variable is defined on the script `server.R`, if more servers than the default (10) are required just update the definition of the variable and relaunch the application.

The last part of the script, loads all the selected tables and resources to the selected study servers. It does everything needed to each particular type of resource, that means converting them to R objects or to tables depending on what they are.

When loading the selected resources or tables into the study servers, the table `available_tables` is created. The name is a little bit confusing since it actually contains the information about tables and resources, the developer apologizes as this variable was set at the beginning of the development and has not been updated. Nevertheless, it's an important variable of the application, the structure of this table is the following.

Column	Description
<code>name</code>	Name of the object ( <code>project.name</code> )
<code>server_index</code>	Index of the study server that contains the table/resource
<code>server</code>	Name of the study server
<code>type_resource</code>	Type of the resource

The Opal server can host different types of resources, to name a few there are `ExpressionSet`, `RangedSummarizedExperiment` and `SQLResourceClient`. Each type of resource needs a special treatment to be used, for example `SQLResourceClient` resources are plain tables, so they need to be converted to tables on the study server to use them. Currently the following resource types are supported by ShinyDataSHIELD.

Resource type	Treatment	Name of the resource type on available_tables
TidyFileResourceClient	<code>as.resource.data.frame(resource)- Append .t to the name</code>	table
SQLResourceClient	Append .r to the name	ssh
SshResourceClient	Append .r to the name	r_obj_vcf
GdsGenotypeReader	<code>as.resource.object(resource) - Append .r to the name</code>	r_obj_eset
ExpressionSet	<code>- as.resource.object(resource)- Append .r to the name</code>	r_obj_rse
RangedSummarizedExperiment	<code>as.resource.object(resource)- Append .r to the name</code>	r_obj
Any other resource type	<code>- as.resource.object(resource)- Append .r to the name</code>	

.r and .t are appended to the resources to allow a resource and a table on the same project to have the same names and not crash the Shiny application.

Now, let's look at some examples to add new resource types on the `connection.R` file. There are different cases for the treatment that the new resource requires.

- Resources that just need to be loaded with no further action performed to them (same treatment as SSH connections). Add another `else if` statement after line 303. Example: New resource called `Simple_resource`

```
else if ("Simple_resource" %in% resource_type){
  # Update available_tables list with the new resource type name
  lists$available_tables <- rbind(lists$available_tables, c(name = name, server_index = server_index,
                                                            server = resources$study_server))
}
```

- Resources that need to be converted into R objects (`datashield.assign.expr(conns, symbol = "methy", expr = quote(as.resource.object(res)))`) and nothing else. Will work out of the box (the `type_resource` column of the `lists$available_tables` table will read `r_obj`).
- Resources that need to be converted into R objects (`datashield.assign.expr(conns, symbol = "methy", expr = quote(as.resource.object(res)))`) and be further processed. Add another `else if` statement after line 320. Example: A new type of resource called `special_resource` that contains some variable names that are desired to be saved on a variable to feed a list on the GUI.

```

else if("special_resource" %in% resource_type) {
  # Update available_tables list with the new resource type name
  lists$available_tables <- rbind(lists$available_tables, c(name = name, server_index =
                                                             server = resource_type))

  # Perform the needed actions for this resource
  [...]
}

```

Finally, once all the connections have been successful, and all the selected tables and resources are loaded, the tabs that make use of the loaded objects are enabled by using (table examples)

```

if(any(unique(lists$available_tables$type_resource) %in% c("table"))){
  show(selector = "ul li:eq(2)")
}

```

There are many if that checks for type of resources and enables tabs if present, on the previous example the second tab `ul li:eq(2)` (there is no way of referring them by ID as far as I know to perform this action) is enabled because it contains a module that works with tables.

If a new type of resource is implemented, add after line 353 (tab 10 is just as example)

```

if(any(unique(lists$available_tables$type_resource) %in% c("new_resource"))){
  show(selector = "ul li:eq(10)")
}

```

Also update this part if a new module is added, make sure to enable the tab only when the resources that the module use are present on the `lists$available_tables`.

### 4.1.3 Structure of the modules

A common structure is followed for all the different modules, this refers to the general structure of `descriptive_stats.R`, `statistics_models.R`, `genomics.R`, `omics.R` and `table_columns.R`.

Before describing the internal structure of the modules, let's briefly describe the GUI structure, which is also common between them. The tabs are filled with a tab box, the first element is always a table with the available tables / resources for that module. For example, the Omics module only displays the resources of type RSE or eSet. The other tabs are disabled by default and can only be accessed once the user has selected which resource to use. Now let's talk about how to accomplish all of this.

At the beginning of all the modules there is an `observeEvent` that is triggered when the user selects an item from the table. The structure of this is the following

```

observeEvent(input$table, {
  if(length(input$table_rows_selected) > 0){ # Check if the user has selected any row
    different_study_server <- TRUE # On this example we are checking that everything selected is
    same_cols <- TRUE # On this example we are checking tables to be pooled, so we are checking i
    if(length(input$table_rows_selected) > 1){ # If more than one table is selected the checks h
      same_cols <- all(lapply(input$tbl_rows_selected, function(i){
        res<-all(match(lists$resource_variables[[as.character(lists$available_tables[type_resource
          lists$resource_variables[[as.character(lists$available_tables[type_resource
        if(is.na(res)){FALSE} else{res}
      })))
      different_study_server <- nrow(unique(lists$available_tables[input$table_rows_selected,3]))
      length(input$table_rows_selected)
    }
    if(same_cols & different_study_server){ # If both tests are OK, remove the "resource_lim" ob
      datashield.rm(connection$conns, "resource_lim")
      for(i in input$table_rows_selected){
        lists$available_tables[type_resource %in% c("table")][i,2]
        # Then assign the selected tables to a new variable on the study servers called "resource
        datashield.assign.expr(connection$conns[as.numeric(lists$available_tables[type_resource %
      ]
      # Enable the analysis tab and update the GUI to display it
      js$enableTab("tab_of_analysis")
      updateTabsetPanel(session, "id",
        selected = "tab_of_analysis")
    }
    else{ # If the tests fail, display an error message
      shinyalert("Oops!",
        if(!same_cols){
          "Selected resources do not share the same columns, can't pool unequal resource
        }else{
          "Selected resources are not on different study servers, can't pool resources o
        }
        , type = "error")
      # Make sure analysis tabs are disabled and the GUI shows the selection tab, this is importa
      js$disableTab("tab_of_analysis")
      updateTabsetPanel(session, "id",
        selected = "table_selection")
    }
  }
})

```

This example can be extended to the developers needs, but as a structure example is more than enough. Please read the source code for the available modules if extra examples are needed.

The body of the modules correspond to whatever is needed on that module, let

that be some `observeEvent` for buttons of the analysis tab, some `renderUI` for dynamic selectors or anything other that the module needs.

The bottom of the modules is also shared, they contain an `observe` clause that is triggered when the tab is selected, it has the following structure

```
observe({
  if(input$tabs == "id") { # The ID here corresponds to the tabname declare on the ui.
    tables_available <- lists$available_tables[type_resource %in% c("table")] # Input i
    if(length(lists$resource_variables) == 0){
      withProgress(message = "Reading column names from available tables", value = 0, .
        for(i in 1:nrow(tables_available)){ # In this example we are reading table col
          lists$table_columns[[as.character(tables_available[i,1])]] <- ds.colnames(as
          incProgress(i/nrow(tables_available))
        }
      })
    }
    # Finally we render the table with the available tables for this module so the use
    output$available_tables_sm <- renderUI({
      dataTableOutput("available_tables")
    })
  }
})
```

Example of the code for the `table_render.R` regarding the selection table

```
output$available_tables <- renderDT(
  lists$available_tables[type_resource == "table"], options=list(columnDefs = list(list
                                                                    paging = FALSE, search
  )
)
```

Now let's take a look at the scripts that are used by all the modules, their use is to render tables, figures and handle the downloads (figures + table downloads)

#### 4.1.3.1 table\_renderers.R

This script creates the displays of all the tables of ShinyDataSHIELD, it uses the DT package to do so. Besides the `descriptive_summary` table, all the other tables just render results from other functions.

There are some things to point of this script:

- As can be seen in `descriptive_summary` table, you can actually perform operations inside of a `renderDT` function and display the result of them.
- The most used options for the tables aesthetics are the following

```
options=list(columnDefs = list(list(visible=FALSE, targets=c(0))),
             paging = FALSE, searching = FALSE)
```

This prevents the `rownames` column to be displayed (usually it just contains the



numeration of rows 1...N, be aware sometimes it's of interest to see this column) and eliminates the paging and searching functionalities of the table. For small tables it makes sense to not show that but on big tables those options are set to `TRUE`, as it's very useful to have a search box on them.

- The tables that display numerical columns (mixed or not with non-numerical columns) are actually passed through the `format_num` function (defined on `server.R`) so the displayed table has only four decimals but the actual table (the one that can be saved) has all the decimals. This is done using the following code

```
as.data.table(lapply(as.data.table(vcf_results$result_table_gwas$server1), format_num))
```

This will pass each column to the function and if it's numerical the decimals will be cut to 4.

- The table output structure of the LIMMA results look different than the others, this is because when performing a LIMMA with pooled resources it returns one table for each study, what is being done is just binding them to display to the user all the obtained results.

There is a concrete render that needs a special mention on this documentation, that is the `column_types_table`, which uses the CellEdit JavaScript plugin to enable drop down menus when editing a table. Let's see what is being done

```
tab <- datatable(
  table_to_be_modified, editable = "cell", callback = # The callback needs to be updated to
  JS(
    "function onUpdate(updatedCell, updatedRow, oldValue){",
    "Shiny.onInputChange('jsValue', [updatedCell.index(), updatedCell.data()]);", # The res
    "}",
    "table.MakeCellsEditable({",
    "  onUpdate: onUpdate,",
    "  inputCss: 'my-input-class'",
    "  columns: [2]",
    "  confirmationButton: {",
    "    confirmCss: 'my-confirm-class'",
    "    cancelCss: 'my-cancel-class'",
    "  }",
    "  inputTypes: [",
    "    {",
    "      column: 2",
    "      type: 'list'",
    "      options: [",
    "        {value: 'numeric', display: 'numeric'}", # Update this lines to declare the c
    "        {value: 'factor', display: 'factor'}",
    "        {value: 'character', display: 'character'}",
    "      ]",
    "    }",
    "  ]",
    "}
```

```

      "    }",
      "  ]",
      "});");
    options = list(pageLength = nrow(table_to_be_modified))
  )
  path <- "../..//www/" # folder containing the files dataTables.cellEdit.js
                        # and dataTables.cellEdit.css, they are already included on ShinyDataSHIELD, so th
  dep <- htmltools::htmlDependency(
    "CellEdit", "1.0.19", path,
    script = "dataTables.cellEdit.js", stylesheet = "dataTables.cellEdit.css")
  tab$dependencies <- c(tab$dependencies, list(dep))

```

Example of what to include on the module script to update the table

```

proxy = dataTableProxy('a') # No need to change this
observeEvent(input$jsValue, { # As stated above, the trigger is actually the value def
  change <- data.table(input$jsValue)
  row <- as.numeric(change[1,1]) + 1
  column <- as.numeric(change[2,1])
  value <- as.character(change[4,1])

  table_to_be_modified[row, column] <- value
  replaceData(proxy, table_to_be_modified, resetPaging = FALSE)
}

```

#### 4.1.3.2 plot\_renders.R

There are two types of plots on ShinyDataSHIELD, the ones created with the base function plot and the ones created with the ggplot library. In order to later recover the plots to download them, they actually have a different structure.

- Base plot structure:

```

output$random_plot <- renderPlot({
  plots$random_plot <- function(){
    function_that_generates_the_plot_using_base_package(arguments)
  }
  plots$random_plot()
})

```

For the base plots, a function is declared that returns the plot and is called to generate the plot to the GUI.

- Ggplot structure:

```

output$manhattan <- renderPlot({
  plots$ggplot <- function_that_generates_the_plot_using_ggplot2_package(arguments)
  plots$ggplot

```

```
})
```

In this case the plot is saved, ggplot will generate a plot variable that can be called to render the plot.

On this script there are two plots that are inside a `renderCachedPlot` function instead of a `renderPlot` because they take really long to calculate and it's better to cache them.

Inside of the `renderPlot` function some other code can be put, such as toggles to GUI elements or `tryCatch()` functions.

#### 4.1.3.3 download\_handlers.R

In this script everything related to downloading plots and tables is found. There are basically three types of structures

- Table downloader: To download a \*.csv. Structure:

```
output$table_download <- downloadHandler(
  filename = "table.csv",
  content = function(file) {
    write.csv(
      variable_that_contains_table
      , file, row.names = FALSE)
  }
)
```

The `row.names = FALSE` argument may not be needed in tables where the row names are important.

- Base plot downloader: To download a \*.png. Structure:

```
output$base_plot_download <- downloadHandler(
  filename = "base_plot.png",
  content = function(file) {
    png(file = file)
    plots$base_plot()
    dev.off()
  }
)
```

Basically this calls the previously declared function and captures the plot into a \*.png.

- GGplot downloader: To download a \*.png. Structure:

```
output$ggplot_download <- downloadHandler(
  filename = "ggplot.png",
  content = function(file) {
```

```
    ggsave(file, plot = last_plot())
  }
)
```

When using `ggplot`, the function `last_plot()` renders the last plot rendered by `ggplot`. This only has one inconvenient, that is when you are downloading a plot that takes a while to render, the application doesn't show the save window dialog until it has rendered again. This should be addressed in the future as it really halts the user experience.

#### 4.1.4 How to add a new block

To add a new block to `ShinyDataSHIELD`, the developer has to create a new `*.R` script inside the `inst/shinyApp/` folder of the project and give it a descriptive name of the function that it will perform.

So the Shiny application actually sees it, the `server.R` needs to be updated and source the new file. Example: New block called `new_analysis.R`, the update to the `server.R` will be

```
source("new_analysis.R", local = TRUE)
```

Afterwards, the `ui.R` can be updated by defining how the new block will be presented to the user. The `sidebarMenu` function needs to be updated so that the new tab appears on the sidebar of the application, follow the structure of the other tabs. Afterwards update the `dashboardBody` function by defining all the different elements of the new tab, follow the structure of the other available tabs to follow the general design lines, all the functions that need to be used here are standard Shiny functions mostly and there's plenty of documentation and examples available online, when in doubt just try to copy an already implemented structure.

Now the user can focus on the types of files that will feed this new block, if it's a table there's no need to worry, if it's a resource that is not implemented the `connection.R` needs to be updated. Read the above documentation for guidance on the changes that need to be done for new resources types.

Once the GUI is setup and the table / resource that this block will use is setup, the backend for this block can be built on the `new_analysis.R` file. Include on that file all the required `renderUI()` functions and steps to process the file and analyze it. Probably a new variable will be required to hold the results, update the `server.R` header and include a new `reactiveValues()` declaration for the new block.

If the new block requires to display tables or figures, update the `table_renders.R` and `plot_renders.R` following the given examples on their sections of the documentation. Make sure to include the download buttons for them on the `download_handlers.R`.

If there is some part of the code that takes some time to process, there's the option of wrapping it inside the `withProgress()` function in order to display a loading animation to the GUI to alert the user that something is being processed.

Make sure to include the custom implementation of the header and footer functions for the module that have been presented before.

When developing a new block there will probably be many problems occurring, in order to debug a Shiny application there is the `browser()` function, if the developer is getting some sort of error at X line of the script, just write `browser()` on the line adobe of the error, the execution will be stopped at that point and the developer can interact with all the available variables of the environment through the RStudio console, usually running the line that is giving an error on the console will provide enough information to kill the bug. If the line breaking is a function call it is advisable to type the variables that are being passed into the function on the console, that way the developer can see what exactly is being passed and can see that some argument is `NULL` when it shouldn't or it's a character when it should be a number, those are quite common problems.

When a new block is developed and integrated into ShinyDataSHIELD, please conclude it by updating this documentation and the user guide with a brief explanation of the new block and some remarks of the most interesting bits of it.